Teradata
a division of NCR

Teradata® RDBMS
Database Design

V2R5.0

B035-1094-122A
May 2003

The product described in this book is a licensed product of NCR Corporation.

BYNET is an NCR trademark registered in the U.S. Patent and Trademark Office.
CICS, CICS/400, CICS/600, CICS/ESA, CICS/MVS, CICSPLEX, CICSVIEW, CICS/VSE, DB2, DFSMS/MVS, DFSMS/VM, IBM, NQS/MVS, OPERATING SYSTEM/2, OS/2, PS/2, MVS, QMS, RACF, SQL/400, VM/ESA, and VTAM are trademarks or registered trademarks of International Business Machines Corporation in the U. S. and other countries.
DEC, DECNET, MICROVAX, VAX and VMS are registered trademarks of Digital Equipment Corporation.
HEWLETT-PACKARD, HP, HP BRIO, HP BRIO PC, and HP-UX are registered trademarks of Hewlett-Packard Co.
KBMS is a trademark of Trinzic Corporation.
INTERTEST is a registered trademark of Computer Associates International, Inc.
MICROSOFT, MS-DOS, MSN, The Microsoft Network, MULTIPLAN, SQLWINDOWS, WIN32, WINDOWS, WINDOWS 2000, and WINDOWS NT are trademarks or registered trademarks of Microsoft Corporation.
SAS, SAS/C, SAS/CALC, SAS/CONNECT, and SAS/CPE are registered trademarks of SAS Institute Inc.
SOLARIS, SPARC, SUN and SUN OS are trademarks of Sun Microsystems, Inc.
TCP/IP protocol is a United States Department of Defense Standard ARPANET protocol.
TERADATA and DBC/1012 are registered trademarks of NCR International, Inc.
UNICODE is a trademark of Unicode, Inc.
UNIX is a registered trademark of The Open Group.
X and X/OPEN are registered trademarks of X/Open Company Limited.
YNET is a trademark of NCR Corporation.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN "AS-IS" BASIS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. IN NO EVENT WILL NCR CORPORATION (NCR) BE LIABLE FOR ANY INDIRECT, DIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS OR LOST SAVINGS, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The information contained in this document may contain references or cross references to features, functions, products, or services that are not announced or available in your country. Such references do not imply that NCR intends to announce such features, functions, products, or services in your country. Please consult your local NCR representative for those features, functions, products, or services available in your country.

Information contained in this document may contain technical inaccuracies or typographical errors. Information may be changed or updated without notice. NCR may also make improvements or changes in the products or services described in this information at any time without notice.

To maintain the quality of our products and services, we would like your comments on the accuracy, clarity, organization, and value of this document. Please e-mail: teradata-books@lists.ncr.com

or write:

Information Engineering
NCR Corporation
100 North Sepulveda Boulevard
El Segundo, CA 90245-4361
U.S.A.

Any comments or materials (collectively referred to as "Feedback") sent to NCR will be deemed non-confidential. NCR will have no obligation of any kind with respect to Feedback and will be free to use, reproduce, disclose, exhibit, display, transform, create derivative works of and distribute the Feedback and derivative works thereof without limitation on a royalty-free basis. Further, NCR will be free to use any ideas, concepts, know-how or techniques contained in such Feedback for any purpose whatsoever, including developing, manufacturing, or marketing products or services incorporating Feedback.

Copyright © 1996-2003, NCR Corporation
All Rights Reserved

Preface

Supported Software Release

This book supports Teradata RDBMS V2R5.0.

Changes to This Book

This book includes the following changes to support the current release:

Date	Description
May 2003	• Eliminated references to BYTE(8) data type for ROWID and differences between ROWID values for PPI and NPPI tables. • Eliminated material about performing DML statements using ROWID in the column-list, adding warnings that any such SQL statements are not valid. • Added note that ROWID can be specified only for CREATE JOIN INDEX statement and then only in the outermost SELECT statement in the join index definition. • Corrected syntax diagrams for CREATE JOIN INDEX.
December 2002	• Added material on partitioned primary indexes • Added material on multivalued compression • Updated material on table headers • Added material on surrogate keys • Added material on identity columns • Added material on PARTITION columns • Added material on ROWID columns • Added material on batch referential integrity • Added material on non-referential-integrity-enforcing referential constraints • Added chapter on database integrity • Added chapter on handling missing information • Added chapter on physical design support for tactical queries • Updated and added material on multiple system limits • Performed DR fixes and enhancement requests

Date	Description
June 2001	• Added material on column value compression • Added material on hash indexes • Added material on extending join index to support direct base table access for partial covers • Added material on single-table join index compression • Changed name of Appendix B • Revised limits on number of volatile and global temporary tables per session • Performed DR fixes and enhancement requests
September 2000	• Clarified some topics, corrected minor errors, and replaced or eliminated some figures • Added material on join index creation and maintenance costs and cost/benefit analysis • Added new appendix to support material on join index creation, maintenance, and cost/benefit analysis
June 2000	• Increased maximum number of databases/users per system • Increased maximum number of data tables per database • Increased maximum string constant size • Increased maximum number of tables that can be joined • Added new aggregate join index feature

About This Book

Purpose

This book describes database design in the Teradata environment including the context of design within data warehousing, logical design, and physical design.

Audience

This book is intended for database administrators and other technical personnel responsible for designing Teradata relational databases.

How This Book Is Organized

This book contains sixteen chapters and six appendixes:

- Chapter 1: "Designing for the Data Warehouse," introduces database design in the context of data warehouses and data marts.
- Chapter 2: "What Is Logical Data Modeling?," introduces the concept of logical database modeling and provides an overview of the events involved in designing relational databases to support the data warehouse.
- Chapter 3: "Requirements Analysis,", describes a framework for developing requirements for designing a relational database, including developing an enterprise data architecture, interviewing the customer base, and analyzing a legacy database.
- Chapter 4: "Semantic Data Modeling," reviews concepts of entity-relationship modeling.
- Chapter 5: "The Normalization Process," introduces dependency theory, the normal forms, and the normalization process with emphasis on the importance of achieving a fully normalized logical model. Selected aspects of denormalization are also described.
- Chapter 6: "Modeling for Joins and Dependency Preservation," introduces the lossy join problem and describes how to ensure lossless joins through dependency preservation.
- Chapter 7: "The Activity Transaction Modeling Process," describes a process for making the transition from logical to physical design.
- Chapter 8: "Teradata Indexes: The Basics," builds a foundation for understanding how the various Teradata indexes work.
- Chapter 9: "Primary Indexes," describes Teradata primary indexes in detail including criteria for selecting the primary index for a table and how table rows are partitioned and retrieved using the primary index.
- Chapter 10: "Secondary Indexes," describes Teradata secondary indexes in detail including criteria for selecting them and how they are used to optimize retrieval.

- Chapter 11: "Join and Hash Indexes," describes Teradata join and hash indexes: their types, their structure, and their use to optimize join and other retrieval processes.
- Chapter 12: "Designing for Database Integrity," describes the importance of defining column and table constraints for maintaining database integrity.
- Chapter 13: "Designing for Missing Information," describes the problems resulting from missing information and suggests several ways to circumvent the need to use nulls.
- Chapter 14: "Database-Level Considerations," describes various aspects of capacity planning at the database level, particularly with respect to sizing databases and their component objects.
- Chapter 15: "System-Level Considerations," describes capacity planning at the system level, including system space as well as the allocation of table space within the database.
- Chapter 16: "Design Issues for Tactical Queries," describes some of the special issues you must consider when designing your databases to support mixed workloads of both strategic and tactical queries.
- Appendix A: "Notation Conventions," explains how to read syntax diagrams as well as defining the predicate calculus and dependency theory notations used in this manual.
- Appendix B: "Teradata Database Maxima," describes the limits on various system, database, and session capabilities.
- Appendix C: "Designing With Task-Oriented Profiles," describes a method for isolating users from direct access to the database by implementing a well-conceived system of view profiles.
- Appendix D: "Summary Physical Design Scenario," describes a high level scenario of the recommended procedure for producing a physical design for a Teradata database.
- Appendix E: "Supporting Information for Join Index Performance Analysis," provides miscellaneous information to support the description of join index creation and maintenance costs in Chapter 11: "Join and Hash Indexes."
- Appendix F: "Sample Worksheet Forms," provides a set of ATM worksheets you can photocopy for use with the protocols described in Chapter 7: "The Activity Transaction Modeling Process."

Prerequisites

You should be familiar with the basic terminology and concepts of data modeling and with the Teradata software and hardware architectures.

It might be helpful to review the following books:

- *Introduction to Teradata RDBMS*
- *Teradata RDBMS Database Administration*
- *Teradata RDBMS Data Dictionary*
- *Teradata RDBMS Performance Optimization*
- *Teradata RDBMS SQL Reference*
- *Teradata RDBMS Utilities*

List of Acronyms

This book uses acronyms, which the following table lists in alphabetical order:

1NF	First Normal Form
2NF	Second Normal Form
2VL	Two-Valued Logic
3NF	Third Normal Form
3VL	Three-Valued Logic
4NF	Fourth Normal Form
4VL	Four-Valued Logic
5NF	Fifth Normal Form
ACM	Association for Computing Machinery
AJI	Aggregate Join Index
AMP	Access Module Process Access Module Processor
ASF	Archive Storage Facility
ATM	Activity Transaction Modeling
BCNF	Boyce-Codd Normal Form
BTEQ	Basic Teradata Query facility
BYNET	Banyan Network (high-speed proprietary interconnect)
CASE	Computer Aided Software Engineering
CI	Cylinder Index
CID	Cylinder Index Descriptor
CLI	Call-Level Interface
CWA	Closed World Assumption
DBA	Database Administrator
DBD	Data Block Descriptor
DDL	Data Definition Language
DIP	Database Initialization Program
DM	Dimensional Model Dimensional Modeling

DML	Data Manipulation Language
DSS	Decision Support System
DSU	Disk Storage Unit
DSW	Destination Selection Word
ELDM	Extended Logical Data Model
E-R	Entity-Relationship
ERQ	The CLIv2 End Request function
ETL	Extraction, Transforming, and Loading
FK	Foreign Key
FSP	Free Space Percent
FTS	Full Table Scan
GDO	Global Database Object
HI	Hash Index
ID	Identifier
IDNF	Inclusion Dependency Normal Form
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input/Output
IT	Information Technology
JI	Join Index
LDM	Logical Data Model
LHS	Left Hand Side
MI	Master Index
MVL	Multivalued Logic
NC	No Changes
ND	No Duplicates
NF^2	Non First Normal Form
NFNF	Non First Normal Form
NN	No Nulls
NPPI	Nonpartitioned Primary Index
NUPI	Nonunique Primary Index
NUPPI	Nonunique Partitioned Primary Index

NUSI	Nonunique Secondary Index
OLAP	On Line Analytical Processing
OLCP	On Line Complex Processing
OLTP	On Line Transaction Processing
PE	Parsing Engine
PI	Primary Index
PJ/NF	Projection-Join Normal Form
PK	Primary Key
PPI	Partitioned Primary Index
RAID	Redundant Arrays of Independent Disks
RDBMS	Relational Database Management System
RHS	Right Hand Side
RI	Referential Integrity Reference Index
RM/V1	Relational Model Version 1
RM/V2	Relational Model Version 2
ROLAP	Relational On Line Analytical Processing
SA	System-Assigned
SIGKDD	ACM Special Interest Group on Knowledge Discovery and Data Mining
SIGMOD	ACM Special Interest Group on Modification of Data
SQL	Structured Query Language
SQO	Semantic Query Optimization
TJ	Transient Journal
UA	User-Assigned
UPI	Unique Primary Index
UPPI	Unique Partitioned Primary Index
USI	Unique Secondary Index
UV	Uniqueness Value
vproc	virtual processor

Technical Information on the Web

The NCR home page (http://www.ncr.com) provides links to numerous sources of information about Teradata. Among the links provided are sites that deal with the following subjects:

- Contacting technical support
- Enrolling in customer education courses
- Ordering and downloading product documentation
- Accessing case studies of customer experiences with Teradata
- Accessing third party industry analyses of Teradata data warehousing products
- Accessing white papers
- Viewing or subscribing to various online periodicals

Contents

Preface

About This Book iii
List of Acronyms vi
Technical Information on the Web ix

Chapter 1:
Designing for the Data Warehouse

The Heart of the Data Warehouse 1–2
Data Marts and Data Warehouses 1–3
Data Marts 1–4
Data Mart-Centrism: Why Data Marts Fail 1–8
Data Warehouses 1–10
Special Design Considerations for the NCR Data Warehouse Environment 1–11
Usage Considerations for Design: OLTP and DSS 1–13
Usage Considerations for Design: Summary Data and Detail Data 1–17
Usage Considerations for Design: Simple and Complex Queries 1–20
Usage Considerations for Design: Ad Hoc Queries 1–24

Section 1:
Semantic Data Modeling

Chapter 2:
What Is Logical Data Modeling?

The Database: Heart of the Data Warehouse 2–2
Designing for OLTP and Designing for Decision Support 2–4
ANSI/X3/SPARC Three Schema Architecture 2–6
Requirements Analysis 2–10
Logical Database Design 2–11
Activity Transaction Modeling 2–12
Physical Database Design 2–13

Chapter 3:
Requirements Analysis
Develop an Enterprise Data Model 3–2
Requirements Analysis 3–7
What to Do Before Interviewing 3–9
Interviewing for Requirements 3–12
Analyzing a Legacy Database 3–16

Chapter 4:
Semantic Data Modeling
The Entity-Relationship Model 4–2
Entities, Relationships, and Attributes 4–4
Translating Entities and Relationships Into Tables 4–7
Relationship Theory 4–9
One-to-One Relationships 4–11
One-to-Many Relationships 4–12
Many-to-Many Relationships 4–13
Moving From an Entity-Relationship Analysis to Normalization 4–14
Semantic Modeling References 4–15

Chapter 5:
The Normalization Process
The Key, the Whole Key, and Nothing But the Key 5–2
Properties of Relations and Their Logical Manipulation 5–4
Functional, Transitive, and Multivalued Dependencies 5–8
The Normal Forms 5–11
First Normal Form 5–13
Second Normal Form 5–15
Third and Boyce-Codd Normal Forms 5–18
Higher Normal Forms 5–22
Fourth Normal Form 5–23
Fifth Normal Form 5–26
Other Normal Forms 5–27
Advantages of Normalization for Physical Database Implementation 5–28
Decomposing Relations 5–29
General Procedure for Achieving a Normalized Set of Relations 5–30
Identifying Primary Keys 5–32
Foreign Keys 5–36
The Referential Integrity Rule 5–37

Domains and Referential Integrity 5–45
Denormalized Forms 5–48
Dimensional Modeling, Star, and Snowflake Schemas 5–52
Denormalizing With Repeating Groups 5–59
Denormalizing Through Views 5–61
Denormalizing Through Prejoins 5–64
Denormalizing Through Join Indexes 5–66
Derived Data 5–67
Denormalizing Through Global Temporary Tables 5–69
Normalization References 5–71

Section 2:
Translating the Logical Model Into a Physical Design

Chapter 6:
Modeling for Joins and Dependency Preservation
Relational Joins 6–2
Lossless Joins 6–5
The Lossless Join Relation 6–11
Semantic Disintegrity 6–17
Dependency Preservation 6–20

Chapter 7:
The Activity Transaction Modeling Process
Domains 7–7
Column Names and Constraints 7–10
Guidelines for Naming Columns 7–12
Key Values and Relationships Among Tables 7–15
Domains Form 7–17
Constraints Form 7–20
System Form 7–22
Application Form 7–24
Report/Query Analysis Form 7–26
Table Form 7–33
Filling Out the Table Form 7–36
Table Form Example 7–37
Table Form: Basic Information 7–38
Table Form: Column-Level Information 7–39
Table Form: Miscellaneous Column-Level Information 7–40
Table Form: Access Information 7–41

Table Form: Data Demographics for Single Column Objects 7–42
Maximum and Typical Column Value Frequencies 7–45
Table Form: Data Demographics for Multicolumn Objects 7–50

Section 3:

Physical Database Design, Part 1: Indexing

Chapter 8:
Teradata Indexes: The Basics

Definitions for Parsing Engine Software 8–2
Indexes 8–3
Keys Versus Indexes 8–10
SQL and Indexes 8–13
Primary Indexes 8–16
Secondary Indexes 8–18
Join Indexes 8–20
Hash Indexes 8–23
Reference Indexes 8–25
Indexing and Hashing 8–26
Tradeoffs Between Hashing and Indexing 8–28
Teradata Hashing Algorithm 8–32
Hash Collisions 8–34
Hash Maps 8–35
Hash-Based Table Partitioning 8–38
Hash-Related Functions 8–42
Hashed Row Access 8–45
Hashing and Data Types 8–50
Teradata Index Comparisons 8–51
Table Access Summary 8–52
Index Selection Summary 8–54

Chapter 9:
Primary Indexes

Primary Indexes 9–2
Unique and Nonunique Primary Indexes 9–4
Partitioned and Nonpartitioned Primary Indexes 9–6
Selecting a Primary Index 9–14
Principal Criteria for Selecting a Primary Index 9–19
Analyzing Column Distribution Demographics to Optimize Primary Index Selection 9–24

PPI Case Studies 9–30
Scenario 1 9–31
Scenario 2 9–35
Scenario 3 9–37
Summary of Primary Index Selection Criteria 9–39
Locating a Row Using Its Primary Index 9–40
Secondary Considerations for Selecting a Primary Index 9–45
Performance Considerations for Primary Indexes 9–46
Normalization, Denormalization, and Primary Index Usage 9–48
Duplicate Row Checks for NUPIs 9–50
Minimizing Duplicate NUPI Row Checks 9–54
Primary Index Uniqueness and Space Utilization 9–56

Chapter 10:
Secondary Indexes
Secondary Indexes 10–2
Unique Secondary Indexes 10–4
Nonunique Secondary Indexes 10–9
Dual NUSI Access 10–16
NUSI Bit Mapping 10–20
NUSIs and Covering 10–25
Value-Ordered NUSIs and Range Conditions 10–27
Selecting a Secondary Index 10–29
Secondary Index Access Summarized by Example 10–32
Secondary Index Build and Access Operations Summarized 10–36
Secondary Index Usage Summary 10–38

Chapter 11:
Join and Hash Indexes
Join Indexes 11–2
Types of Join Index 11–7
Join Index Syntax Terminology 11–14
Join Index Definition Restrictions 11–17
Single-Table Join Indexes 11–23
Aggregate Join Indexes 11–26
Optimizer Criteria for Deciding When to Use Aggregate Join Indexes in a Query Plan 11–28
Hash Indexes 11–31
Collecting Statistics on a Hash Index 11–36
Hash Index Definition Restrictions 11–39

Hash and Join Index Interactions With Other Teradata Systems and Features 11–41
Improving Join Index Performance 11–43
Value-Ordered Storage of Join Index Rows 11–48
Example 1: Defining a Simple Join Index on a Binary Join Result 11–49
Example 2: Using Outer Joins to Define Join Indexes 11–52
Example 3: Different Join Index Created Using an Outer Join 11–54
Example 4: Defining and Using a Simple Join Index With an n-way Join Result 11–59
Example 5: Single-Table Join Index 11–61
Example 6: Aggregate Join Index 11–64
Example 7: Aggregate Join Index With EXTRACT Function 11–67
Join Index Design Tips 11–68
Join Index Benefits and Costs 11–71
Hash and Join Index Storage 11–74
A Note on the Accuracy of These Figures for the Current Release 11–78
Creation Costs of Join Indexes 11–79
Maintenance Costs of Join Indexes 11–83
Maintenance Cost as a Function of Number of Hits per Data Block 11–85
Maintenance Cost as a Function of Row Size 11–88
How to Extrapolate Approximate Join Index Performance from This Data 11–93
Maintenance Cost as a Function of Insert Method 11–95
Cost/Benefit Analysis for Join Indexes 11–99

Section 4:
Physical Database Design, Part 2: Integrity

Chapter 12:
Designing for Database Integrity
Integrity Issues 12–2
Databases, Tables, Rows, and Constraints Are Formal Logical Entities 12–4
An Illustration of How Databases Are Built From Logical Propositions 12–9
Business Rules 12–11
Integrity Constraint Types 12–12
Constraint Specifications 12–15
Constraint Enforcement 12–22
Updatable Cursors and Database Integrity 12–23
Integrity Constraints for Updatable Views 12–25
References 12–30

Chapter 13:
Designing for Missing Information
Semantics of SQL Nulls 13–2
Summary of Inconsistencies in How SQL Treats Nulls 13–3
Bivalent and Higher-Valued Logics 13–5
Perspectives on the Advantages and Disadvantages of SQL Nulls 13–8
Arguments in Favor of Using Nulls to Record Missing Information 13–9
Alternative Proposals for Dealing With Missing Information in Relational Databases 13–11
Manipulating Nulls With SQL 13–16
How SQL Operators and Functions Handle Nulls 13–18
Miscellaneous Topics on Manipulating Nulls With SQL 13–22
NULL Literals 13–24
Nulls and the Outer Join 13–25
References 13–27

Section 5:
Physical Database Design, Part 3: Capacity Planning

Chapter 14:
Database-Level Considerations
Base Table Row Format 14–4
Table Header Format 14–7
Column Sizing Guidelines 14–12
System-Derived Columns 14–13
Value Compression 14–20
Compression of Zipfian Data 14–24
Determining the Top *n* Values for a Column 14–30
Storing Data Efficiently 14–31
Compressing A World Name Database 14–34
Typical Uses For The Capacity Reclaimed By Compression 14–36
How the Row Header Indicates Column Nullability and Compressibility 14–37
Tradeoffs Between Compression and the Net Capacity Required to Store Compressed Values 14–43
Computing and Interpreting the Break-Even Point for Compression 14–44
Determining How Much Compression Can Be Realized 14–47
Calculating the Efficiency of Compression 14–50
Data Type Considerations 14–75
Row Size Calculation 14–84

Sizing Base Tables and Index Subtables 14–88
Sizing Base Tables, Hash Indexes, and Join Indexes 14–89
Sizing a Unique Secondary Index Subtable 14–91
Sizing a Nonunique Secondary Index Subtable 14–92
Sizing a Reference Index Subtable 14–94
Sizing Spool Space 14–96
Sizing a Query Capture Database 14–99
How to Size Table Space Empirically 14–101

Chapter 15:
System-Level Considerations
Database Size Considerations 15–2
Contents of System Disks 15–3
Contents of Data Disks 15–4
Data Disk Space Allocation 15–5
Permanent Space Allocations 15–6
Estimating Database Size Requirements 15–8
Determining Available User Table Data Space 15–11
Designing for Backups 15–17

Section 6:
Designing for Tactical Query Support

Chapter 16:
Design Issues for Tactical Queries
Table Definitions 16–2
What Are Tactical Queries? 16–4
Scalability Considerations for Tactical Queries 16–5
Localizing the Work 16–8
Application Opportunities 16–20
Join Indexes and Tactical Queries 16–25

Section 7:
Appendixes

Appendix A:
Notation Conventions
Syntax Diagram Conventions A–2
Table Column Definition and Constraint Abbreviations A–6
Predicate Calculus and Set Theory Notation Used in This Manual A–7
Dependency Theory Notation Used in This Manual A–8

Appendix B:
Teradata Database Maxima
System Maxima B–2
Database Maxima B–3
Session Maxima B–5

Appendix C:
Designing With Task-Oriented Profiles
Isolating Users from the Database Using Views C–2
Flow Diagram of the Database Structure C–3
Policies and Procedures C–4

Appendix D:
Summary Physical Design Scenario
Physical Design Scenario D–2

Appendix E:
Supporting Information for Join Index Performance Analysis
Test System Configuration E–2
Test Table Information E–3
Test Suites E–4
Test Table DDL E–5
Test Join Index DDL E–6
Test Join Index DML E–10

Appendix F:
Sample Worksheet Forms

Index..... Index–1

Chapter 1:

Designing for the Data Warehouse

This chapter introduces the topic of designing a relational database management system for the data warehouse environment.

Data marts are defined, their relationship with data warehouses are described, and their shortcomings are listed.

General guidelines concerning the basic requirements for a proactive-query-enabled data warehouse are described with limited industry-specific focus.

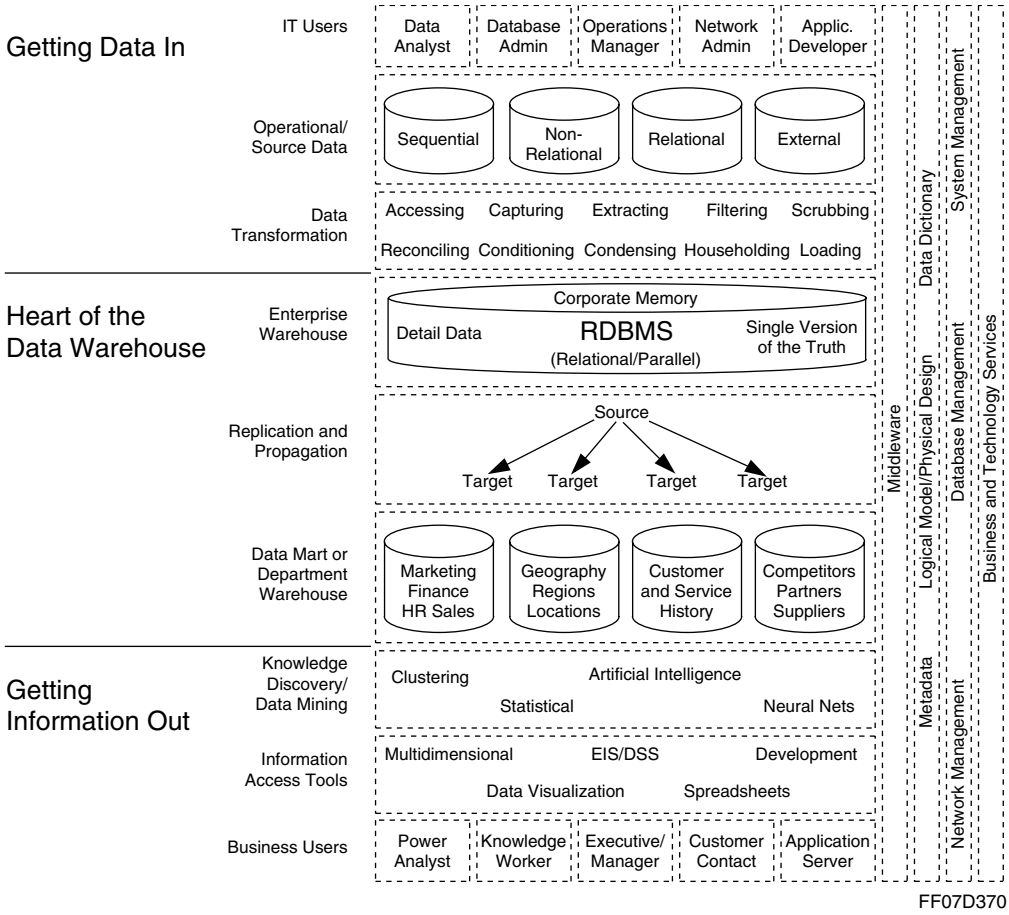
No detailed information about specific applications is introduced.

The Heart of the Data Warehouse

The database is the heart of the data warehouse. Without a properly designed and configured database, no data warehouse can provide the sort of ready access to the enterprise-wide data store required to answer the ad hoc tactical and decision support queries and data mining requests that the modern business enterprise generates in its quest to remain ahead of its competition.

Different users require different sorts of data, ranging from canned batch reports to complex ad hoc queries to detailed exploratory analyses examining the finest-grained atomic data of the enterprise. The database must be capable of supporting all these users and all the possible requests they might make of the enterprise data store.

The following graphic indicates the sorts of processes that comprise the living, active data warehouse, including the wide range of data sources and the equally wide range of information extraction tools.



Data Marts and Data Warehouses

You have no doubt read about data marts and data warehouses. The principal purpose of this topic is to differentiate between the two and to provide commentary on when each is or is not appropriate using Teradata technology.

The topic also introduces the concept of the active data warehouse and contrasts it with the historical definition of data warehousing.

Data Marts

Introduction

A data mart is generally a relatively small application- or function-specific subset of the data warehouse database created to optimize application performance for a narrowly defined user population.

Data marts are often categorized into three different types.

- Independent data marts

  Independent data marts are isolated entities, entirely separate from the enterprise data warehouse. Their data derives from independent sources and they should be viewed as data pirates in the context of the enterprise data warehouse because their independent inputs, which are entirely separate from the enterprise data warehouse, have a high likelihood of producing data that does not match that of the warehouse.

  These independent data marts are sometimes referred to as *data basements*, and Teradata strongly discourages their use (see "Independent Data Marts" on page 1-4).
- Dependent data marts

  Dependent data marts are derived from the enterprise data warehouse. Depending on how a dependent data mart is configured, it might or might not be useful.

  The recommended process uses only data that is derived from the enterprise data warehouse date store and also permits its users to have full access to the enterprise data store when the need to investigate more enterprise-wide issues arises.

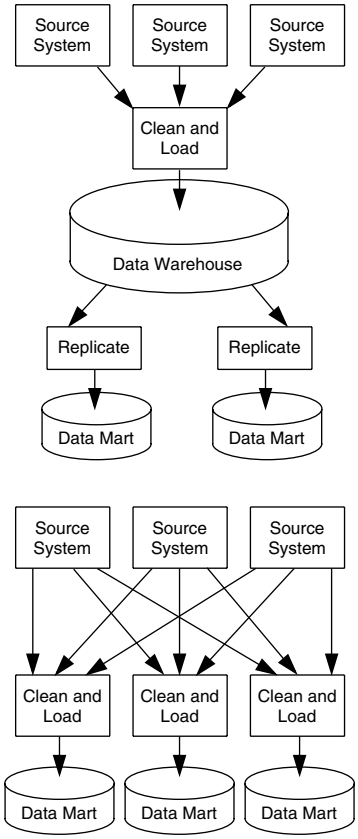
  The less useful forms of dependent data mart are sometimes referred to as data junkyards (see "Dependent Data Marts" on page 1-6).
- Logical data marts

  The logical mart is a form of dependent data mart that is constructed virtually from the physical data warehouse. Data is presented to users of the mart using a series of SQL views that make it appear that a physical data mart underlies the data available for analysis (see "Logical Data Marts" on page 1-7).

Independent Data Marts

An independent data mart has neither a relationship with the enterprise data warehouse nor with any other data mart. Its data is input separately and its analyses are conducted autonomously. Because the data is not derived from the central warehouse, the likelihood that it does not match the enterprise data is high. Which version of reality is correct? How can a user know?

Teradata strongly discourages the use of independent data marts, sometimes referred to disparagingly as “data basements.” Implementation of independent data marts is antithetical to the motivation for building a data warehouse in the first place: to have a consistent, centralized store of enterprise data that can be analyzed in a multiplicity of ways by multiple users with different interests seeking widely varying information.



Independent Data Marts

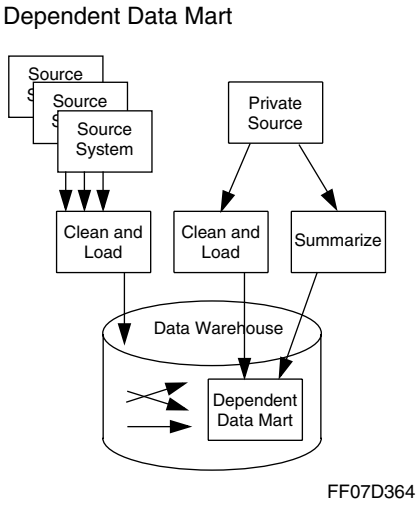
FF07D365

A data basement is a collection of independent data marts. Suppose you have parts that you decide to store in your basement. There is no particular rhyme or reason to what part is stored or where it is stored other than convenience. Continuing the analogy, what is stored in the basement depends on what any family member decides needs to be stored there. If you need to locate a part that you think might have been stored in the basement, you ask everybody in the family if they have seen it recently and then you make your search based on their recollections. If you need to visit more than one basement to find your parts, it is unlikely they will be compatible even if you are able to find them.

This method of storing data is essentially the same as the mix of paper databases and mixed hierarchical and relational online databases spread among multiple departments that supports many businesses today. It is the sort of situation that businesses generally want to escape, not automate.

Dependent Data Marts

If you need to develop one or more physical data marts in the Teradata environment, you should strongly consider configuring them as dependent data marts. Dependent data marts can be built in one of two ways: either where a user can access both the data mart and the complete data warehouse, depending on need, or where access is limited exclusively to the data mart. The latter approach is deprecated and the type of data mart it produces is sometimes referred to as a data junkyard.



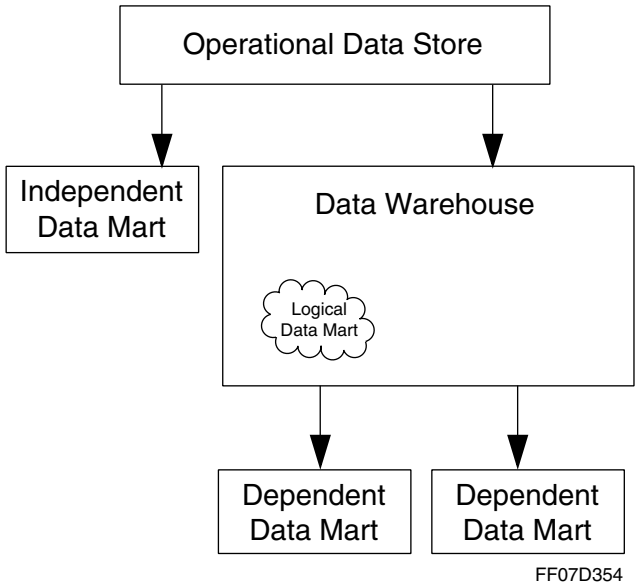
In the data junkyard, all data begins with a common source (in this analogy, “cars”), but they are scrapped, rearranged, and generally junked to get some common parts that the yard operator believes are useful to his customers. The parts collection in the junkyard relates more to what has been useful in the past: previous supply and demand determines what the user can access.

Continuing the analogy, you, as a user, visit the junkyard and search through the various wrecks you encounter in hopes of finding the part you need. To find your part (to answer your question), you will probably need to scavenge through several different junkyards.

The approach results in a decision support environment molded, and compromised, from a specific, well known set of questions and responses rather than around your ever-changing business needs.

Logical Data Marts

Perhaps the ideal approach to incorporating the data mart concept into your data warehouse is to construct one or more logical, or virtual, data marts. By using a system of carefully constructed views on the detail data of the warehouse, you can design multiple user- or department-specific virtual data marts that provide the same sort of highly tailored information a physical data mart would without the need for massive data loads, cleansing, and other necessary transformations.



Data Mart-Centrism: Why Data Marts Fail

Common Problems With Data Marts

Prospective builders of data warehouses are frequently advised to “start small” with a data mart and use that kernel to expand gradually into a full blown data warehouse. This approach to warehousing generally leads to failed projects for several reasons.

Sometimes the new data mart is so successful that the configuration is overrun by user demands. The databases grow too large too fast, response times become unacceptably long, and user frustration leads to searching for other ways to get the answers.

The more common reason for failure is that the data mart is immediately unsuccessful because it is designed in such a way that users are unable to retrieve the sort of information they want and need to extract from the data. Databases are highly denormalized to respond to a small set of canned queries; summaries, rather than detail data, comprise the database so that fine-grained exploratory data analysis is not possible; and support for ad hoc queries is either absent or so poor as to discourage users from bothering with them.

The very factors that frequently defeat data mart projects are also the most commonly recommended approaches to designing data marts and data warehouses in the popular data warehousing literature:

- Denormalization (dimensional modeling)
- Storing aggregates at the expense of detail data
- Skewing performance toward a select few queries at the expense of all other exploratory analyses

Teradata refers to this approach to building a data warehouse as “data mart-centric.” Instead of designing a data warehouse from the ground up, the data mart-centric approach begins with a highly customized data mart and then attempts to expand the kernel into a full blown data warehouse at some point.

Avoid the Data Mart-Centric Approach

Data marts can have a place in a well-designed data warehouse, but when they are designed as dead ends for user information searches or they cannot meet the needs of your users. Both the data junkyard and the data basement are examples of data mart-centric strategies, and both are doomed to failure.

You are probably asking yourself “If this approach to warehousing is so counterproductive, why do so many vendors and data warehousing guides recommend it?” The answer to this question is fairly simple: vendors recommend these approaches because the products they are selling are incapable of performing at the levels required to do true data warehousing. Authors recommend these approaches because they are only experienced using systems whose deficiencies require data mart-centric design in order to perform at all.

An effective data warehouse is not only *not* data mart-centric, it is a foundation for any information analyses that can be made within the enterprise.

Data Warehouses

Introduction

This topic introduces the data warehouse, approaching the subject from several perspectives.

The Classic Concept

The term data warehouse was introduced by William Inmon in his 1993 book *Building the Data Warehouse*. The original concept was essentially a historical database containing tables derived from an active operational database. In this original formulation, the data in the warehouse had the following set of attributes.

- Subject-oriented
- Integrated
- Time-variant
- Nonvolatile

In this data warehouse, the rows in the tables supporting the operational database were to be loaded to a historical database (the data warehouse) after they exceeded some well defined ageing date. The information in the data warehouse was for historical queries only. What distinguished these data in data warehouse tables from the tables of the operational database was the addition of a timestamp to their primary key.

The Teradata Concept

Perhaps the most noteworthy attribute in the Inmon list is the nonvolatility of the data warehouse. No inserts. No deletes. No updates. No changes of any kind. The Inmon data warehouse is a *static*, historical structure.

By contrast, the data warehouse as conceived by Teradata is a dynamic process. The Teradata warehouse is an *active* data warehouse that supports the ongoing, day-to-day business of the living enterprise, including many aspects of the traditional operational data store. Enterprise business rules are isomorphic with the rules of the data warehouse. Its most characteristic attribute is its volatility. The Teradata data warehouse is a *dynamic* entity.

In summary, the heart of the data warehouse is your data modeled in such a way that it matches your business. Just as the warehouse is used as a tool to change the business, so it must change over time if it is to continue to be a critical tool for supporting the enterprise. A true data warehouse sows the seeds that force its own change. The only static warehouse is the warehouse with little or no value to the business.

Special Design Considerations for the NCR Data Warehouse Environment

Introduction

An earlier topic in this chapter ("Avoid the Data Mart-Centric Approach" on page 1-9) observed that the most frequently seen advice for how to design the database that supports a data warehouse is antithetical to the overriding reason for having a warehouse: unrestrained access to enterprise-wide business data for any possible question that could possibly be conceived.

Special Design Considerations?

Are there any special database design considerations for building a Teradata data warehouse? The short answer is no, there are no *special* concerns, but certain aspects of commonly recommended data warehousing strategies should be evaluated for their veracity before you begin designing the database that supports your data warehouse. For these evaluations, see "Usage Considerations for Design: OLTP and DSS" on page 1-13, "Usage Considerations for Design: Summary Data and Detail Data" on page 1-17, "Usage Considerations for Design: Simple and Complex Queries" on page 1-20, "Usage Considerations for Design: Ad Hoc Queries" on page 1-24, and Chapter 16: "Design Issues for Tactical Queries."

The Teradata parallel technology is optimized to perform tasks in a normalized environment that other relational DBMSs cannot match with a denormalized schema. Among the special performance advantages built into the Teradata system are the following: star join and other join optimizations, full table scan optimization, specially designed index types, a full complement of SQL aggregate and ordered analytical functions, and the most sophisticated SQL query optimizer available.

There are a few things you should consider when designing the database to support your Teradata data warehouse, but none of them involve taking any nontraditional approaches to the task.

- One major consideration is whether third party vendors from whom you are considering buying tools are NCR partners. Query tools, for example, should take advantage of the Teradata optimizations (for example, they should incorporate the standard Teradata aggregate and ordered analytical functions rather than performing in-line calculations of OLAP statistics).
- Another major consideration is that if you intend to perform a mix of ad hoc tactical and decision support queries or if you plan to undertake any serious data mining projects, then you should limit denormalization of the database to a minimum.

  For example, if you perform frequent ROLAP analyses of your data and performance is not what you expect to see, consider off loading the data to a dependent data mart designed around the multidimensional model that many OLAP proponents advocate. Do *not* denormalize the entire database just to support a few OLAP applications.
- If you plan to use data mining technology, then a normalized database is the key to your success. Data mining techniques do not perform well in a denormalized environment.

Usage Considerations for Design: OLTP and DSS

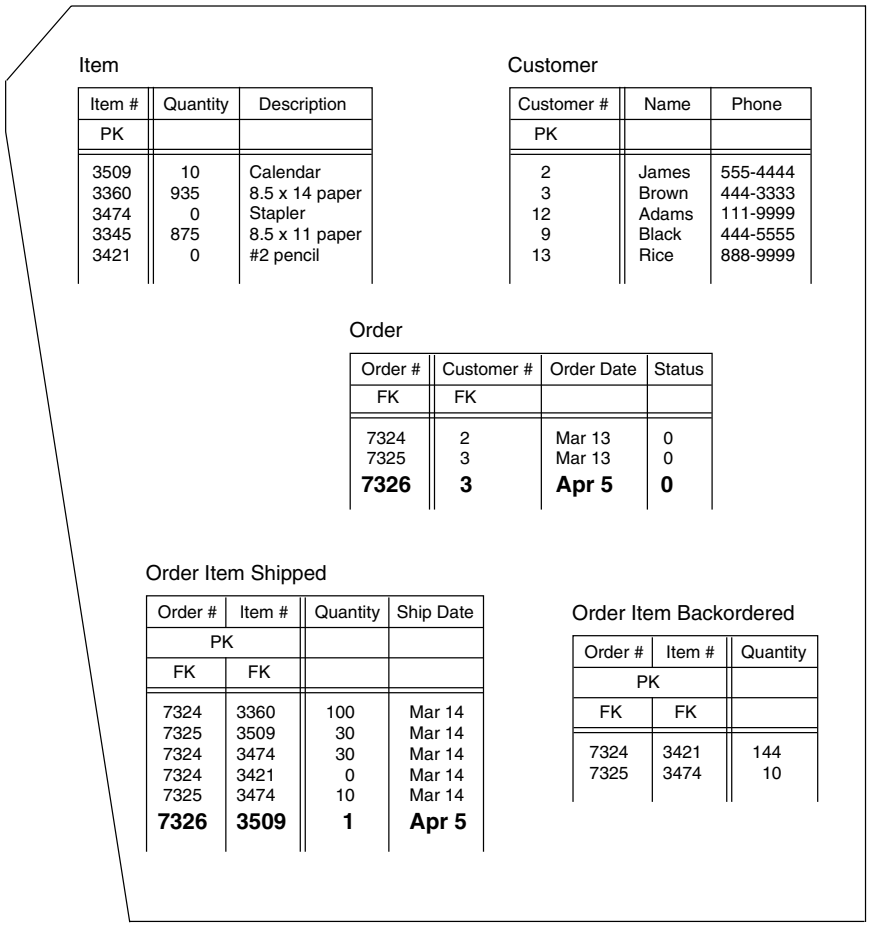
Introduction

The two principal commercial uses for relational systems are Online Transaction Processing (OLTP) and Decision Support (DSS). The access patterns of these two approaches are very different and they make very different demands on the underlying database engine.

The next few topics examine the different access patterns of OLTP and DSS processing. Once the two styles of processing have been compared, you should readily recognize just how dramatically different they are.

OLTP

Consider the following simple example: A customer, Mr. Brown, orders a calendar. The graphic diagrams the actions taken against the database.



FF07D357

The transaction flow is outlined in the following table.

Stage	Process
1	A new order, 7326, is opened on the Order table.
2	The remaining columns for the order are filled out, including the Customer number for Mr. Brown, 3, and the date the order was placed, April 5.
3	A new shipping order, with order number 7326 and item number 3509, is opened on the Order Item Shipped table.
4	The remaining columns for the shipper are filled out, including the quantity shipped, 1, and the date the order was shipped, April 5.
5	The transaction is complete.

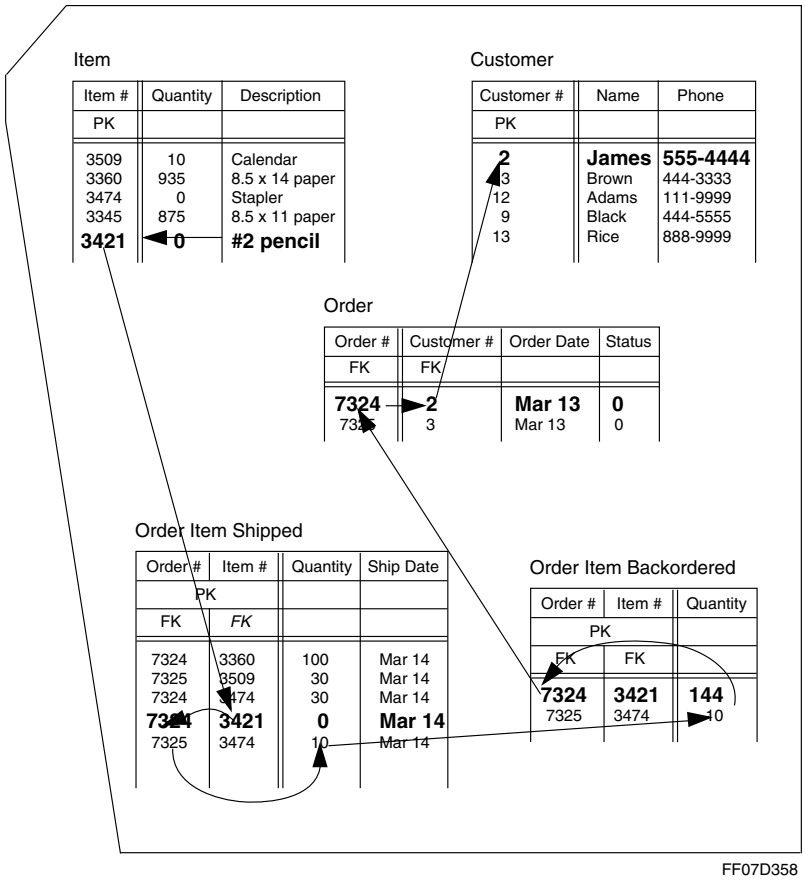
Three key aspects of this transaction deserve your attention.

- Only a few of many possible tables were accessed.
- None of the accessed tables, some of which might have billions of rows, were scanned.
- Very little I/O processing was required to complete the transaction.

These three attributes are characteristic of all OLTP environments. The requirements are simple and involve only a quick-in, quick-out approach to the database. Most modern relational database engines are designed specifically for maximum efficiency with these kinds of workloads.

DSS

Consider the following simple decision support example using the same database to emphasize the contrast in processing requirements: A business analyst needs to find the answer to a specific question about pencil customers so he can target them for a promotion. The natural language query the analyst performs is this: which customers placed the majority of their orders for pencils in the first quarter of this year? The graphic diagrams the actions taken against the database.



The processing flow is outlined in the following table.

Stage	Process
1	The database engine interrogates the Item table for the item number for pencils, which is 3421.
2	The engine next scans the Order Item Shipped table looking for all orders associated with item number 3421.
3	The particular row examined in this example shipped 0 pencils, indicating that the order was back ordered.
4	The engine accesses the Order Item Back ordered table row having an order and item number matching that examined in stage 3 and finds that 144 pencils were back ordered.

Stage	Process
5	The database engine must now locate the customer who placed this order because the whole point of the query was to identify customers who had ordered pencils in the first quarter of the year. To do this, the engine must access the Order table, where it finds that customer number 2 placed the order in question.
6	The database engine accesses the Customer table to identify the name of the customer associated with customer number 2, which is James.

There are also three key aspects of this query that deserve your attention.

- The large number of tables that had to be accessed in order to answer it.

  This complex access path through many tables is a hallmark of decision support analysis. And remember: this is an extremely simple example!
- The tables were not just touched lightly in response to the query, but required massive searches, and sometimes multiple scans are required.
- Once the data required to answer the query has been gathered, it must be processed further using aggregation, joins, sorts, conditional requirements, and so on.

All this processing is heavily resource-intensive.

The following table presents a summary contrast of DSS and OLTP processing.

OLTP Transaction	Attribute	DSS Query
No	Multiple tables scanned	Yes
No	Large volumes of data examined	Yes
No	Processing-intensive	Yes
No	Response time is a function of database size	Yes

Usage Considerations for Design: Summary Data and Detail Data

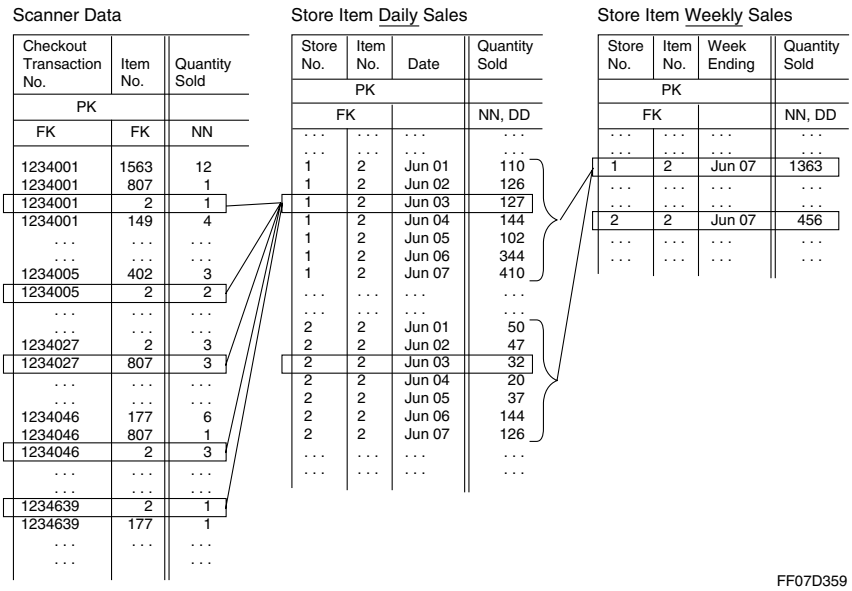
Introduction

You will frequently see data warehousing "authorities" recommend that the warehouse store summarized data rather than the raw detail from which those summaries would be derived.

This topic examines the nature of the data you keep in your data warehouse and attempts to indicate why storing detail data is a better idea, particularly for ad hoc tactical and decision support queries and data mining explorations.

A Simple Case Study of the Effects of Summarization

Suppose we have an application that gathers check stand scanner data and stores it in relational tables. The raw detail data captured by the scanner includes a system-generated transaction number, codes for the individual items purchased as part of the transaction, and the number of each item purchased. The table that contains this detail data is named Scanner Data in the following graphic.



The middle table, Store Item Daily Sales, illustrates a first level summary of the data in Scanner Data. Notice that where we knew which items sold at which store at which time of day in Scanner Data, now we only know the quantity of an item sold for an entire business day. The clarity of the detail data has been replaced by a more fuzzy snapshot. Potential insights have been lost.

The right most table, Store Item Weekly Sales, illustrates a further abstraction of the detail data. Now all we have is the quantity of each item sold by each store for an entire week. Still fewer insights can be garnered from the data.

Of course, the data could be further abstracted. Summarization can occur at many levels. The important lesson to be learned from this study is that summaries hide valuable information. Worse still, it is not possible to reverse summarize the data in order to regain the original detail. The sharper granularity is lost forever.

Consider this simple, and highly logical, query that an analyst might ask of the sales data: How effective was the mid-week promotion we ran on sales for an item on Tuesday and Wednesday? If the only data available for analysis is a unit volume by week entity, then it is not possible to answer the question. The answer to the question is to be found in the detail, and the analyst has no way to determine the effectiveness of the promotion.

Other basic questions that cannot be answered by summary data include the following.

- What is the daily sales pattern for item 2 at any given store?
- When a customer purchases item 2, what other items are most frequently purchased with it?
- What is the profile of a customer who purchases item 2?

Summary Data Has Low Information Content

The information value of summary data is extremely limited. As we have seen, summary data cannot answer questions about daily sales patterns by store, nor can it reveal what additional purchases were made in the same market basket, nor can it tell you anything at all about the individual customer who made the purchase.

What summary data can provide is summary answers and nothing more. This puts you in the position of always being reactive rather than proactive. Two classic retail dilemmas posed by this summary-only situation are the following, indicating that both extremes of a given problem can be caused by only having access to summary data.

- The case where an out-of-stock situation has only been discovered after it is too late to remedy the problem.
- The case where there is too much stock on hand, forcing an unplanned price reduction promotion to eliminate the unwanted inventories.

Proactive Use of Detail Data

Typical examples of how detail data provide the ability to deal abound.

Suppose the retailer in this case example used detail data to analyze which products tend to cluster in the same market basket. Once the product clusters have been determined, it is possible to rearrange the layout of shelf displays to encourage yet more of this buying behavior.

As an another example, a retailer could use detail data to determine what types of customer tend to buy a particular product or product family. With this information in hand, the retailer could then target a specialized promotion to cross-sell those customers on other products.

Usage Considerations for Design: Simple and Complex Queries

Introduction

Users query their databases not because they need an answer as an end result; rather, they want guidance for performing a business action that will have optimal decisive effects. The actions taken cause your bottom line to change, not the answers that informed those actions.

Simple Queries and Complex Queries

A query can be simple or it can be complex. The answers provided by simple queries tend to be more expected than not, while the answers provided by complex queries tend to produce far less certain answers that are that much richer for resolving the questions they pose.

Consider the following case example from the financial industry. A bank offers its customers a financial instrument that a competitor has decided to offer "without charge." This challenge must be responded to quickly or the bank will lose many of its customers who are currently paying what they see as an unnecessary fee for the financial instrument under discussion.

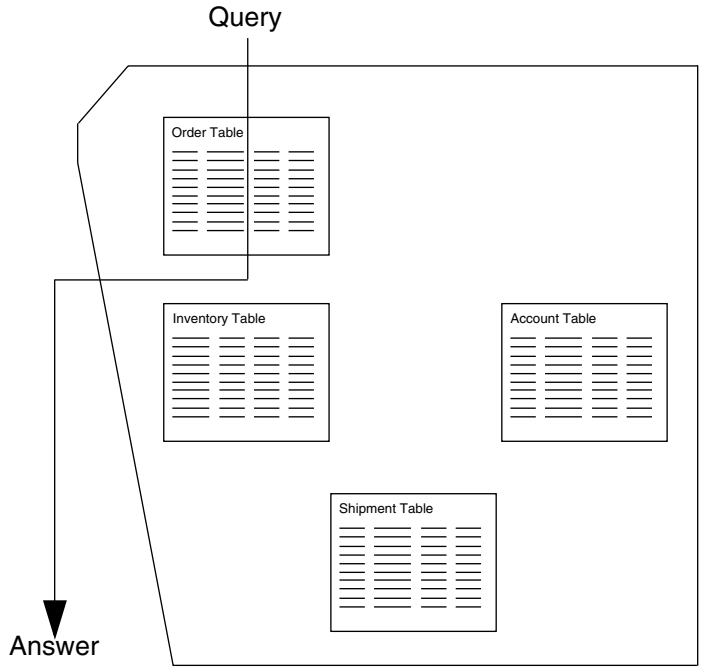
Suppose the bank responds to this situation by asking a simple question such as "which current customers use this product"? The most likely response to this information, which is merely a list of current customers, would be to eliminate the fee for the product. While this action is likely to forestall the erosion of the customer base, it also has the negative effect of reducing profits for the bank.

Suppose, instead, that the bank asks a more sophisticated question such as "which current customers using this product would remain profitable clients even if the fee were eliminated"? The bank now knows not only which consumers of its financial instrument are profitable for reasons other than their consumption of that product, but also knows which consumers do *not* otherwise contribute to the bottom line. The latter customer set can be released to the competition, which, by the way, almost certainly does not know that the new customers it is luring away from our bank are not profitable.

The impact of this more complex query is profit maximization, and its example illustrates very clearly the value of complex over simple queries.

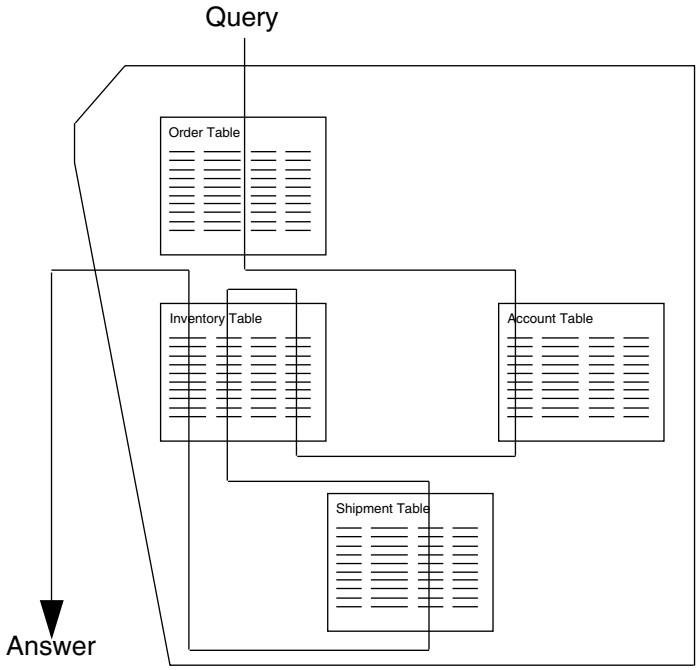
Relationship Between Query Complexity and the Value of Its Answer

A simple query typically accesses only a few tables, as illustrated by the following graphic.



The simplicity of such a query maps directly into the simplicity of the answer it returns. In other words, simple queries tend to deliver low-value answers which, in turn, enable low-value actions.

More complex queries, on the other hand, investigate the multivariate relationships among many tables in search of the high-value answers that come from mining the many interrelationships among the tables accessed. The following graphic illustrates the concept of a moderately complex query. The example winds its way through four individual tables, accessing one of them several times. A more realistic complex query could easily access hundreds of tables, including as many as 64 of them in a single join!



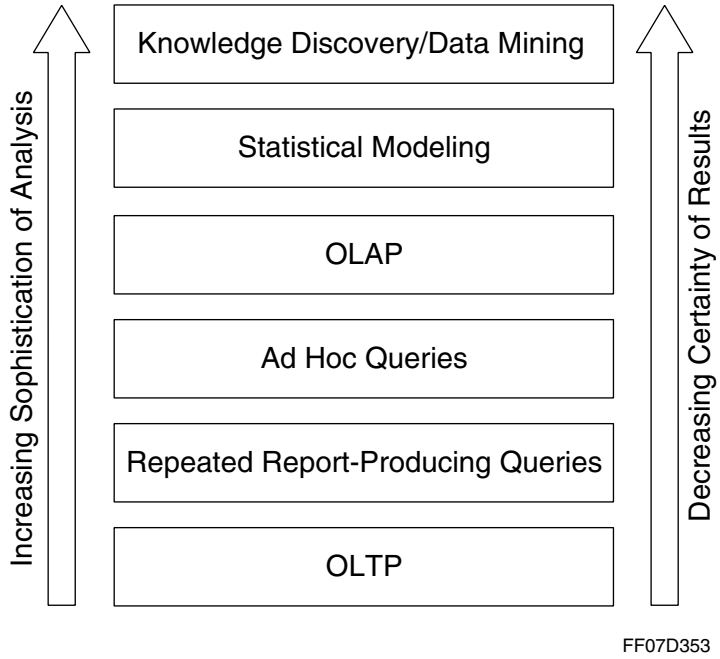
Query complexity exerts a processing an I/O burden that many commercial relational database management systems are not capable of handling, which is the principal reason that most data warehouse vendors advocate the use of summary data.

The relationship between query complexity and the extent of detail in the database is direct and profound. While summary data is often good enough to answer simple queries, it cannot deliver the sorts of answers that more complex queries seek. This is an extremely important concept to understand before designing your databases, because you need to provide the level of detail in the data that can deliver answers to the types of questions you will be asking.

Valuable Information Is Worth Waiting For

The value of information is often inversely proportional to the length of time required to derive it.

As the following graphic illustrates, the more sophisticated the analysis, the less expected the answers obtained. More explicitly, this principle can be stated as the more complex the query, the more likely that heretofore unknown information hidden in the data is revealed.



Sophisticated explorations of the data universe might take longer to produce results, but when those results are finally produced, they are invaluable to the business. A quick response is a minimal requirement for simple queries, but such queries rarely provide a business-critical response, and designing a database to ensure nothing *but* quick responses is the quickest path to failure for your data warehouse project. The Teradata parallel architecture ensures that *all* queries are answered in an optimal time frame.

Your data warehouse can be a source of unimaginable information richness, but only if it is designed with the thought that any possible question, no matter how involved or abstract, should be possible to answer as readily as a simple query.

Usage Considerations for Design: Ad Hoc Queries

Introduction

Ad hoc is a Latin phrase meaning "for this." More generally, the expression refers to performing an act for specific or immediate needs, or improvisation. When used to modify the word "query," it might be taken to mean "unplanned," and that is sometimes the case, particularly for simple queries used to provide reports.

However, in the data warehousing world, the phrase *ad hoc query* generally refers to a query composed at the keyboard for immediate performance (as opposed to a permanent query stored in a macro, stored procedure, or embedded SQL application that is performed over and over again without alteration). Such a query frequently undergoes careful, extended planning before being performed for the first time and then is often revised and refined interactively until the desired response is returned. The goal of this type of ad hoc query is not the reporting of data, but the discovery of information.

Ad Hoc Queries and Enterprise Tactical Support

Tactical queries are focused on operational decision making rather than enterprise discovery or bookkeeping activities. They typically have the following characteristics:

- Relatively simple syntax
- Direct access
- Highly tuned
- Have expected response latencies on the order of 20 seconds or less, and are often expected to return a response in less than one second.

The Teradata data warehouse provides facilities such as the Priority Scheduler (see *Teradata RDBMS Utilities, Volume 2*) to fully integrate workload mixes of both tactical and strategic queries across the enterprise, enabling executives and frontline decision makers alike to access the same single version of the truth concurrently.

See Chapter 16: "Design Issues for Tactical Queries" for many of the special considerations you must take into account when designing to support tactical queries.

Ad Hoc Queries and Enterprise Discovery

"Usage Considerations for Design: Simple and Complex Queries" on page 1-20 established that discovery is the critical focus of data warehousing, not simple reporting. The trends fueling the data warehousing revolution are based on the need for information that can be acted upon proactively. Report-oriented systems rely on predefined questions that produce reports of what has already happened. This type of information can only produce evidence that a situation that has already occurred must be reacted to.

A data warehouse must have the capability of performing high performance, complex, ad hoc queries if it is to capitalize on the advantages of warehousing its business data. Furthermore, that data must be available for analysis in detail, as well as summary, formats. The value of data warehousing comes from being able to ask unplanned questions on detail data.

Business Situations That Drive the Need for Ad Hoc Queries

Two general situations typically drive ad hoc queries: exploratory analysis to discover business opportunities and detailed analysis of why some complicated event that had a negative impact on the enterprise occurred. You must be able to ask questions as they present themselves in either situation and the data warehouse you use for your analysis must be capable of accommodating that need.

Consider the search for new business opportunities. The following table lists some likely business opportunity searches by industry.

Industry	Typical Query Focus
Financial	Cross-selling opportunities
Retail	Market basket analysis
Communications	Calling patterns indicating high risk of losing a customer to the competition.

Now consider the "what went wrong" type of analysis. The following table lists some likely scenarios for this type of ad hoc query by industry.

Industry	Typical Query Focus
Transportation	• Over capacity • Under capacity
Communications	Sudden customer attrition

Businesses cannot know what questions they will need answered in the future, but they must be able to ask questions that permit them to influence the future positively. Neither issue can be dealt with by means of an analysis of the data warehouse if the system does not provide support for complex ad hoc queries.

Section 1:

Semantic Data Modeling

Chapter 2:

What Is Logical Data Modeling?

This chapter introduces the concept of database modeling, both logical and physical, providing a preview for the remaining chapters in this manual.

Topics described include the following.

- "The Database: Heart of the Data Warehouse"
- "Database Design Life Cycle"
- "ANSI/X3/SPARC Three Schema Architecture"
- "Requirements Analysis"
- "Logical Database Design"
- "Physical Database Design"

The Database: Heart of the Data Warehouse

Introduction

The heart of any data warehouse is its database. The usefulness of a data warehouse is directly proportional to the quality of database that supports it.

The purpose of this manual is to provide guidelines that should lead you to designing a Teradata relational database that provides any user of the warehouse with all of the following characteristics.

- A single, unified, unambiguous view of the data
- Optimal access for any possible valid query
- Freedom from navigation

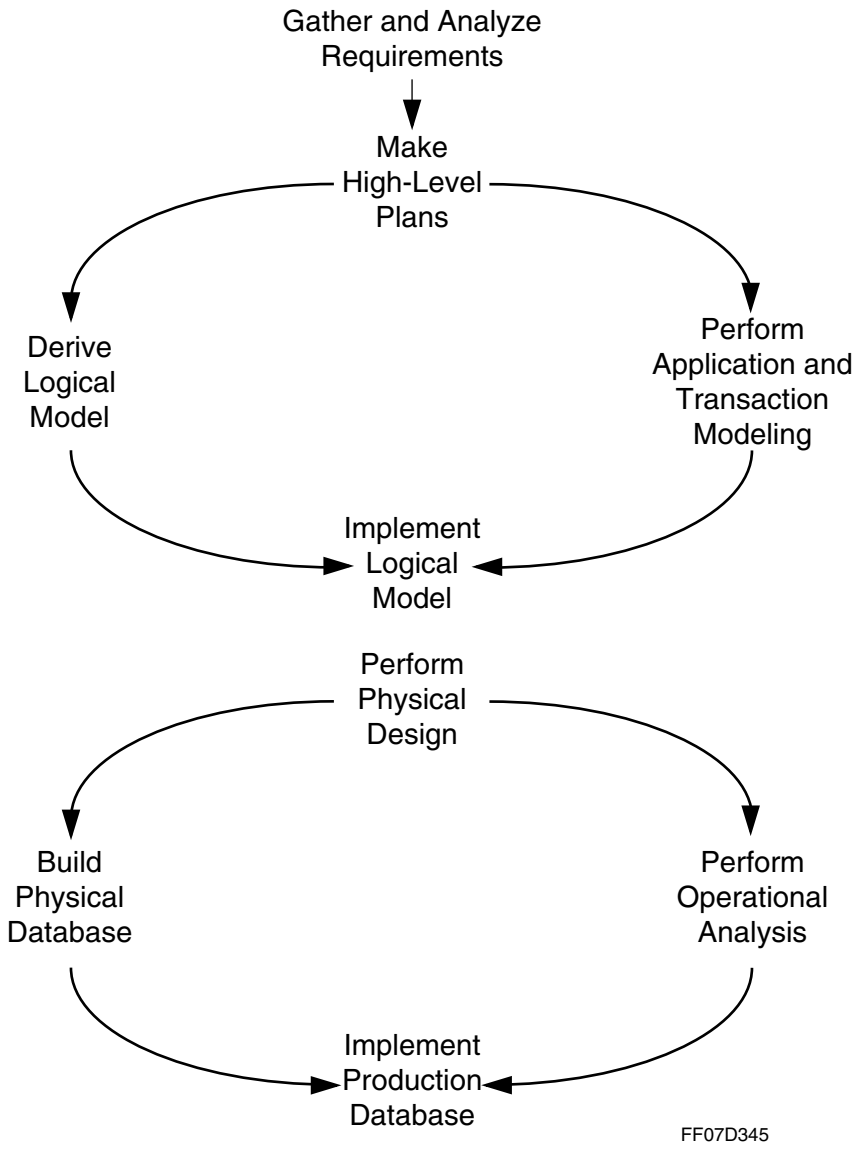
These attributes can only be achieved by careful collection and analysis of requirements and through careful planning of how to implement those requirements. The name used to describe the planning of the structure and relationships of a database is data modeling[1]. Data modeling comes in two modes: logical and physical.

Database Design Life Cycle

The following graphic indicates the flow of the database design cycle. Database design is a living process: it never ends. Not only do new features and optimizations come with each new release of the Teradata relational database software, but new business needs and supporting information requirements continually present themselves and must be integrated with the production database.

In some circumstances, the integration process can be as involved as reengineering entire components of the database. Fortunately, if you have adhered to an implementation that is fully normalized, the reengineering process is a relatively harmless and predictable activity.

1 Note that the expression "data model" has two very different meanings. In the original definition, a data model is taken to be a set of abstract constructs that can be applied to many different specific applications. Examples would be the relational and CODASYL models for database management. In the case described in this section, the term refers to logical database modeling, which is a specific application of the properties provided by the relational data model. An analogy that is sometimes made is that a data model in the first sense is akin to a programming language while a data model in the second is akin to a particular application program written in a particular programming language.


Gather and Analyze Requirements
Make High-Level Plans
Derive Logical Model
Perform Application and Transaction Modeling
Implement Logical Model
Perform Physical Design
Build Physical Database
Perform Operational Analysis
Implement Production Database
FF07D345

Designing for OLTP and Designing for Decision Support

Introduction

The following table examines some of the principal differences in OLTP and Decision Support processing that are critical to database design.

Attribute	Processing Type	
	OLTP Transaction	Decision Support Query
Multiple tables scanned?	No	Yes
Large data volumes?	No	Yes
Processing intensive?	No	Yes
Processing time a function of database size?	No	Yes

As you can see from this side-by-side comparison, OLTP transactions and decision support queries are dramatically different entities and, by inference, might be expected to require physically different database support.

Decision Support

Relational databases that support data warehousing are optimized to support decision support applications in all the forms listed below and more.

- Ad hoc SQL or natural language-generated SQL queries
- Repeatedly stated queries in the form of macros, embedded SQL applications, or stored procedures
- Data mining and OLAP explorations

The data in the warehouse is stored in the form of multiple normalized tables that model your enterprise. The decision support applications that explore these data often perform full file scans of multiple large tables, making them processing intensive. Because processing time is directly related to data volume, the responsiveness of the system is heavily affected by the size of the underlying database.

OLTP

In contrast, OLTP transaction processing tends to operate orthogonally. OLTP systems do not scan multiple tables simultaneously, nor do they access large volumes of data. This makes their consumption of processing and I/O resources miniscule in comparison with decision support query processing. Because the database size determinant is factored from the equation, size is not a factor in the responsiveness of the system.

ANSI/X3/SPARC Three Schema Architecture

Introduction

In 1975, the ANSI/SPARC (American National Standards Institute/Standards Planning And Requirements Committee) Study Group on Data Base Management Systems made its initial interim report on an architecture for database management systems that the committee felt should become an industry standard for the design of commercial database management systems.

The ANSI/SPARC architecture was *not* specifically designed for relational databases, which were not commercially available at the time the interim report was published. Explicitly referring to the hierarchical, network, and relational models, the report notes that "Much of the work [on the architecture] has been driven by the desire to accommodate the various requirements statements and differing viewpoints."

Perhaps the most interesting aspect of the model for relational databases is its pioneering work highlighting the need for data independence. Data independence is a default property of relational databases that hides both the organization of the data and its access from the requesting user or user application.

This property makes it possible to extract information from a database without specifying, or even knowing, how to access the underlying data in the database. In a relational database management system, the query optimizer plays the role of a surrogate user and specifies all the particulars of accessing the data.

The implementation of data independence also makes it possible to change the storage structure of the database without changing, in any way, the code used to access the data in the database.

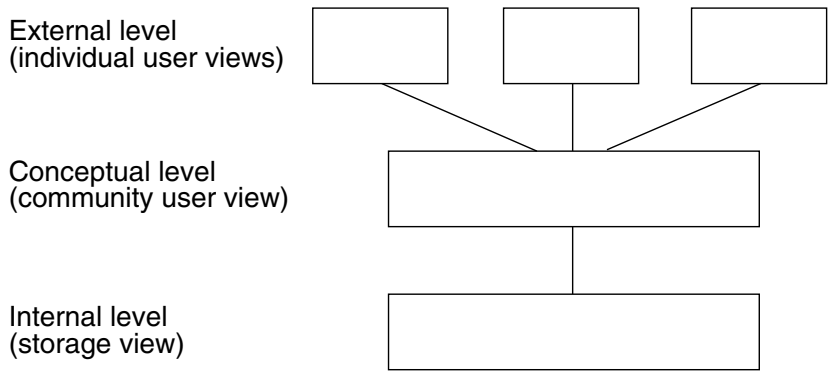
High Level Architecture

The following graphic presents a high level (and highly simplified) view of the ANSI/SPARC architecture.

ANSI/SPARC specifically names three levels of the architecture.

- External
- Conceptual
- Internal

The External level is composed of all the different views (not specifically in the sense of views in relational systems) of the underlying physical database.



The Conceptual level is concerned with transparently mapping External level views to the physical storage of the actual data in the database. In terms of a relational system, this level is composed of the database management system and the file system.

The Internal level describes the physical storage of the data on the storage media.

Detailed Architecture

The following diagram outlines the ANSI/X3/SPARC architecture in greater detail and explicitly relates the various levels to their manifestations in a relational database management system. Note the following points.

- Depending on the situation and object privileges, end users communicate with the database in two possible ways.
  - Through an external view
  - Directly
- Communications through views must be converted, or mapped, in both directions.
- Communications between the database and the disk subsystem are made through the file system.

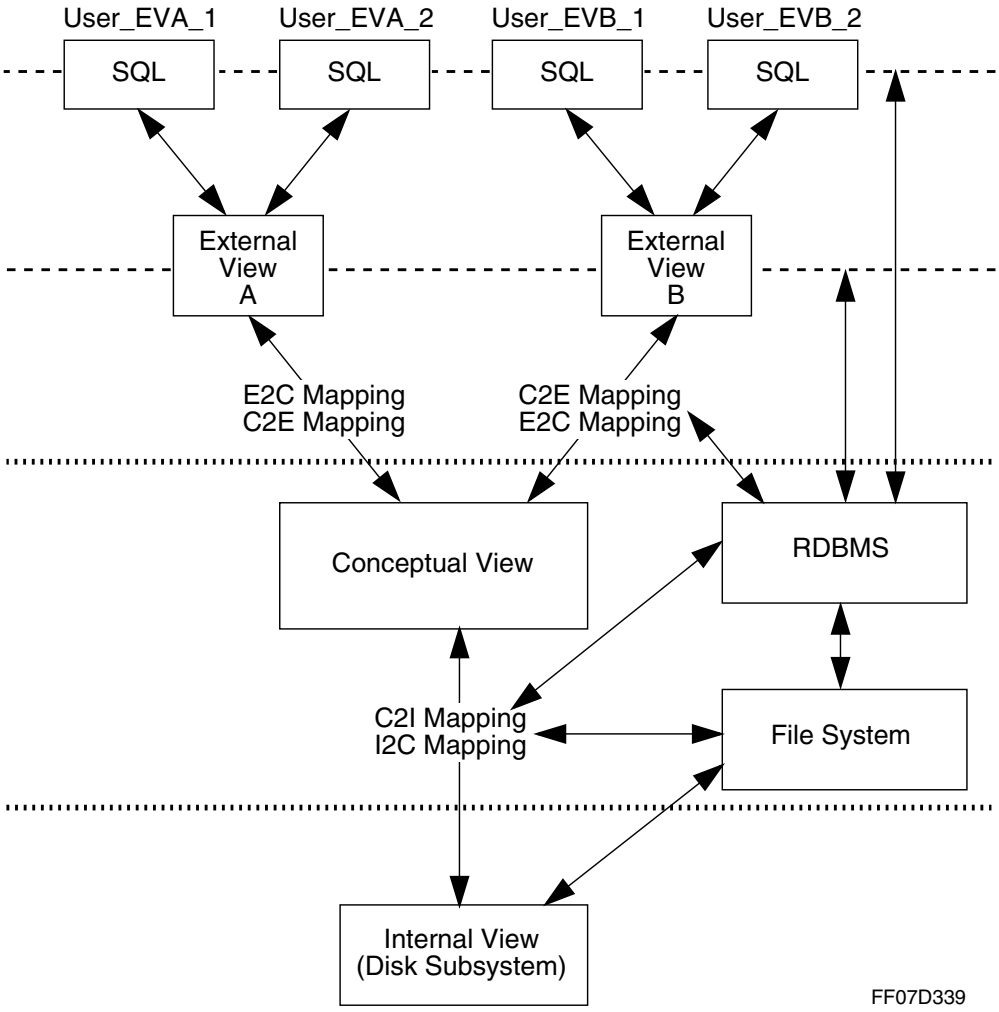
The higher level within the External level is represented by the SQL language for relational systems. The SQL can be embedded within a client application, within a stored procedure, or presented to the database in the form of an ad hoc, interactive query made through a query manager like BTEQ or through a natural language SQL code generator.

The lower level with the External level is represented by a relational view. Views not only mask the underlying database storage, but also the conceptual structure it supports, acting as a virtual tabular interface on the physical base tables.

Note that not all users communicate with the database through views: some, particularly administrative users, communicate with the database directly.

The Conceptual level of the architecture is represented by the relational database management system and the file system. The role of the file system is more nebulous than the diagram portrays and is actually intermediate between the Conceptual and Internal levels, but for purposes of this description of the architecture, it is regarded as a component of the Conceptual level.

The Internal level of the architecture is represented by the disk subsystem. Depending on the configuration, the Internal level could also include storage media such as tape and optical disk. From the perspective of the architecture, only the file system has direct access to the disk subsystem, and it must map data requests and responses in a manner appropriate to the level or component it communicates with.



Key

The following table explains the abbreviations used in this detailed architecture flow diagram.

Abbreviation	Definition
EVA	External View A
EVB	External View B
E2C	External-to-Conceptual
C2E	Conceptual-to-External
C2I	Conceptual-to-Internal
I2C	Internal-to-Conceptual

Reference

The following report defines the ANSI/X3/SPARC model in detail.

Anonymous, ANSI/X3/SPARC Study Group on Data Base Management Systems. Interim Report. *FDT, ACM SIGMOD Bulletin* 7(2):1-139, 1975.

Requirements Analysis

Any design process must begin with the knowledge of what is to be designed. This includes not only the proposed morphology of the end product but also the systems, policies, and procedures - the processes - of the designed product.

This fundamental knowledge is derived through a process of accumulating facts about what the eventual users of the product require to do their work in support of the enterprise. The process includes, at minimum, the following tasks.

- Interviewing notable employees, both management and support staff, for information such as the following.
  - What information do they need?
  - What is the source of that information?
  - What are the tasks involved with creating and reporting the information?
  - How is the information used?
- Gathering all input screens and reports generated by the legacy system and interviewing management and support staff about what is right and wrong about these components as well as determining what sorts of new or different input and report items should be added to the new system.
- Compiling and circulating the cumulative research information you have gathered to obtain affirmation of its accuracy from all involved parties.
- Writing a requirements specification from the approved research information and making it available to the designer of the logical database.

Logical Database Design

Introduction

The requirements analysis phase of the design process reveals the real world objects and their attributes that the database must represent as well as the relationships among them.

The logical database design phase formalizes the objects, or entities, and their relationships. Another primary task of logical design is to ensure that the modeled entities are modified by attributes that uniquely pertain to them. No attribute should appear in an entity unless it describes the uniqueness identifier for the entity, its primary key.

Normalization

Fortunately, the entire structure of relational database management is built on a solid theoretical underpinning that makes the operations used to associate the proper attributes with the proper entities predictable, repeatable, and formally correct.

This process is referred to as normalization. Normalization is not an art form; it is a science. The normal forms and their derivations operate strictly within a series of inference rules and formal logical operations drawn from formal logic and set theory.

Unfortunately, the determination of entities and their relationships is subjective, unpredictable, unrepeatable, and more of an art form than a science. In spite of the subjectivity of entity-relationship-attribute analysis, the normalization process almost always rescues any blunders made in the determination of entities and relationships, so it all works out in the end.

Activity Transaction Modeling

Once a fully normalized logical model has been realized, the next step is to enter the transition phase. In this phase, which Teradata refers to as Activity Transaction Modeling, you translate the entities, attributes, and relationships into a set of worksheets that feed into the physical database design process.

Activity Transaction Modeling

The Activity Transaction Modeling (ATM) process extends the logical data model by beginning to attach physical attributes to it. In doing so, the ATM process undertakes the following activities:

- Identifies the business rules of the enterprise that apply to the information to be stored in the data warehouse
- Initiates the process of identifying and defining attribute domains and constraints for physical columns
- Identifies and models database applications
- Identifies and models application transactions
- Summarizes table and join accesses by column across transactions
- Compiles a preliminary set of data demographics by computing table cardinalities, value distributions, and attaching change ratings to columns

This information is compiled and used as input to the physical design process.

The ATM process is described in detail in Chapter 7: “The Activity Transaction Modeling Process.”

Physical Database Design

Physical database design is the commitment of all the previous design stages to a physical reality.

In previous phases of the design process entities, attributes, and relationships were identified and normalized. Some attribute sets were then identified as primary and secondary index candidates, some were assigned constraints, and all were assigned to domains.

The physical design phase identifies and creates the actual databases, tables, indexes, views, macros, triggers, and other objects that define the physical database that drives your data warehouse.

Chapter 3:

Requirements Analysis

This chapter briefly reviews selected aspects of requirements analysis.

Topics include enterprise data models, the reasons for performing a requirements analysis, and essential techniques for interviewing users in preparation for writing a requirements definition.

Develop an Enterprise Data Model

Introduction

The concept of developing an enterprise data model is, at least from the perspective of data warehousing, an old one. IBM Corporation introduced its *Business Systems Planning* booklet in the late 1970s, began its *A Management System for the Information Business* document series in 1980 and has continued to develop related concepts of enterprise modeling.

Definition

An enterprise data model[1] is a blueprint: a means of ensuring that standards for creating the IT function exist and that they are appropriately integrated with the overall function of the business enterprise the function is designed to support.

About Enterprise Logical Data Modeling

As conceived by Zachman, the enterprise logical data model is based on the idea of a logical construct that defines and controls the interfaces and the integration of all components of the enterprise information structure. The concept draws its ideas primarily from the field of architectural planning and includes not only the architectural drawings created as a framework for a structure but also builder plans, fabrication plans, and the actual construction of the building.

The basic blueprint for the Zachman framework is a table having the dimensional component represented by its columns and the perspectives of the various role players as its rows. In its simplest form, the framework looks like the following table.

1 See "The Database: Heart of the Data Warehouse" on page 2-2 for a description of the different uses of the term "data model." In this chapter, the usage relates to the second definition.

Role	Entities	Activities	Locations	People	Times	Motivations
Planner						
Owner						
Designer						
Builder						
Subcontractor						

Components of the model are more fully defined in the following table.

Component	Definition	Product
	Perspectives	
Planner	Define the scope of the project including constraints on costs and regulations.	Scope definition.
Owner	Define the real world product envisioned, including constraints on usage and policies.	Business model.
Designer	Transform the scope and requirements into a product specification.	System model.
Builder	Build the product, including oversight of the subcontractors.	Technology model.
Subcontractor	Build individual components of the product.	Generic models.
	Dimensions	
Entities	Addresses the question *What?*	• Employees • Departments
Activities	Addresses the question *How?*	• Hiring and assignment procedures • Promotion procedures
Locations	Addresses the question *Where?*	Geographical workplace decisions
People	Addresses the question *Who?*	• Human resources • R&D management
Times	Addresses the question *When?*	Scheduling decisions
Motivations	Addresses the question *Why?*	Ensure adequate staffing and skill levels

Rules for the Framework

You must proceed through the rows in the table in order, beginning at the top and working down. Each perspective is dependent on the product of the perspective that precedes it.

The dimensions of each perspective are independent attributes and have no associated sequence. All have equal weight. Their function is similar to that of the classic Who? What? Where? When? Why? questions that are the framework for journalism. All dimensions must be defined for each perspective before proceeding to the next perspective, but they can be resolved in any order.

This manual is not the place to provide a detailed summary of the Zachman framework nor of enterprise data modeling in general. The topic is raised only because an existing enterprise data model can be a major asset when you begin any new IT project. Building or reengineering the database that will be the heart of your data warehouse is only one such example.

You should consult the references provided in "Enterprise Data Model References" on page 3-5 for details on the concepts.

Milestones in the Concept of the Enterprise Data Model

The following table lists some important milestones in the development of enterprise architecture planning. References to support the milestones are provided in "Enterprise Data Model References" on page 3-5.

Date	Milestone
Late 1970s	IBM publishes *Business Systems Planning: Information Systems Planning Guide*.
1980	IBM Corporation begins to publish its series *A Management System for the Information Business*.
1985	Edward Van Schaik publishes *A Management Information System for the Information Business*, based on the IBM Information Systems Management Architecture.
1987	John Zachman publishes his paper *A Framework for Information Systems Architecture*.
1992	John Sowa and Zachman extend the 1987 model in the paper *Extending and Formalizing the Framework for Information Systems Architecture*.
1997	Bill Inmon, John Zachman, and Jonathan Geiger publish their book *Data Stores, Data Warehousing, and the Zachman Framework: Managing Enterprise Knowledge*, the first systematic application of the Zachman principles to the data warehouse.

Enterprise Data Model References

The following references support the milestones noted in "Milestones in the Concept of the Enterprise Data Model" on page 3-4.

IBM Corporation, *A Management System for the Information Business, Volume I: Management Overview*, GE20-0662-00, 1980.

IBM Corporation, *A Management System for the Information Business, Volume II: The Information Systems Service Mission*, GE20-0749-00, 1983.

IBM Corporation, *A Management System for the Information Business, Volume III: The Information Systems Development Mission*, GE20-0750-00, 1983.

IBM Corporation, *A Management System for the Information Business, Volume IV: Managing Information System Resources*, GE20-0751-00, 1983.

IBM Corporation, *Business Systems Planning: Information Systems Planning Guide* (3rd ed.), GE20-0527-03, 1981.

Inmon, W.H., John A. Zachman, and Jonathan G. Geiger, *Data Stores, Data Warehousing, and the Zachman Framework: Managing Enterprise Knowledge*, New York: McGraw-Hill, 1997.

Sowa, J.F. and J.A. Zachman, Extending and formalizing the framework for information systems architecture, *IBM Systems Journal*, 31(3):590-616, 1992.

Van Schaik, Edward A., *A Management System for the Information Business: Organizational Analysis*, Englewood Cliffs, N.J.: Prentice-Hall, 1985.

Zachman, J.A., A framework for information systems architecture, *IBM Systems Journal*, 26(3):276-292, 1987.

The following two books provide information about implementing data warehouses with an emphasis on the Teradata solution.

- The following book provides information about the development of the largest data warehouse implementation in the world; a system based on Teradata technology. The work details business, project, and enterprise planning as well as technological implementation, with particular emphasis on the retail industry.

  Westerman, Paul, *Data Warehousing Using the Wal-Mart Model*, San Francisco, CA: Morgan Kaufmann Publishers, 2001.
- The following book should be thoroughly studied and absorbed by anyone who is considering implementing a data warehouse.

  Armstrong, Rob, Tom Coffing, and Rolf Hanusa, *Secrets of the Best Data Warehouses in the World: A Fundamental Guide for Executives and Decision Makers*, Thomas Coffing, Rolf Hanusa, Rob Armstrong, 2000.

  Its contents are the distilled essence of what world class data warehousing projects are all about. According to the authors, the following people are the audience for their book:

- Executives who champion the data warehouse cause within the enterprise
- Users whose understanding of and imaginative extension of the data warehouse provide the data by which the success of the project is measured
- IT managers who gather the requirements for the warehouse and instantiate the tools, policies, and procedures that meet and extend those needs.

Requirements Analysis

Introduction

Nobody likes to go through the process of determining requirements for a database project: not enterprise management, not users, and certainly not IT personnel. Nevertheless, undertaking an IT project without having certain knowledge of what is expected, and needed, is a formula for failure.

Why an Analysis of Requirements Is Necessary to Ensure Project Success

The number of formal publications on project management is astronomical. Diversity of opinion about the details of managing projects is common. However, the need for deriving a set of requirements for any project and the fundamental reasons they are necessary are a topic all experts agree on.

Three Reasons for Requirements

There are three principal reasons for performing a requirements analysis.

- Getting the system right the first time
- Supporting easy user access to the system
- Producing reasonable and reliable estimates of costs

Get the System Right the First Time

A formally agreed upon set of requirements is necessary to get the system right the first time.

Industry analysts have reported IT project failure rates as high as seventy percent, largely because no requirements were derived for the project apart from IT guesses as to what their users needed.

Support User Access to the System

As more and more of the task of updating and maintaining the database is shifted to the user community, their input into the design and implementation of the system becomes critical.

Produce a Reasonable Estimate of Development and Operational Costs

Without requirements, it is not possible to estimate the developmental and operational costs of a new database management system. Projects are often terminated by enterprise management because they see costs spiraling out of control. This is as much a problem of psychology as it is an issue of failure to plan.

When a project estimate of X dollars is approved by management, they expect to see IT spend X dollars on the project and no more. When they see costs exceeding initial estimates by an order of magnitude or more, they perceive a project to be a failure even when the system is progressing at a normal rate simply because costs so greatly exceed their expectations. When all parties are aware of what a project is going to cost, the psychological dissonance of budgets is greatly reduced if not eliminated altogether.

Without a requirements analysis, you cannot have any grasp of how much an IT project will cost.

What to Do Before Interviewing

Introduction

This topic reviews some basic guidelines that should be understood before you begin to interview intracorporate users and management and external customers about their database needs and expectations.

Base Requirements on Business Needs

Base the project requirements on business needs, not what is easy or cheap to do or, worse still, "because we have always done it that way." Management can always negotiate projected IT costs downward by postponing or eliminating components of the final requirements report, but it is important to know what the needs are that need to be satisfied.

The project should *not* be led by IT, but it *must* be mediated by IT.

Write a Project Mission Statement

People often think that mission statements are not only unnecessary, but ridiculous. This perception usually comes from a misunderstanding of the point of writing a mission statement, which is to keep IT designers, system analysts, and programmers on the same "page."

The mission statement establishes the purpose of the database and provides a focus for developers. It is never wrong to start a mission statement with "The purpose of this project is …"

The intent of the mission statement is to ensure that the database you design meets the needs of your enterprise. It ensures that the appropriate information is collected to support the design and that an appropriate design is architected.

Keep the mission statement brief (one sentence is usually best), simple, and to the point. Keep details, specific tasks, and supporting statements out of the mission statement. If details, tasks, and supporting statements are deemed absolutely necessary, then use them for the various mission objective statements.

To write a mission statement that supports the goals of the project, you must interview key corporate personnel about their intents.

Begin by interviewing the responsible executive or manager for the organization. Get their input on the purpose of the new system. Try to obtain as much detailed information about the organization as possible, even if it is not directly related to the database project. The point is to derive the most complete overview of the entire organization as possible: what it does and how it does it.

Conduct your interview by asking open-ended questions. “How would you describe …” or “Tell me about how your organization does …”

Use the traditional survey technique of asking the same question in slightly different ways several times to validate responses (and to ensure that you understand them).

At this stage in the process, keep interviews at a general, conceptual level rather than probing analytically for details. Reserve questions of that nature for later interviews.

Write Project Mission Objectives

Mission objectives have the following purposes.

- Provide support for the mission statement and ensure that it is met by project end.
- Define tables, entity relationships, views, business rules (which will later become database objects such as integrity constraints and triggers), and application development.
- Represent the tasks to be performed against the data collected in the database. This information is used to support the mission statement and to help determine various aspects of the database structure.

You should write only one mission objective per task. Tasks should be general. Write your objectives as declarative sentences such as the following examples.

- We must provide selective vendor access to our inventory files.
- We must track overall store-by-store sales on a weekly basis.
- We must track product-by-product sales within stores on a daily basis to stay on top of trends.
- We must track aisle-by-aisle sales within stores on a weekly basis to ensure optimal product placement.

Hold a Project Kickoff Meeting

Before proceeding to the requirements acquisition phase of the project, hold a meeting with your chief user and as many management and end user participants as you can. You should perform at least all of the following actions during the meeting.

- Explain to the participants that the approach you are using is probably new to them and that the systems that result from the analysis might differ significantly both in terms of capability and operation from the legacy system.
- Make it clear that the intent of the project is not to recreate the legacy system or to automate current manual procedures. Emphasize that one of the fundamental purposes of conducting the analysis is to eliminate outmoded policies and procedures.
- Emphasize that a great deal of their time will be required during this phase of the design process. Ensure that the subject community understands that when necessary interview subjects are not available, the course of development is delayed.
- Provide basic instruction on the methods you will use in the logical design process. Ensure that the participants have at least some familiarity with all of the following concepts.
  - Entities
  - Attributes
  - Associations
  - Tables
  - Primary keys
  - Foreign keys
  - Business rules and integrity constraints

Interviewing for Requirements

Introduction

This topic provides some very basic guidelines for interviewing subjects for your requirements analysis.

Why Interviewing Is Important

Interviewing enterprise management and end users provides insights into the actual use of the data that cannot be obtained in any other way. Among the factors that must be determined by the process are the following.

- How best to input data
- How best to report information from the database
- How management interprets and analyzes the reports it receives from the database.

There is no way to make these determinations other than talking with the people who engage in the processes in their daily job of supporting the enterprise.

Determine Who Should Be Interviewed

It is often good practice to select one individual to be the "chief user" for the project. This user represents the entire end user community and coordinates their needs. The chief user is also a good candidate for arranging the time and place for all interviews the IT analysts need to conduct.

What sorts of users should you include in your interview pool? The following list is a bare minimum.

- Speak with people involved in the *process* of the business. A data warehouse is not a thing in a place; it is a process in a population of data.
- Talk to vendors if you plan to use Supply Chain Management and Customer Relationship Management applications.
- Talk to enterprise management about their information needs including what sorts of reports, queries, and data mining capabilities they require. The objective is not to determine *how* to get those answers, but to determine what sorts of answers they think they would like to get *whether it is possible or not*.
- Talk to end users about what they need, what problems they have with legacy systems, and obtain their wish lists of features, interfaces, and processes. The sort of information you are typically seeking when interviewing end users is related to the processes of getting data into and out of the database and not so much how it is interpreted.

Be Alert for Hobby Horses, Hidden Agendas, and Other Suspicious Reports

Though you must be objective while collecting and analyzing requirements, you should never take anything a user tells you at face value. Everybody has a hobby horse they like to ride whenever they get the chance.

It is always important to derive an integrated overview of needs from a variety of sources and not to rely exclusively on “key” people because “key” is always in the eye of the beholder, and it is unfortunately not uncommon for hidden agendas to counter the greater needs of the entire enterprise. It is all too common for users appointed as key people to neither perform the tasks you need to understand nor to understand the processes they are supposed to be representing to you.

End users have been known to insist on the need for something that, on further analysis, turns out to be not only unnecessary, but often useless. We have all heard stores about “must-have” reports that, on further analysis, turn out to be hex memory dumps of aborted programs that are dutifully bound and filed without ever being read by anyone. This is one of the key reasons that report requirements should remain the domain of management interviews.

Solicit a wide variety of input and consider it all carefully.

Interviewing Techniques and Personal Agendas to Consider Before You Begin

The following list of miscellaneous topics is provided to suggest a few commonly suggested methods for maximizing the results of your interviews.

- Always keep the objective of the interview in mind, which is to learn about the business.
  - Identify relevant business processes.
  - Determine the grain by which those processes are measured.
  - Identify the measures used.
  - Determine the dimensions by which the measures are evaluated.
  - Understand how the measures will be used by the end consumer of the process.
- Conduct user and management interviews separately. This should be done not only because of the unavoidable intimidation factor, but also because the two groups have different, often radically so, perspectives on the issues.
- When possible, arrange for 1:1 interviews. If not, keep interview groups as small as possible, preferably no more than five people. This reduces both the chaos and intimidation factors (one or two people tend to dominate sessions larger than a very few participants, while the rest just sit and do not participate).
- For group interviews, arrange for an intermediary to conduct any issues that surface after the interview back to the interviewer(s) as well as preparing the group for its interview prior to the event so all know what to expect and what is expected of them.

- Approach the design as if you had no knowledge of the enterprise or its data model.
- Create a set of interviewing guidelines that you can make your subjects aware of so they know the protocols, what is expected of them, and so on.
- Listen actively.

  This means *listen*. Play dumb: do not approach the process with preconceived notions about any aspect of the design or of the business. The less you think you know, the better the final result of the process is likely to be.

  When an interview subject thinks you have in-depth knowledge of the topic you are interviewing them about, they will literally think you know everything and will overlook explaining details that are critical to the business.

  Psychotherapist Carl Rogers developed active listening as a therapeutic technique, but it is an extremely useful interviewing tactic. When an interview subject tells you something, repeat it back to them - as you understand it - in your own words. This will either elicit a correction or assent. Either way, you have a better handle on how thorough the answer was and whether you need to follow it up with more in-depth questioning.
- Keep the number of interviewers small: generally no more than two per session.
- Before beginning, explain what the interview process is. Tell your subjects what to expect and what is expected from them. Thank them for their time and make certain they understand that their observations *will* be taken into consideration.
- Have a prepared list of questions. Stay on track, but do not hesitate to ask aggressive follow-up questions where an answer raises an important issue. When finished, follow up where necessary for clarification and expansion. Don't forget to unpack any implicit items that might go misconstrued if not made explicit.

  Subjects will occasionally wander off topic; however, what may at first seem to be off topic might well be very much on topic. Recommendation: do not cut a subject off too soon when they stray off topic - at least not until you have definitely determined that they are pursuing rabbits down a hole.

  Bottom line: Always ask *all* of your questions, but use the opportunity to pursue topics down whatever fruitful path opens to you during the course of the discussion.
- Ask open-ended questions. Not "Is the daily sales system a) not adequate b) barely adequate c) adequate d) excellent, but "How do you feel about the daily sales system?

- Take notes or tape record the session.
- Review and summarize your notes (or taped transcription) as soon as possible after the completion of each interview.
- Take control of the interview and maintain it, arbitrating any disagreements among participants that may arise, and keeping things moving at all times.
- Stay on track.

  Review the project scope (its mission statement and objectives) before beginning each interview to ensure that *you* are familiar with the scope of the project.
- When the interview is finished, thank the subjects for their participation and acknowledge the importance of their input.

Analyzing a Legacy Database

Introduction

Legacy databases are often a mess. Like Topsy, they have just grown, and often nobody remembers, for example, why columns were added (notably columns that nobody seems to use any longer), nobody knows why the same information is stored redundantly, often with different data types, in different places, why denormalized (or, perhaps worse, unnormalized) relations were created, and so on.

The task of this phase is to analyze the existing database for its strengths, weaknesses, uses, and disuses and to develop a list of what needs to be retained, what needs to be changed, and what needs to be eliminated.

Analyze the legacy system both from an end user perspective and from an IT perspective and ask how IT could have designed past projects better. Use that kind of anecdotal information when building the new system, but do not allow war stories and naysayers to impede the design.

General Procedure

The following procedure provides a general means for conducting an analysis of a legacy database.

Step	Action
1	Capture all data collection screens (or any other means by which data is inserted into the legacy database).
2	Capture all reports (including terminal-only reports).
3	With captured information presentation media in hand, conduct interviews again, this time capturing *how* users use these media. Construct a list of both open and closed (specific) questions.
4	Drill down by following up with detail-oriented questions aimed at capturing the details of all I/O transactions. Begin generally and drill down successively to more detailed questions about user tasks.
5	Begin to capture entity information by isolating and notating each job-related entity (noun) in the response set.

<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>6</td><td>Begin to capture entity attribute information by isolating and notating all descriptive words in the user narrative.

For example, consider the following subject response: “One thing I do is type in customer information. For each customer, I type in a customer name, shipping address, billing address, shipping phone number, billing phone number, a receiving contact name, and a billing contact name.”

Entity names are single-underlined and attribute names are double-underlined in this response.</td></tr>
<tr><td>7</td><td>Interview for types of data collected and input and how that data is used.</td></tr>
<tr><td>8</td><td>Interview with data collection screens and reports in hand.

For example, “When we last spoke, you told me that one of the things you do is type in customer information.

<present captured customer input screen to user>

How does this screen help you to perform that task?”

Iterate questioning until you are satisfied that you have the task and the input screen/report nailed down. Do not forget to ask about coded information, including report headings.</td></tr>
<tr><td>9</td><td>Review requirements with users.

For example, “Do you contribute all the data that is input to this screen?”

<table>
<tr><th>IF the response is …</th><th>THEN …</th></tr>
<tr><td>Yes</td><td>ask a brief follow-up question to verify the response.

<table>
<tr><th>IF the response is …</th><th>THEN …</th></tr>
<tr><td>Yes</td><td>Go to the next question.</td></tr>
<tr><td>No</td><td>Follow up and clarify.</td></tr>
</table>
</td></tr>
<tr><td>No</td><td>follow up and clarify.</td></tr>
</table>
</td></tr>
</table>

Identify New Data and Reports Required by Users

The following procedure describes how to perform a follow-up interview with a user with the intent of identifying new system inputs and outputs.

Step	Action
1	Reviewing input screens and reports, ask the user if there is any other information that they think should be there that is not. Remember to ask if there is anything already included that should not be, either because it is not used or because nobody knows what it is for.
2	Ask the user to talk about the proposed new information: why they think it is necessary, what reports it applies to, and so on.
3	Analyze the comments for entities and attributes as before.

Identify New Data and Reports Required by Management

The following procedure describes how to perform a follow-up interview with a manager with the intent of identifying new system inputs and outputs.

Step	Action
1	Reviewing reports, ask the manager what they do with the information and how they do it. Ask if there is any other information that they think should be in the report that is not. Remember to ask if there is anything already included that should not be, either because it is not used or because nobody knows what it is for.
2	Ask the manager how they use existing reports.
3	If they *do* use any of the existing reports, ask them to review the output and ask about new subject matter, columns that can be eliminated, and so on. Have them notate the reports with respect to the new information requested and have them justify their request by telling you why they think the new information is necessary.
4	Analyze the comments for entities and attributes as before.
5	Review results of your analysis with the subject manager.

Determine the Final List of Attributes Derived from the Interviews

The following procedure describes how to perform a follow-up interview with a manager with the intent of identifying new system inputs and outputs.

Step	Action
1	Review and refine the attribute list, eliminating duplicates and grouping like attributes into clusters.
2	Determine which of the attributes are derived (that is, not atomic) and separate them from the atomic attributes.
3	Review the attribute lists with both management and the user base.

Use Your Approved Data to Begin the Logical Design of the Database

After your subjects have approved the list of entities, attributes, and reports you have compiled, begin the entity relationship modeling process.

Chapter 4:

Semantic Data Modeling

This chapter describes the basics of semantic, or logical, data modeling, focusing on entity-relationship analysis.

The normal forms, the process of normalizing the database schema, dimensional modeling, and various aspects of denormalization are described in Chapter 5: "The Normalization Process."

Teradata has created several different industry-specific logical data model (LDM) frameworks, including LDMs for the following industries:

- Communications
- E-Business
- Financial Management
- Financial Services
- Manufacturing
- Retail
- Travel

Contact your Teradata representative for details about the availability of these LDMs.

The Entity-Relationship Model

The Relational Model for Database Management

After E.F. Codd introduced the relational model for database management in 1969, a number of alternative models immediately began to appear in the literature. The usual justification given for introducing these new models was a perceived failure of the relational model to capture enough semantics about the data. The failure of the original relational model to capture a satisfactory level of data semantics was acknowledged by Codd himself, who has extended the model several times in an attempt to better capture data semantics. This effort has even led to a second version of the relational model, which is presented in his book *The Relational Model for Database Management, Version 2* (Codd, 1990).

The Entity-Relationship Model for Database Management

Peter Chen (Chen, 1976) introduced the Entity-Relationship (E-R) model as a data model that was intended to replace the relational model. The E-R model was a serious attempt to extend the semantics of the relational model to capture more meaning from the data stored in a database. There is a naive appeal to the E-R model because it is based on a far less abstract approach to the world than that used by the relational model. In the E-R approach, a real world object is captured as an *entity*. Each entity has certain unique characteristics that are captured as *attributes*. Just as real world objects interact with one another, so do entities in the E-R model, and they do this by way of *relationships*. Of all the alternatives to the relational model proposed since its introduction, the Entity-Relationship model has perhaps garnered the most support, albeit not in the manner its author had intended.

In large part, the failure of the E-R model to catch on as a serious data model derives from its informality: the very attribute that makes it so appealing in the first place. Because the E-R model is not derived from formal principles, it is art masquerading as science, greatly lacking in precision and predictability, both of which are the most appealing attributes of the relational model.

Of all the models proposed to replace the relational model, Codd himself finds the E-R model to be the most objectionable. He writes, “The major problem in the entity-relationship approach is that one person’s entity is another person’s relationship. There is no general and precisely defined distinction between these two concepts, even when discussion is limited to a particular part of a business that is to be modeled by means of a database. If there are 10 people in a room and each is asked for definitions of the terms “entity” and “relationship,” 20 different definitions are likely to be supplied for each term” (Codd, 1990, p. 477).

The lack of precision in the definition of relationships in the Chen model is underscored by the ongoing research effort to generalize the concept and to provide a formal definition 25 years after the E-R model was originally described (see, for example, Dey, Storey, and Barron, 1999 and Wand, Storey, and Weber, 1999).

Applications of the E-R Model

Because of its intuitive appeal and naive usefulness, the E-R model has become the most widespread technique for deriving an initial logical model for relational databases. Virtually every commercially available CASE tool in existence uses some form of the E-R approach as the basis for its design tools.

It is this use of the E-R model that is presented here.

Entities, Relationships, and Attributes

Definition: Entity

An entity is a database object that represents a thing in the real world. Entities are expressed as nouns.

Entities can be concrete, like buildings and employees or they can be more abstract things like departments and accounts.

Loosely speaking, an entity corresponds to a relation in relational theory. When a relation is made physical, it is normally referred to as a table, though the term is also used to describe physical as well as conceptual tables.

Types of Entities

There are several different schemes for categorizing entities based on qualitatively different criteria. For purposes of this manual, the following two schemes are defined.

- Major and minor entities
- Supertype and subtype entities

The following table provides definitions for these types.

Entity Type	Definition	Example
Major	An entity with relatively large cardinality and degree that is updated frequently.	Order table
Minor	An entity with small cardinality and degree that is rarely updated. Minor entities are typically used in a single, `1:M` association, and their primary key is often nonnumeric.	Nation Code table
Supertype	A generic entity that is a superclass of one or more subtype entities. Supertype and subtype entities model the same real world entity at a high level. Supertypes must, by definition, have one or more reciprocal subtypes.	Publications table
Subtype	A specific entity that is a disjoint subclass of one and only one supertype entity. Subtype and supertype entities model the same real world entity at a high level. Subtype entities typically have a higher degree than their supertypes, with the additional attributes describing detailed characteristics of the subtype that distinguish it from the other subtype entities of a mutual supertype.	Book table Magazine table Professional journal table Conference proceedings (all as subtypes of the supertype Publications)

Definition: Relationship

A relationship is an association among two or more entities or other relationships. Relationships are expressed as verbs.

Relationships among entities are described by one of three ratios:

Relationship	Shorthand Notation
One-to-one	`1:1`
One-to-many	`1:M`
Many-to-many	`M:M`

Relationships have no direct counterpart in relational theory. The closest property of relational theory to expressing what a relationship is in E-R theory is the primary key-foreign key relationship.

"Relationship Theory" on page 4-9, "One-to-One Relationships" on page 4-11, "One-to-Many Relationships" on page 4-12, and "Many-to-Many Relationships" on page 4-13 describe the properties of relationships in greater detail.

Definition: Attribute

An attribute is a characteristic of an entity. Every entity has at least one attribute: its primary key. Attributes are expressed as nouns qualified by adjectives that clarify their role.

An attribute plays one of three possible roles in any table.

- Primary key attributes identify the entity or relationship modeled by a table (note the fuzzy boundaries among entities, relationships, and tables, which led Codd to his pointed criticism of the E-R approach).

  Primary key attributes are said to be *identifier* attributes because they uniquely identify an instance of an entity.
- Foreign key attributes define relationships between and among entities or among entities and relationships.

  A foreign key attribute can be an identifier attribute if it is part of a composite primary key; otherwise, foreign key attributes are descriptor attributes.
- Nonkey attributes further describe the entity or relationship modeled by a table.

  Nonkey attributes are said to be *descriptor* attributes because they specify a nonunique characteristic of an instance of an entity.

In the relational model, attributes have the same properties as they do in E-R theory.

Definition: Derivative Attribute

So-called derivative attributes violate the rules of normalization in relational theory because they are not atomic. A derivative attribute is any attribute that can be derived by calculation from other data in the model.

The issue of derivative attributes should not concern you during the logical design phase other than knowing that they should not be modeled. Derivative attributes are an important consideration for physical database design, where they are often modeled as a means for enhancing system performance.

Note that Teradata offers several features like hash and join indexes, aggregate join indexes, and global temporary tables that lessen the temptation to denormalize the physical design of your base tables by using derivative attributes.

Translating Entities and Relationships Into Tables

Introduction

This topic describes how entities and relationships are mapped to tables.

Definition: Prime Table

A prime table is a table that has a single column primary key.

Prime tables always model entities and all entities are modeled by prime tables.

Ensure that all prime tables have been defined prior to defining any associative tables (see "Definition: Associative Table" on page 4-8).

Example: Prime Table

The following table is prime because it has a single-column primary key.

A
A_Key
PK
A1
A2
A3

Definition: Non-Prime Table

A non-prime table is a table that has a composite primary key.

Example: Non-Prime Table

The following table is non-prime because it has a composite primary key.

A	
A_Key_1	A_Key_2
PK	
A1_1	A2_3
A1_2	A2_2
A1_3	A2_1

Definition: Associative Table

An associative table is a non-prime table whose primary key columns are all foreign keys.

Because associative tables model pure relationships rather than entities, the rows of an associative table do not represent entities. Instead, they describe the relationships among the entities the table represents.

Always define all your prime tables before defining any associative tables.

Guideline for Naming Associative Tables

Use the following general form to name associative tables.

```
prime_table_name_A-prime_table_name_B
```

This convention helps to keep the alphabetic and logical sequences of tables synchronized, making it easier to locate the prime tables for the foreign keys that make up its composite primary key.

Example: Associative Table

The following associative table associates prime table entity A with prime table entity B.

A-B	
A_Key	**B_Key**
PK	
FK	FK
A1	B1
A2	B5
A3	B2

Relationship Theory

Introduction

This topic describes the various entity relationships in some detail.

Definition: Degree

The degree of a relationship is the number of entities associated in the relationship. Typical degree descriptions are provided in the following table.

For a relationship among this many entities ...	The following term is commonly used to describe the relationship ...
1	Unary
2	Binary
3	Tertiary
n	*n*-ary

Definition: Connectivity

Connectivity refers to the mapping of entity occurrences in a relationship. The possible values for the connectivity of a relationship are three:

- 0
- 1
- Many

Definition: Cardinality

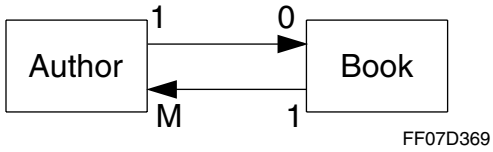
With respect to E-R theory, the term cardinality refers to the integer number aliased by the word *many* in a `1:M` or `M:M` relationship. This number describes the constraint on the number of entity instances that are related through the relationship.

In this context, cardinality does *not* refer to the number of tuples in a relation.

Definition: Existence Dependency

Whenever the existence of an entity depends on the existence of another entity, that relationship is described as an existence dependency.

For example, suppose there are two entities named Author and Book. A writer might have been provided with an advance to write a book for a publisher, but if this is the first book written by the writer for this publisher, there will be no entry in the Book entity until the contracted book has been written and published. In this case, the relationship between Author and Book is 1:0, indicating that the existence of an instance of Book is optional.



One-to-One Relationships

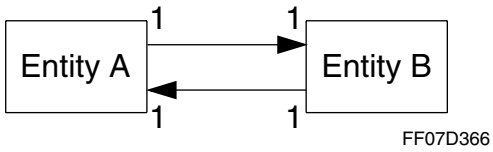
Definition

Assume two entities, A and B.

A `1:1` relationship exists between A and B when each occurrence of A is related to at most one occurrence of B, while each occurrence of B is related to at most one occurrence of A.

`1:1` relationships are not commonly seen in real world situations.

`1:1` relationships are graphed as follows.



Modeling 1:1 Relationships

`1:1` relationships are modeled by placing the primary key of A as a foreign key component of B with no duplicates allowed. Because the relationship is symmetrical, you could just as well place the primary key of B as a foreign key component of A.

Guideline for Placing the Foreign Key

Place the foreign key in whichever entity minimizes or eliminates the possibility of nulls.

Example

Because of the symmetry of `1:1` relationships, the following table pairs both model the same relationship.

A

A_Key
PK
A1
A2
A3

B

B_Key	A_Key
PK	**FK, ND**
B1	A1
B2	A3
B3	A5

A

A_Key	B_Key
PK	**FK, ND**
A1	B1
A2	B5
A3	B2

B

B_Key
PK
B1
B2
B3

One-to-Many Relationships

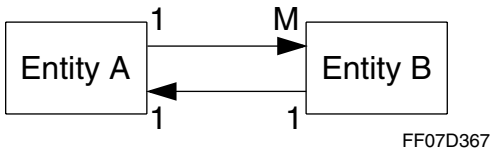
Definition

Assume two entities, A and B.

A `1:M` relationship exists between A and B when each occurrence of A is related to 0, 1, or more occurrences of B, while each occurrence of B is related to at most one occurrence of A.

`1:M` relationships are commonly seen in real world situations. For example, each department has many employees, but each employee has only one department.

`1:M` relationships are graphed as follows.



Modeling 1:M Relationships

`1:M` relationships are modeled by placing the primary key of A as a foreign key component of B.

Example

Because `1:M` relationships are asymmetric, the A key must be placed in B. The reciprocal relationship does not model the same E-R relationship.

A

A_Key
PK
A1
A2
A3

B

B_Key	A_Key
PK	**FK**
B1	A1
B2	A3
B3	A1
B4	A1
B5	A3

Many-to-Many Relationships

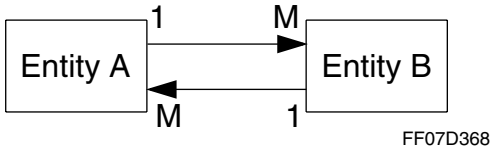
Definition

Assume two entities, A and B.

A M:M relationship exists between A and B when each occurrence of A is related 0, 1, or many occurrences of B and each occurrence of B is related to 0, 1, or many occurrences of A.

M:M relationships are commonly seen in real world situations.

M:M relationships are graphed as follows.



Modeling M:M Relationships

M:M relationships are modeled by placing the primary key of A and the primary key of B as foreign keys comprising the entire primary key of a separate entity referred to as an associative table. This is another example of the subjectivity of E-R theory, because a relationship is expressed as a physical table in the database.

Guideline for Drawing M:M Tables for Users

Draw tables that represent M:M relationships in a way that data is presented in a form most familiar to users.

Example

The following set of tables models the M:M relationship between A and B.

A

A_Key
PK
A1
A2
A3

B

B_Key
PK
B1
B2
B3

A-B

A_Key	B_Key
PK	
FK	FK
A1	B1
A2	B5
A3	B2

Moving From an Entity-Relationship Analysis to Normalization

The goal of an E-R analysis is to generate a reasonable set of entities and to map the relationships among them.

The next step in the process of achieving a logical data model for your database is to translate those entities into a set of fully normalized tables. The principal task of this translation is to cluster attributes with entities, or relations, in such a way that they uniquely describe the primary keys of those relations. Attributes that do not describe the primary key of a relation must be migrated to a relation whose primary key they *do* describe uniquely.

The topics of normal forms and the normalization process are described in Chapter 5: “The Normalization Process.”

Semantic Modeling References

Topic	Reference
Entity-Relationship Model	Peter Pin-Shan Chen, "The Entity-Relationship Model—Toward a Unified View of Data," *ACM Transactions on Database Systems*, 1(1):9-36, 1976. Introduces the concepts of entities and the relationships among them as an alternative data model to the relational model of Codd. The E-R approach has become a guiding principle for logical database design, though it has had virtually no impact as a concrete data model.
Extended E-R Model for Relationships	Debabrata Dey, Veda C. Storey, and Terence M. Barron, "Improving Database Design Through the Analysis of Relationships," *ACM Transactions on Database Systems*, 24(4):454-486, 1999. Informally extends the concept of relationships beyond the binary model described by Chen and other workers to make binary relationships a special case of *n*-ary relationships among multiple entities.
Formal Support for Defining Relationships	Yair Wand, Veda C. Storey, and Ron Weber, "An Ontological Analysis of the Relationship Construct in Conceptual Modeling," *ACM Transactions on Database Systems*, 24(4):494-528, 1999. Attempts to formalize the concept of entity relationships using a abstract model derived from the philosophical concepts of ontology and ontological frameworks.
The Codd Critique of E-R Modeling	E.F. Codd, *The Relational Model for Database Management, Version 2*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1990. Criticizes E-R modeling for its vague definitions and lack of formal grounding.

Chapter 5:

The Normalization Process

This chapter reviews some concepts of the normalization process. The assumption is made that any designer will use a CASE tool that has its own set of notational structures and conventions, so the presentation is as generic as possible. Neither is the presentation intended to provide any Teradata-specific information or to instruct a novice on how to normalize a database.

The following topics are described in the chapter.

- Keys
- Functional, transitive, and multivalued dependencies
- Normal forms
- Decomposing relations into the various normal forms
- Identifying primary keys
- Foreign keys
- Referential integrity
- Denormalization

The Key, the Whole Key, and Nothing But the Key

Introduction

William Kent (1983) sums up the goal of normalization in a single, pithy phrase in his classic summary paper in the *Communications of the ACM* titled "A Simple Guide to Five Normal Forms in Relational Database Theory": "The Key, the Whole Key, and Nothing But the Key" (to which some individual with a clever sense of humor appended the closer "So help me Codd").

The normalization process helps you to structure your thinking about the entities you have identified and the relationships they share among one another. It heightens your awareness of the problems that can occur when all the attributes of a schema are not sorted out and stored in one and only one place.

Normalization Is a Logical Process

It is important to understand that database normalization is a logical process; therefore, the term denormalization is usually a misnomer because it typically applies to the physical design of the database rather than its logical derivation. The application of denormalization to the logical model, as endorsed by advocates of Dimensional Modeling, is never a good idea for an enterprise data warehouse.[1]

Because database management systems map logical relations directly to physical tables, it can sometimes appear difficult to separate the logical model from its physical realization. Nevertheless, you should always design a fully normalized logical model, then, and only if absolutely necessary to achieve performance levels that are otherwise not possible to realize, denormalize the physical design.

The key to retaining the integrity of the relational model is to delineate this denormalized physical design in such a way that it continues to present the appearance of full normalization to users. This can be achieved by the use of carefully designed views.

1 You might be confused by this statement. After all, most "expert" advocates of data warehousing begin with the assumption that a dimensional database design is a necessity. The reason for this is that most commercially available database engines are not capable of delivering the level of performance required to support massive data warehouses. Because of its innate parallel architecture, Teradata easily supports multi-terabyte databases within a physical design that is directly derived from a fully normalized logical design.

Because of the widespread usage of the term, this manual sometimes refers to denormalization in the context of physical database design. This is done with the understanding that the logical and physical models of a database are independent things, that the physical model applies *only* to implementation, and that the term denormalization strictly applies only to logical modeling.

The Cost of Normalization

There is a cost to normalization when the logical model is physically implemented. In a very real sense, normalization optimizes update performance at the expense of retrieval performance. Because of this, it sometimes becomes necessary to denormalize relations to some extent in order to achieve reasonable overall system performance.

Before examining the details of the normalization process, a number of properties of relations and their manipulation need to be spelled out.

Properties of Relations and Their Logical Manipulation

Introduction

This topic examines some basic properties of relations and their logical manipulation.

Definitions of Commonly Used Relational Terms

The following table defines some terms that are used throughout any discussion of normalization.

Term	Definition
Alternate key	Any candidate key not selected to be the primary key for a relation.
Attribute	A property of a relation that describes its primary key. Each attribute has a unique name, is drawn from a domain, and can be constrained in various ways. Because its values are all drawn from the same domain, the data for any attribute is homogeneous by definition. Attribute is the term used in set theory and logical design. Column is the equivalent term used in physical design and database management.
Candidate key	An set of one or more attributes that uniquely identifies a tuple. A candidate key has the following minimal properties. • The value of the key uniquely identifies the tuple in which it appears. • Attributes defining the candidate key cannot be redundant. If an attribute is removed from the candidate key, then its uniqueness must be destroyed else it is not a properly defined candidate key. Compare with "Superkey."
Composite key	A primary or foreign key defined on more than one attribute. Compare with "Simple key."
Domain	The set of all possible values that can be specified for a given attribute. The physical representation of a domain is a data type.
Field	The intersection of a tuple and an attribute.

Term	Definition
Foreign key	A foreign key is an attribute set based on an identical attribute set that acts as the primary key for, or as an alternate key in, a different relation. Foreign keys reflect relationships between tables and are often used as join columns.
Instance	A tuple drawn from the complete set of tuples for a relation. The term is sometimes used to describe any selected set of tuples from a relation.
Intelligent key	An overloaded simple key that encodes more than one fact[a], which is a violation of 1NF. The principal implementation problem with intelligent keys is that if any of the components of the key change, then all applications that access the key are affected. This is why you should always select unchanging attributes for your keys. The classic example of an intelligent key is the International Standard Book Number (ISBN) used by publishers to identify individual books. Each ISBN is composed of a group identifier, a publisher identifier, a title identifier, and a check digit.
Key	An attribute set that uniquely identifies each tuple in a relation. Implicitly synonymous with primary key. See "Primary key" on page 5-5.
Natural key	The representation of a real world tuple identifier in a relational database. For example, a common identifier of employees in a corporation is a unique employee number. An employee is assigned an employee number whether that information is stored within the database or not. Natural keys are sometimes confused with intelligent keys, but they are very different concepts.
Primary key	The primary key for a relation is a attribute set that uniquely identifies each tuple in the relation. By the Entity Integrity Rule, primary keys cannot contain nulls.[b] Every relation must have one and only one primary key. The primary key is often used as the primary index for a table when a relation is manifested physically.
Relation	A two-dimensional representation of data in tabular form. Relation is the term used in set theory and logical design. Table is the equivalent term used in physical design and database management.
Relational schema	A set of relations in a logical relational model. In the physical model, a relational schema is manifested as a database.
Repeating group	A collection of logically related attributes that occur more than once in a tuple.

Term	Definition
Simple key	A primary or foreign key defined on a single attribute. Compare with "Composite key."
Superkey	Any set of attributes that uniquely identifies a tuple, whether redundantly or not. This permission of redundancy distinguishes a super key from a "Candidate key."
Surrogate key	An artificial, non-composite, key used to identify individual entities when there is no natural key or when the situation demands a non-composite key, but no natural non-composite key exists. Surrogate keys do not identify individual entities in a meaningful way: they are simply an arbitrary method to distinguish among them. Surrogate keys are typically arbitrary system-generated sequential integers. See "Identity Columns" on page 14-13 and "CREATE TABLE" in *Teradata RDBMS SQL Reference, Volume 4*, for information about how to generate surrogate keys in Teradata.
Tuple	A unique instance of a relation consisting of, at minimum, a primary key and zero or more attributes that describe the primary key. Tuple is the term used in set theory and logical design. Row is the equivalent term used in physical design and database management. Nonrelational systems use the term record in the same way relational systems use row.

a. A combination of identification as well as characterization facts.
b. See "Rules for Primary Keys" on page 5-34 for an explanation of the Entity Integrity Rule.

Some Properties of Relations

Relations have the following additional properties.

Property	Description
Order independence	Ordering of tuples and attributes has no meaning. When attributes are manifested physically as columns, their order does have significance for SQL.
Uniqueness	Duplicate rows are not permitted. When relations are manifested physically as tables in SQL databases, duplicate rows are permitted for multiset tables only. This is not a recommended practice: you should always define your tables as set tables to avoid the many problems that multiset tables present.

Logical Operations on Relations

Relations are formally decomposed and constructed using various logical operators.

The first four operators in the following table derive directly from set theory, while the last four were developed by E.F. Codd specifically to suit the needs of relational databases. These operators are described in detail in his 1969 and 1970 papers.

Logical Operator	Description
Set Theory Operators	
DIFFERENCE	X - Y The set of all attributes contained in relation X but not in relation Y.
INTERSECTION	X ∩ Y The set of all attributes contained in both relation X and relation Y.
PRODUCT	X · Y The set of all multiples of all attributes contained in relations X and Y.
UNION	X ∪ Y The set of all attributes contained in either relation X or relation Y or both.
Special Relational Database Operators	
DIVIDE	The division of relation R of degree $m + n$ by relation S of degree n produces a quotient relation of degree m.
JOIN	The join of relation R on attribute X with Relation S on attribute Y is the set of all tuples t such that the concatenation of a tuple r, belonging to R, and a tuple s, belonging to S, and the predicate r.R *equality_operator* s.S evaluates to TRUE. In SQL, this is expressed in an ON clause in a DML join statement.
PROJECTION	The projection of relation R on attributes X, Y, …, n is the set of all tuples (x, y, …, n) such that a tuple n appears in R with X value x, Y value y, and so on. Less formally, a projection on relation R is any subset of the attributes of R. In SQL, this is expressed as a column_list in a DML statement.
RESTRICTION/ SELECTION	The restriction of relation R is the set of all tuples t such that the comparison t.X operator t.Y evaluates to TRUE. Less formally, a restriction (or selection) on relation R is any subset of the tuples of R satisfying the condition X *equality_operator* Y. In SQL, this is expressed in a WHERE clause in a DML statement.

Functional, Transitive, and Multivalued Dependencies

Introduction

Attribute dependencies provide the foundation upon which normalization is performed. 1NF, 2NF, 3NF, and BCNF all derive from functional dependencies, while 4NF derives from multivalued dependencies, and 5NF derives from projection and join dependencies.

Dependency theory is analogous to the theory of functions in algebra where, for example, the value of y in an equation depends on the value for x and is expressed as $y = f(x)$.

Dependency theory provides a set of inference axioms to determine which attributes depend on the primary key of a relation (or on one another) and which do not.

Definitions

The following table defines some important dependency terminology.

Term	Definition
Determinant	For any relation R, the set of attributes X in the functional dependency $X \rightarrow Y$ is the determinant group for the dependency.
Full functional dependence	An attribute set X is fully functionally dependent on another attribute set Y if X is functionally dependent on the whole set of Y, but not on any subset of Y.
Functional dependence	For any relation R, attribute X is functionally dependent on attribute Y if for every valid instance, the value of Y determines the value of X. The symbolic notation for this is $Y \rightarrow X$.
Multivalued dependence	For any relation R with attribute set (X,Y,Z), the set X multivalue determines the set Y if for every pair of tuples containing duplicates in X then the instance also contains the pair of tuples obtained by interchanging the Y tuples in the original pair. The symbolic notation for this is $X \Rightarrow Y$.

Term	Definition
Transitive dependence	Transitive dependencies are dependencies that exist among three or more attributes in a relation in such a way that one attribute determines a third by way of an intermediate attribute. For example, consider the relation R(X,Y,Z), where X is the primary key. Attribute Z is transitively dependent on attribute X if attribute Y satisfies the following dependencies. • $X \rightarrow Y$ • $Y \rightarrow Z$ • $Y \overline{\rightarrow} X$ where the symbol $\overline{\rightarrow}$ indicates does *not* determine.

Inference Axioms for Functional Dependencies

Beeri, Fagin, and Howard (1977) produced a complete set of inference axioms for functional and multivalued dependencies for relations. Their axioms are described in the following table.

Axiom	Formal Expression
Reflexive rule	$X \rightarrow X$
Augmentation rule	IF $X \rightarrow Y$ THEN $XZ \rightarrow Y$
Union rule	IF $X \rightarrow Y$ AND $X \rightarrow Z$ THEN $X \rightarrow YZ$
Decomposition rule	IF $X \rightarrow Y$ THEN $X \rightarrow Z$ WHERE Z is a subset of Y
Transitivity rule	IF $X \rightarrow Y$ AND $Y \rightarrow Z$ THEN $X \rightarrow Z$
Pseudotransitivity rule	IF $X \rightarrow Y$ AND $YZ \rightarrow W$ THEN $XZ \rightarrow W$

For the sake of some examples, assume the attribute set R, S, T, U, V with the following set of dependencies.

- $R \rightarrow S$
- $TU \rightarrow R$
- $T \rightarrow V$
- $V \rightarrow T$
- $SU \rightarrow T$

The following table lists many of the dependencies among these attributes implied by the inference axioms.

Dependency	Supporting Axiom
$R \rightarrow R$	Reflexive rule
$RT \rightarrow S$	Augmentation rule
$TU \rightarrow RV$	• Augmentation rule • Union rule
$RU \rightarrow T$	Pseudotransitivity rule
$UV \rightarrow R$	Pseudotransitivity rule
$TU \rightarrow S$	Transitivity rule
$SU \rightarrow V$	Transitivity rule
$VU \rightarrow R$	Pseudotransitivity rule
$RU \rightarrow V$	• Transitivity rule • Pseudotransitivity rule
$UV \rightarrow S$	• Transitivity rule • Pseudotransitivity rule

Inference Axioms for Multivalued Dependencies

The following table lists the complete set of multivalued dependencies derived by Beeri, Fagin, and Howard in their 1977 paper.

Axiom	Formal Expression
Reflexive rule	$X \Rightarrow X$
Augmentation rule	IF $X \Rightarrow Y$ THEN $XZ \Rightarrow Y$
Union rule	IF $X \Rightarrow Y$ AND $X \Rightarrow Z$ THEN $X \Rightarrow YZ$
Decomposition rule	IF $X \Rightarrow Y$ AND $X \Rightarrow Z$ THEN $X \Rightarrow Y \cap Z$ AND $X \Rightarrow (Z - Y)$
Transitivity rule	IF $X \Rightarrow Y$ AND $Y \Rightarrow Z$ THEN $X \Rightarrow (Z - Y)$
Pseudo-transitivity rule	IF $X \Rightarrow Y$ AND $YW \Rightarrow Z$ THEN $XW \Rightarrow (Z - YW)$
Complement rule	IF $X \Rightarrow Y$ AND $Z=R-XY$ THEN $X \Rightarrow Z$
Mixed inference rules	IF $X \rightarrow Y$ THEN $X \Rightarrow Y$
	IF $X \Rightarrow Y$ AND $Z \Rightarrow W$ THEN $X \rightarrow W$ WHERE W is contained in Y AND $Y \cap Z$ is not empty

The Normal Forms

Introduction

Normalization theory is constructed around the concept of a universe of normal forms that define a system of constraints. If the form of a relation meets the constraints of a particular normal form, it is said to be *in* that form.

You can think of the normal forms as an onion, with the outermost layer being the set of all relations, including unnormalized relations. As you work your way to the core of the onion, you must pass through each lower normal form. As a result, a relation that has achieved fifth normal form has also achieved first, second, third, and fourth normal forms.

By definition, a relational database is always normalized because its field values are always atomic. In other words, all relations are in 1NF by definition. This means, in the strict sense, that it is not possible to have unnormalized relations. But to simply leave it at that invites a number of problems with redundancy and potential update anomalies, which is why the higher normal forms were developed. In this manual, the expression unnormalized table (or unnormalized relation) means not normalized beyond 1NF, and the topics usually described casually under the heading *normalization* technically should be described as *further normalization*.

The Objective of Normalization

The intent of normalizing a relational database can be reduced to one simple aphorism: One Fact In One Place. By decomposing your relations into normalized forms, you can eliminate the majority of update anomalies that can occur when data is stored in unnormalized tables. Decomposition is attained through a series of projections of a relational schema into a set of normalized relations that optimize the concision of attributes.

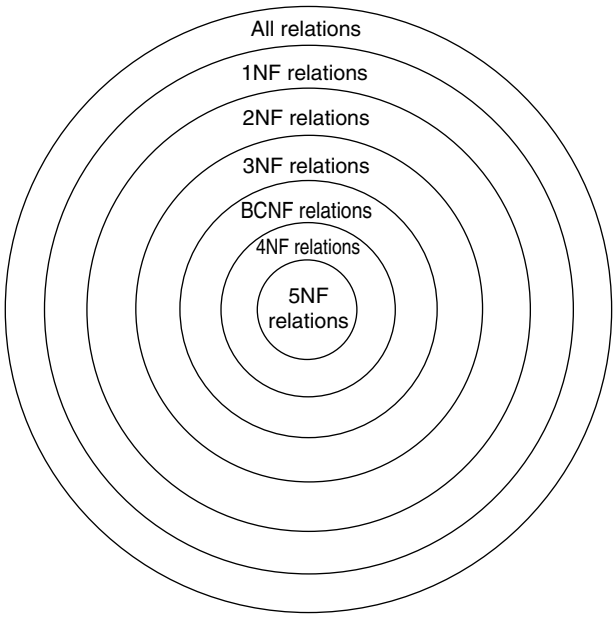
A slightly more detailed statement of this principle would be the definition of a relation (or table) in a normalized relational database: A relation consists of a primary key, which uniquely identifies any tuple, and zero or more additional attributes, each of which represents a single-valued (atomic) property of the entity type identified by the primary key.

Advocates of dimensional modeling argue against the method of decomposition because it tends to produce "too many" relations, whatever that means. When you design with dimensional modeling in mind, you typically create only a few so-called “fact” tables with a composite primary key that is not free of transitive dependencies and smaller satellite tables containing detailed data about the “fact” table.

When you create a dimensional logical model, you are denormalizing your database in the strict sense of the term. This is never a good idea.

The Universe of Relations

The following diagram illustrates how the normal forms are like the layers of an onion, progressing from unnormalized relations inward to the center, where all relations meet the definition of Fifth Normal Form.



FG04A001

The following normal forms are described here.

- "First Normal Form" on page 5-13
- "Second Normal Form" on page 5-15
- "Third and Boyce-Codd Normal Forms" on page 5-18
- "Fourth Normal Form" on page 5-23
- "Fifth Normal Form" on page 5-26
- "Other Normal Forms" on page 5-27
- "General Procedure for Achieving a Normalized Set of Relations" on page 5-30

First Normal Form

Introduction

First normal form (1NF) is definitive for a relational database. All relations in a relational database are in first normal form by definition. Because of this property, the formal name for the normalization process is *further normalization*.

Definition

A relation is said to be in First Normal Form (1NF) when all its fields (simple domains in mathematics) are atomic. This means that a field can contain one value and one value only. No hierarchies of data values are allowed. This concept is sometimes referred to as the elimination of repeating groups from a relation.

Another way of stating the rule is this: the relationship between the primary key of an entity and each of its nonkey attributes must be one-to-one in that direction.

The formal definition is as follows: For a relation to be in first normal form, the relationship between its primary key and each of its other attributes must be one-to-one (in that direction). In other words, all underlying simple domains of the relation contain atomic values only.[2]

The nonkey attributes are said to be functionally dependent on the key, where the term *nonkey attribute* means any attribute that is not part of the primary key for the relation.

2 Unfortunately, there has never been a formal definition for the term *atomic*, and its meaning has evolved considerably since Codd originally formulated the relational model. It is even argued, with considerable formal justification, that relation-valued attributes (that is, relation attributes that are themselves relations) can be atomic. Note that the multivalued attributes advocated by proponents of so-called Non First Normal Form (NFNF, or NF^2) are not atomic in any sense of the term.

Example

For example, consider the following set of relations.

Order

OrderNumber
PK
1
2

Part

PartNumber
PK
1
2

Order-Part

OrderNumber	PartNumber	Quantity
PK		NN
FK	FK	
1	1	110
1	2	275
2	1	152

If we had designed these prototype relations in such a way that the Quantity attribute was in either the Order or the Part relation, 1NF would have been violated because there is more than one quantity for Order Number 1 and more than one quantity for Part Number 1. Multiple quantities in these cases constitute a repeating group.

Second Normal Form

Introduction

Second normal form (2NF) deals with the elimination of circular dependencies from a relation.

Definition

A relation is said to be in Second Normal Form (2NF) when the following criteria are satisfied:

1. The relation is in 1NF.
2. Every nonkey attribute is fully functionally dependent on the entire primary key. This means that no nonkey attribute can be functionally dependent on a partial key.

The formal definition is as follows: For a relation to be in second normal form, the relationship between any portion of the primary key of a relation and each of the other attributes must *not* be one-to-one (in that direction). In other words, the nonkey attributes are *fully* functionally dependent on the key.

Another way to state 2NF is as follows. A relation in 1NF is also in 2NF if any one of the following statements is also true.

- The primary key is not composite.
- There are no nonkey attributes in the relation.
- Every nonkey attribute is dependent on the complete set of primary key attributes.

The first two bullets in this list are trivial cases. It is the third bullet that provides a truly generalized definition for 2NF.

The following restatement of 2NF is perhaps the most straightforward: The relationship between any portion of the primary key of a relation and each of the other attributes must not be one-to-one in that direction.

Example

For example, consider the following set of relations.

Order

Order Number	Order Date
PK	
1	2000/01/15
2	

Part

Part Number	Part Name	Part Price
PK	NN, ND	
1	Nut	1.15
2	Bolt	2.75

Order-Part

OrderNumber	PartNumber	Quantity
PK		NN
FK	FK	
1	1	110
1	2	275
2	1	152

Example Violating 2NF

If we had designed these relations so that the Part Name attribute was in the Order-Part relation, 2NF would have been violated because there would be a one-to-one relationship between the PartNumber attribute of the primary key and the Part Name attribute.

Suppose we were to structure the Order-Part relation like this.

Order-Part

OrderNumber	PartNumber	PartName	Quantity
PK			NN
FK	FK		
1	1	Nut	110
1	2	Bolt	275
2	1	Nut	152

The potential update anomalies associated with this violation of 2NF are the following.

- Changed part name. To change the name of a part, you must find every occurrence of its part name.
- Inconsistent part names. Nut might be misspelled Nutt in an occurrence, which would cause the tuple containing the misspelled part name to be missed in a query having the WHERE predicate partname = nut.

  Worse still, part number 2 might be labeled Bolt in one tuple and Screw in another.

Third and Boyce-Codd Normal Forms

Introduction

Third Normal Form (3NF) deals with the elimination of nonkey attributes that do not describe the primary key.

Definition: 3NF

A relation is said to be in Third Normal Form (3NF) when it is in 2NF and one of the following statements is also true for every nontrivial functional dependency. Assume the functional dependency $X \rightarrow A$.

- The attribute set *x* is a superkey.
- Attribute A is part of some candidate key.

Another way of stating the rule is this: The relationship between any two nonkey attributes or attribute sets (excluding attributes having *no duplicates allowed* constraints) must not be one-to-one in either direction.

Using the phrase coined by Kent (1983), the relation is in 3NF when all its attributes depend on the key, the whole key, and nothing but the key. This condition is met when both of the following criteria have been met.

- Every nonkey attribute depends on all attributes of the primary key: the *entire* primary key.
- No nonkey attribute is functionally dependent on another nonkey attribute of the relation.

The formal definition for Third Normal Form is as follows: For a relation to be in 3NF, the relationship between any two nonprimary key attributes or groups of attributes in a relation must not be one-to-one in either direction. In other words, the nonkey attributes are nontransitively dependent upon each other and the key. Having no transitive dependencies in a relation implies no mutual dependencies.

Attributes are said to be mutually independent if none of them is functionally dependent on any combination of the others. This mutual independence ensures that individual attributes can be updated without any danger of affecting any other attribute in a tuple.

This is an incomplete definition for 3NF that fails to account for the case where functional or transitive dependencies occur. To account for this special case, Codd and Raymond Boyce completed 3NF with the definition described in "Definition: BCNF" on page 5-19.

Definition: BCNF

When the relational model of database management was originally proposed, it only addressed the first three normal forms. Later theoretical work with the model showed that 3NF required further refinement to eliminate certain update anomalies.

The classic definition for third normal form does not handle situations in which a relation has multiple composite candidate keys with overlapping attributes. While this situation does not occur frequently, it does present itself from time to time, and once the problem was recognized, a solution had to be found.

To eliminate the deficiencies of the classic 3NF definition, Codd and Boyce developed what has traditionally been called Boyce-Codd Normal Form (BCNF), which reduces to 3NF whenever the special situation that defines this problem does not apply.

A relation is in BCNF if and only if every determinant is a candidate key. This means that only determinants can be candidate keys.

Somewhat more formally, a relation is said to be in Boyce-Codd Normal Form (BCNF) when it is in 2NF and the following is true: if whenever $X \rightarrow A$ and A does not belong to X, then X is a superkey.

Relationship Between 3NF and BCNF

Zaniolo (1982) demonstrates that BCNF is strictly stronger than 3NF with an elegant proof. First, consider his definition of 3NF.

Let R be a relation, let X be any subset of the attributes of R, and let A be any single attribute of R.

R is in 3NF iff for every functional dependency $X \rightarrow A$ in R, at least one of the following assertions is true:

- X contains A, making the functional dependency trivial.
- X is a superkey.
- A is contained in a candidate key of R.

If we eliminate the third assertion from consideration, we have the definition for BCNF, indicating clearly that BCNF is a stronger form than 3NF. Note that the third assertion constitutes the inadequacy of the original formulation of 3NF proposed by Codd (1969, 1970).

Meaning of Third Normal Form in This Manual

For purposes of this manual, use of the term 3NF generally covers BCNF as well. Relations in BCNF are often awkward to deal with, however, and are usually more tractable if decomposed into other relations in 4NF.

Example: 3NF

For example, consider the following relations.

Customer

CustomerNumber	CustomerName
PK	
1	Wright
2	Adams

Order

OrderNumber	OrderDate	CustomerNumber
PK		FK
1	2000/01/15	1
2	2000/01/17	2
3	2000/02/01	1

Example Violating 3NF

If we had designed these relations so that the Customer Name attribute was in the Order relation, 3NF would have been violated because there would be a one-to-one relationship between two nonkey attributes, Customer Number and Customer Name.

Suppose we were to structure the Order relation like this.

Order

OrderNumber	OrderDate	CustomerNumber	CustomerName
PK		FK	
1	2000/01/15	1	Wright
2	2000/01/17	2	Adams
3	2000/02/01	1	Wright

The potential update anomalies associated with this violation of 3NF are the following.

- Changed customer name. To change the name of a customer, you must find every occurrence of its customer name.
- Inconsistent customer names. Adams might be misspelled Addams in an occurrence, which would cause the tuple containing the misspelled customer name to be missed in a query having the WHERE predicate customername = Adams.

  Worse still, customer number 2 might be labeled Adams in one tuple and Zoller in another.
- Inability to add new customer names unless they have an order placed.

Example: BCNF

Now consider the following entity, which is in 3NF, but still has some problems.

Schedule

Campus	Class	Section	Date	Building/Room
PK				
Rancho Bernardo	Physical Database Design	1	2000/05/15	A/225
El Segundo	Relational Database Modeling Workshop	2	2000/08/15	ES/16-201
Dayton	Teradata Basics	3	2000/06/30	CTEC/120
Rancho Bernardo	Advanced SQL for the Teradata Database	1	2000/11/23	A/225

Note that the buildings determine campus because no two buildings on any of the NCR campuses have the same abbreviation. As a result, Building/Room → Campus. The relation is not in Boyce-Codd form.

Normalize the relation by decomposing it into the two following BCNF relations.

Campus-Class

Campus	Class	Section	Date
PK			
Rancho Bernardo	Physical Database Design	1	2000/05/15
El Segundo	Relational Database Modeling Workshop	2	2000/08/15
Dayton	Teradata Basics	3	2000/06/30
Rancho Bernardo	Advanced SQL for the Teradata Database	1	2000/11/23

Building-Campus

Building/Room	Campus
PK	
A/225	Rancho Bernardo
ES/16-201	El Segundo
CTEC/120	Dayton

Higher Normal Forms

Formal research in the theory of relational databases has derived several normal forms beyond the first "three" (including BCNF). Many treatments of normalization state that achieving 3NF or BCNF is "good enough." This is not strictly true: when the situations to which 4NF and 5NF apply are found, it is absolutely necessary to normalize to those levels in order to avoid the data manipulation anomalies that will otherwise occur.

For most database schemas, these higher forms are achieved by default. Date and Fagin (1992) have proven that most schemas normalized to 3NF or BCNF are also, by default, normalized to 4NF and 5NF (see “Definition: 4NF” on page 5-23 and “Definition: 5NF” on page 5-26).

Fourth Normal Form

Introduction

Fourth normal form deals with the elimination of multivalued dependencies from relations having no residual functional dependencies.

Definition: 4NF

A relation is said to be in Fourth Normal Form (4NF - see Fagin, 1977) when it is in BCNF and has no anomalies associated with multivalued dependencies.

In other words, the relation is said to be in 4NF iff whenever there is a multivalued dependency (for example, say X multiply determines Y), then all attributes of the relation are also functionally dependent on X.

More formally, a relation is in 4NF if the implying set X of every nontrivial multivalued dependency $X \Rightarrow Y$ is a superkey.

In practice, the need for 4NF is rarely seen, only presenting itself when 3NF or BCNF relations have no remaining functional dependencies but retain multivalued dependencies that must be removed by decomposition through a series of projections into new relations.

Date and Fagin (1992) proved that any relation that is in BCNF is also in 4NF (but not necessarily 5NF) if at least one candidate key in the relation is simple.

See "Lossless Joins" on page 6-5 and "The Lossless Join Relation" on page 6-11 for descriptions on when 4NF is an important consideration to address.

4NF and Joins

Fourth Normal Form eliminates all update anomalies caused by undesirable functional, transitive, and multivalued dependencies. Additional criteria must be evaluated to determine whether a set of 4NF decompositions is valid for join processing because of the so-called lossy join phenomenon in which spurious tuples can be created by joining fully normalized relations that have not been normalized according to lossless join and dependency preservation criteria.

For more information, see "Lossless Joins" on page 6-5, "The Lossless Join Relation" on page 6-11, "Semantic Disintegrity" on page 6-17, and "Dependency Preservation" on page 6-20.

Example: 4NF

Identifying multivalued dependencies is often a difficult task. A common example of multivalued dependence is the so-called all-to-all relationship. The following relationships are all examples of the all-to-all relationship.

- Each individual company manufactures multiple products; the same product is manufactured by multiple companies.
- Each individual company has multiple plant locations; multiple companies have a plant in the same location.
- Each individual company manufactures all its products in all its plants; therefore, all combinations of products and plant locations are valid for an individual company.

Consider the following relation, which is in BCNF, but still has some problems.

Company-Product-Location

CompanyName	Product	PlantLocation
	PK	
NCR	Disk drive 3230	Omaha
NCR	Disk drive 3230	West Columbia
NCR	Disk drive 3230	Hong Kong
NCR	Disk drive 3230	Rancho Bernardo
NCR	Teradata warehouse suite	Hong Kong
NCR	Teradata warehouse suite	Rancho Bernardo
NCR	Teradata warehouse suite	Omaha
NCR	Teradata warehouse suite	West Columbia
SDWTech	Teradata warehouse suite	Tokyo
SDWTech	Teradata warehouse suite	Garden Grove
SDWTech	Teradata warehouse suite	Chicago
SDWTech	Teradata warehouse suite	Atlanta
DataShareBase	Disk drive 3230	San Diego
DataShareBase	Disk drive 3230	Corvallis

Suppose DataShareBase adds three new plant locations that produce the Teradata warehouse suite. The new plants are located in Singapore, Mexico City, and Dublin. This means that three additional tuples must be added to Company-Product-Location.

Worse, suppose NCR adds four new plants, all of which manufacture Disk drive 3230 and the Teradata warehouse suite. Now eight new tuples must be added to account for these new plants. Notice that we are only dealing with two products. Imagine the complication of adding and removing new products, new plants, and new companies on a regular basis and you begin to understand that this is not a fully normalized relation.

Normalize the relation by decomposing it into the two following 4NF relations.

Product

Company	Product
PK	
NCR	Teradata warehouse suite
NCR	Disk drive 3230
SDWTech	Teradata warehouse suite
DataShareBase	Teradata warehouse suite
DataShareBase	Disk drive 3230

Plant

Company	Location
PK	
NCR	Omaha
NCR	West Columbia
NCR	Hong Kong
NCR	Rancho Bernardo
SDWTech	Tokyo
SDWTech	Garden Grove
SDWTech	Chicago
SDWTech	Atlanta
DataShareBase	San Diego
DataShareBase	Corvallis

Fifth Normal Form

Introduction

For each the lower forms, it is possible to normalize relations by decomposing them into two projections. On rare occasions, simple projections are not sufficient to decompose a nonnormal relation into two relations. In these rare instances, fifth normal form (5NF - see Fagin, 1979) is used to decompose the unnormalized relation into three or more projections of the original relation.

Definition: 5NF

A relation is said to be in fifth normal form (5NF - sometimes called projection-join normal form, or PJ/NF) iff every join dependency in the relation is a consequence of the candidate keys of the relation.

This makes 5NF the final possible normal form to be achieved by taking projections and using joins. It is guaranteed to be free of all anomalies that can be removed by taking projections, but not necessarily of all possible anomalies.

Date and Fagin (1992) proved that any relation that is in 3NF is also in 5NF if *every* candidate key in the relation is simple.

Other Normal Forms

Domain/Key Normal Form

Fagin (1981) has identified a sixth normal form that he calls Domain/Key Normal Form. This form seems to have no practice in reality, but the logic underlying its derivation is based on solid formal principles.

Inclusion Dependency Normal Form

Levene and Vincent (2000) have recently identified a normal form that deals with inclusion dependencies, a type of dependency that has received little practical attention. Their normal form, which they call Inclusion Dependency Normal Form, or IDNF, is an extension of BCNF that ensures that a relation in BCNF also is noncircular and key-based.

Advantages of Normalization for Physical Database Implementation

The following list summarizes the advantages of physically implementing a normalized logical model:

- Greater number of relations
  - More primary index choices
  - Optimal distribution of data
  - Fewer full-table scans
  - More joins possible
  - Enhanced likelihood the Optimizer will use the performant merge or nested join methods
  - Optimal data separation
  - Optimal user isolation from the underlying schema
  - Optimal control of data
- Fewer columns per row
  - Optimal application separation
  - Optimal control of data
- Smaller rows
  - Optimal data blocking by the file system
  - Reduced transient and permanent journal space
  - Reduced physical I/O

Decomposing Relations

The principal goal of normalization is to eliminate update anomalies. By decomposing your relations into normalized forms, you can eliminate the vast majority of update anomalies.

Advocates of dimensional modeling[3] argue against the method of decomposition (indeed, they advocate a denormalized logical design!) because normalization tends to produce many relations. This is an implementation issue, not a logical design issue, and it originates in the inability of most vendor database products to make reasonably performant joins. The goal of dimensional modeling is to optimize query performance at the expense of all else.

Because the Teradata architecture is inherently parallel and optimized for join performance, there are few reasons to consider implementing your databases with a dimensional model and *no* reasons to follow the dimensional model for logical design.

3 When you design with dimensional modeling in mind, you typically create only a few so-called “fact” relations with a composite primary key that is not free of transitive dependencies and smaller satellite relations containing detailed data about the “fact” relation.

General Procedure for Achieving a Normalized Set of Relations

Introduction

Although a designer having relatively little experience with normalizing a relational database can often put relations into 2NF and even 3NF without giving the process much thought, the following procedure is provided so that even a novice can pursue a set of steps through to completion and achieve normalization of a fairly simple set of relations.

Procedure

Step	Action
1	Identify the attributes of the database.
2	Group related attributes into relations.
3	Identify the candidate keys for each relation.
4	Select the most useful primary key from among the set of candidate keys.
5	Identify and remove all repeating groups. The result is a relation in 1NF.
6	If any of the resulting relations have identical primary keys, then combine them into a single relation.
7	Identify all functional dependencies between the attributes of a relation and its primary key.
8	Decompose the relations to a form where each nonkey attribute is dependent on all the attributes of the key. Do this by taking projections of the 1NF relation that eliminate any non-full functional dependencies. The result is a set of relations in 2NF.
9	If any of the resulting relations have identical primary keys, then combine them into a single relation.
10	Identify any transitive dependencies in the relations decomposed to this point.

Step	Action
1	Examine relations for dependencies among nonkey attributes.
2	Examine relations for dependencies among key within the primary key.

Step	Action
11	Remove transitive dependencies by decomposition. Do this by taking projections of the 2NF relations produced by steps 8 and 9 that eliminate any transitive functional dependencies. The result is a set of relations in 3NF.
12	If any of the resulting relations have identical primary keys, then combine them into a single relation if and only if no transitive dependencies result from the combination.
13	Remove any remaining functional dependencies from the 3NF set of relations. Do this by taking projections of the 3NF relations produced by steps 8 and 9 that eliminate any remaining functional dependencies where the determinant is not a candidate key. The result is a set of relations in BCNF.
14	Remove any multivalued dependencies that are not also functional dependencies. Do this by taking projections of the BCNF relations produced by step 13. The result is a set of relations in 4NF. Also see Date and Fagin (1992).
15	The likelihood that any join dependencies not implied by the candidate keys remain by this point is virtually nil. If you can identify any such relations, then take projections of them to eliminate the non-implied join dependencies. The result is a set of relations in 5NF. Also see Date and Fagin (1992).

Identifying Primary Keys

Introduction

The primary key for a relation is an attribute set that uniquely identifies each tuple in the relation.

Primary keys do not denote either of the following properties.

- Order

  Tuples within a relation are not ordered in any way.
- Access path

  Keep in mind that normalization is a logical process, not a physical implementation of the database.

  Primary keys are not used to access rows on disk, though they are a frequent choice as the primary index for tables. The primary index does define the storage path and at least one access path for table rows.

Procedure for Identifying Primary Keys

The selection of a primary key for a relation is the final step in a (possible) series of identifications of unique attribute sets as follows.

Step	Action
1	Identify any superkeys that exist in the attribute set. A superkey is any set of (possibly redundant) attributes that uniquely identifies the tuples of a relation. Every relation has at least one superkey and might have only one.
2	Eliminate any redundant, or otherwise unnecessary, attributes in the identified superkeys and produce a candidate key set. A candidate key is a nonredundant attribute set that uniquely identifies the tuples of a relation. Note that it is possible for composite candidate keys to overlap. Every relation has at least one candidate key and often has *only* one candidate key.

Step	Action
3	Select the primary key from the set of identified candidate keys. Selection of the primary key from a set of candidate keys is ultimately an arbitrary decision, but when there are multiple candidate keys to choose from, a good rule of thumb is to select the one having the fewest attributes, particularly if you plan to use the primary key as the primary index for the table. See "Performance Considerations for Primary Indexes" on page 9-46 for specific information on selecting primary indexes to optimize retrieval and hashing performance. The unselected candidate keys, if any, are referred to as alternate keys. Assign a UNIQUE constraint to any alternate key not selected to be the primary key for a relation to ensure its integrity with respect to the Referential Integrity Rule (see "The Referential Integrity Rule" on page 5-37). Every relation has one and only one primary key.

Surrogate Keys

Situations occur where the identification and choice of a simple primary key is difficult, if not impossible. There might be no single column that uniquely identifies the tuples of a relation or, looking ahead to physical design, there might be performance considerations that argue against using a composite key. In these situations, surrogate keys are an ideal solution.

As previously defined (see "Surrogate key" on page 5-6), a surrogate key is an artificial, non-composite, key used to identify individual entities when there is no natural key or when the situation demands a non-composite key, but no natural non-composite key exists. Surrogate keys do not identify individual entities in a meaningful way: they are simply an arbitrary method to distinguish among them.

Surrogate keys are typically arbitrary system-generated sequential integers. See "Identity Columns" on page 14-13 and "Generating Surrogate Keys" in the description of "CREATE TABLE" in *Teradata RDBMS SQL Reference, Volume 4*, for information about how to generate surrogate keys in Teradata.

Rules for Primary Keys

There are several rules that define the bounds of primary keys. The first (the Entity Integrity Rule) and second rules in the list are absolute and the third is strongly advised.

- Primary key attributes cannot be null.

  By definition, nulls are not unique because they represent missing values that cannot be distinguished from any other missing value.

  This is one of the two fundamental integrity rules of the relational model, known as the Entity Integrity Rule (see "The Referential Integrity Rule" on page 5-37 for a definition of the other fundamental integrity rule).

  Although the database relational model does not explicitly state that alternate keys cannot be null, the constraint is implicit because a column set cannot be a potential primary key (a candidate key) if it is null.
- Primary key attributes cannot contain duplicate values.

  By definition, a primary key is a unique identifier. If it contains duplicate values, it cannot be unique (this means that multiset tables cannot have true primary keys).
- Primary key values should never be modified.

Guidelines for Selecting Primary Keys

This topic presents several recommendations for selecting the attributes that make up the primary key for a relation.

- Select numeric attributes as primary keys.

  Numeric keys are both easier to produce uniquely and easier to manage than character data.

  Always assign numeric attributes as the primary key if the key is to be system-generated.
- Whenever possible, use system-assigned keys to simplify the maintenance of uniqueness on the primary key attribute set. This is another factor that argues in favor of numeric keys.
- Select primary key attributes that can be unique for the life of a relation.
- Construct primary keys on attributes that rarely, if ever, change during the life of a relation.
- Never use intelligent keys.[4]

  An intelligent key carries semantics about the tuple it identifies.

  The expression is also used to describe a key constructed around one or more misprojected attributes that would not be in the relation were the database properly normalized.

---

4 Note that intelligent keys and natural keys are *not* the same thing. See "Intelligent key" on page 5-5 and "Natural key" on page 5-5 for details.

- Whenever relations are paired in a supertype-subtype relationship, always assign the same primary key to both.
- Use a consistent convention when naming primary key attributes (see "Guidelines for Naming Columns" on page 7-12).

  Because it is important to track the originating entity for attributes, always identify foreign keys with a unique code.

  For example, if you use the column naming convention described in "Guidelines for Naming Columns" on page 7-12, you might extend the convention as indicated in this example.

  Suppose the relation Lineitem has the following primary key composed of the three attributes indicated by preliminary attribute names.

Lineitem

LineitemNumber	OrderNumber	OrderYearDate
	PK	
	FK	FK

You might consider naming the primary key attributes as follows.

- LineitemNumber
- LineitemOrder_FK_Number
- LineitemOrder_FK_YearDate

The FK embedded in OrderNumber and OrderYearDate indicate that the attributes are foreign keys that reference the Order relation.

When attribute naming has been completed, the primary key attribute names should look like the following relation definition fragment.

Lineitem

LineitemNumber	Order_FK_Number	Order_FK_YearDate
	PK	
	FK	FK

Foreign Keys

Definition

A foreign key is an attribute set in one relation based on an identical attribute set that acts as the primary key for, or an alternate key in, a different relation.

Purpose

Foreign keys provide a mechanism for performing primary index joins, sometimes referred to as primekey joins.

Foreign keys are also used to maintain referential integrity among related tables in a database (see "The Referential Integrity Rule" on page 5-37).

Rule

Foreign key values are restricted to three types.

- A mirror image, drawn from the same domain, of a primary or alternate key in an associated relation.
- Wholly null
- Partially null

  The partially null case applies to compound foreign keys only, where one or more columns of the key might be null while others might contain references to primary keys in other tables, which are, by definition, non-null.

Though they are permitted by the relational model, you should avoid designing foreign keys that permit nulls because of the many potential problems nulls can cause in database management systems.

Chapter 13: "Designing for Missing Information," provides recommendations for dealing with nulls.

The Referential Integrity Rule

Definition: Referential Integrity Rule

The Referential Integrity Rule states the following:

If a base relation R has a foreign key FK that matches the primary key PK of base relation S, then every value of FK in R must have one of the following properties.

- Equal to the value PK of some tuple in S
- Wholly null (each value in the attribute set that defines FK is null)

  If your enterprise business rules prohibit nulls in foreign key columns, you can enforce a non-null constraint on a column by defining the column with a NOT NULL attribute using either CREATE TABLE or ALTER TABLE.

In other words, a row cannot exist in a table with a non-null value for a referencing column if no equal value exists in its referenced column.

This relationship is referred to as referential integrity and the act of ensuring that this rule is enforced is referred to as maintaining referential integrity.

Purpose of the Referential Integrity Rule

The intent of the Referential Integrity Rule is to ensure that if some tuple *r* in relation R references some tuple *s* in relation S, then tuple *s* must exist.

This prevents you from corrupting the integrity of the database by deleting data that is still being used by another relation.

Entity Integrity and Foreign Key Nulls

Permitting a foreign key to be wholly null seems to contradict the Entity Integrity Rule (see "Rules for Primary Keys" on page 5-34). This property is, to be sure, somewhat ad hoc, but it can have some practical usefulness. For example, consider the following case: an employee is working for some enterprise but has not yet been assigned to a department. Assuming that Department Number is a FK in the Employee relation, you can readily see that even when all the remaining information required to create an Employee relation tuple for the employee in question is available, we cannot create such a tuple in the Employee relation unless we also allow the FK to be null.

The tuple in question should be updated with the appropriate Department Number data just as soon as it is available, but allowing the null FK permits the enterprise to pay this employee even though she is not yet assigned to a department.

Alternatively, the problem can be avoided entirely with a minor change in the logical design of the database. See "Redesigning the Database to Eliminate the Need to Record Nulls" on page 13-12 for an example.

As another alternative, you can assign default values to the FK field set and then update with the appropriate Department Number as soon as you know what it is. SQL facilities for assigning default values to columns are described in the chapter "Default Value Control Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Enforcing the Referential Integrity Rule

The relational model states the Referential Integrity rule without providing a mechanism to enforce it. In practice, the rule is enforced through the PRIMARY KEY and FOREIGN KEY clauses in the CREATE TABLE statement. The keyword REFERENCES describes the table and column set in which the primary key image of the foreign key resides.[5]

When referential integrity is enforced, you cannot delete a row from the referenced (foreign) table as long as there is a row in the referencing table whose foreign key matches it.

Enforcing Referential Integrity in Teradata

To implement referential integrity (RI) in Teradata, you have three choices, ranked in order of preference:

1. Use the referential integrity constraint checks supplied and enforced by the database software.
2. Write your own, site-specific macros, triggers, or stored procedures to enforce RI.
3. Enforce constraints through application code.

See "Validating the Integrity of Tables In an Referential Constraint Relationship" and "Scenario for Data Corruption With Referential Constraints" in the description of the CREATE TABLE statement in *Teradata RDBMS SQL Reference, Volume 4*, for a description of the problems inherent in attempting to enforce database integrity using non-declarative constraints.

---

5 A REFERENCES WITH NO CHECK OPTION specification explicitly instructs Teradata *not* to enforce referential integrity on the specified relationship.

Definition: Referencing (Child) Table

The referencing table is referred to as the Child table, and the specified Child table columns are called referencing columns.

Referencing columns should be of the same number and have the same data type as the referenced table key.

Definition: Referenced (Parent) Table

A child must have a parent, and the referenced table in a PK-FK relationship is referred to as the Parent table. The key columns in the Parent table that see a Child table are called referenced columns.

Since the referenced columns are defined as unique constraints, the referenced columns must be one of the following:

- A unique primary index (UPI), not null
- A unique secondary index (USI), not null

Definition: Primary Key

With respect to RI, a primary key is a parent table column set that is referred to by a foreign key column set in a child table.

Definition: Foreign Key

With respect to RI, a foreign key is a child table column set that refers to a primary key column set in a parent table.

Why Referential Integrity Is Important

Referential integrity is a mechanism to keep you from corrupting your database. Suppose you have a table like the following:

OrderPart

Order Number	Part Number	Quantity
PK		NN
FK	FK	
1	1	110
1	2	275
2	1	152

Part number and order number, each a foreign key in this relation, also form the composite primary key.

Suppose you were to go the Part Number table and delete the row defined by the primary key value 1. The keys for the first and third rows in the OrderPart table are now parentless because there is no row in the Part Number table with a primary key of 1 to support them. Such a situation exhibits a loss of referential integrity.

Now, suppose you had a mechanism to prevent this from happening? If you attempt to delete the row with a primary key value of 1 from the Part Number table, the database management system does not allow you to do so. This is the way Teradata maintains referential integrity.

Besides data integrity and data consistency, referential integrity has these benefits.

Benefit	Description
Increases development productivity	It is not necessary to code SQL statements to enforce referential integrity constraints because Teradata automatically enforces Referential Integrity.
Requires fewer programs to be written	All update activities are programmed to ensure that referential integrity constraints are not violated because Teradata enforces Referential Integrity in all environments: no additional programs are required.

Referential Integrity Constraints

The combination of the foreign key, the parent key, and the relationship between them defined by the Referential Integrity Rule is called the referential integrity constraint.

This is a constraint defined on a table column (using the CREATE TABLE or ALTER TABLE SQL statements) that represents a referential integrity link between two tables.

A referential integrity constraint is always defined for a foreign key column set in the child table in a relationship.

See Chapter 12: “Designing for Database Integrity” for additional information about how referential integrity is essential to maintaining the integrity of databases.

Teradata provides two other features related to referential integrity constraints: batch referential integrity constraints and referential constraints. The basic differences among these different referential constraint types are summarized in the following table:

Referential Constraint Type	DDL Definition	Does It Enforce Referential Integrity?	Level of Referential Integrity Enforcement
Referential constraint	REFERENCES WITH NO CHECK OPTION	No	None
Batch referential integrity constraint	REFERENCES WITH CHECK OPTION	Yes	Implicit transaction
Referential integrity constraint	REFERENCES	Yes	Row

Referential constraints do not enforce the referential integrity of the database. Instead, they signal the Optimizer that certain referential relationships are in effect between tables, thus providing a means for producing better query plans without incurring the overhead of system enforcement of the specified RI constraints.

You should specify referential constraints only when you enforce the integrity of the referential relationship in some other way, or if the possibility of query errors and the potential for data corruption is not critical to your application. See “REFERENCES Constraints” on page 12-17 and “CREATE TABLE Column Definition Clause” in *Teradata RDBMS SQL Reference, Volume 4* for more information.

Batch referential integrity constraints are externally identical to regular referential integrity constraints. Because they enforce referential integrity in an all-or-none manner for an entire implicit transaction, they can be more performant than standard referential integrity constraints. In most circumstances, there is no difference between the two. Batch referential integrity becomes important when a single statement inserts many rows into a table, like the massive inserts that can occur with INSERT … SELECT statements.

Example: Referential Integrity Constraints

The following CREATE TABLE statement defines the following referential integrity constraints on table_A.

- A column-level constraint on the foreign key column, column_A1, and the parent table (table_B) key column, column_B1.
- A table-level constraint on the composite foreign key column set (column_A1, column_A2) and the parent table table_C.

```
CREATE TABLE table_A
(
 column_A1 CHAR(10) REFERENCES table_B (column_B1),
 column_A2 INTEGER
)
PRIMARY INDEX (column_A1)
FOREIGN KEY (column_A1, column_A2) REFERENCES table_C;
```

According to this definition, the single column primary key column_A1 must reference the primary key column column_B1 of the parent table table_B. Table_B must have a primary key, of which one column is column_B1, having the following definition.

```
CHARACTER(10) NOT NULL
```

The composite foreign key (column_A1, column_A2) must reference the primary key of the parent table table_C. Table_C must have a two-column primary key, the first column of which has the following definition.

```
CHARACTER(10) NOT NULL
```

The second column of the Table_C primary key must have the following definition.

```
INTEGER NOT NULL
```

Rules for Referential Integrity Constraints

Referential integrity constraints must obey the following set of rules:

- The parent key must exist when the referential integrity constraint is defined.
- The parent key columns must be either a unique primary index (UPI) or a unique secondary index (USI).
- The foreign and parent keys must have the same number of columns and their data types must match.
- The foreign and parent keys cannot exceed 16 columns.
- Duplicate referential integrity constraints are not allowed.

- You cannot drop or alter foreign key or parent key columns using an ALTER TABLE statement after a referential integrity constraint has been defined on them.

  To drop a foreign or parent key column after a referential integrity constraint has been defined on it, you must first drop the referential constraint and then alter the table to drop the foreign or parent key columns.
- The foreign key must be equal to the parent key or it must be null.
- Self-reference (a condition in which the Parent and Child tables are the same table) is allowed, but the foreign and parent keys cannot consist of identical columns.
- You can define no more than 64 referential integrity constraints per table.

Referential Integrity Constraint Checks

Teradata performs referential integrity constraint checks whenever any of the following things occur:

- A referential integrity constraint is added to a populated table.
- A row is inserted or deleted.
- A parent or foreign key is modified.

The following table summarizes these actions:

Action on Teradata	Constraint Check Performed
INSERT into parent table	None.
INSERT into child table	Must have matching parent key value if the foreign key is not null.
DELETE from parent table	Abort the request if the deleted parent key is referenced by any foreign key.
DELETE from child table	None.
UPDATE parent table	Abort the request if the parent key is referenced by any foreign key.
UPDATE child table	New value must match the parent key when the foreign key is updated.

Overhead Costs of Standard Referential Integrity

By implementing standard referential integrity, you incur certain overhead costs that can have a negative effect on performance. The following table lists the various referential integrity overhead operations that affect performance:

<table>
<tr><th>Operation</th><th>Description</th></tr>
<tr><td>Insert, update, or delete rows in a referencing table.</td><td>Overhead is similar to that for USI subtable row handling.
A redistributed spool for each reference is dispatched to the AMP containing the subtable entry.
BYNET traffic incurs the majority of the cost of this operation.</td></tr>
<tr><td>Insert row into a referencing table</td><td>A referential integrity check is made against the Reference Index subtable.
<table><tr><th>IF the referenced field is …</th><th>THEN the …</th></tr><tr><td>in the Reference Index subtable</td><td>count in the Reference Index subtable is incremented.</td></tr><tr><td>not in the Reference Index subtable</td><td>Reference Index subtable is checked to verify that the referenced field exists.
If it does, an entry with a count of 1 is added to the Reference Index subtable.</td></tr></table></td></tr>
<tr><td>Delete a row from a referencing table</td><td>A referential integrity check is made against the Reference Index subtable and its count for the referenced field is decremented.
When the count decrements to 0, then the subtable entry for the Referenced field is deleted.</td></tr>
<tr><td>Update a referenced field in a referencing table</td><td>Overhead is similar to that for changing the value of a USI column.
A referential integrity check is made against the Reference Index subtable. Both the inserting-a-row and deleting-a-row operations execute on the Reference Index subtable, decrementing the count of the old Referenced field value and incrementing the count of the new Reference field value.</td></tr>
<tr><td>Delete a row from a referenced table</td><td>The Reference Index subtable is checked to verify that the corresponding Referenced field does not exist. When nonexistence is confirmed, the row is deleted from the Referenced table.
No BYNET traffic is involved because the Referenced field is the same value in the Referenced table and the Reference Index subtable.</td></tr>
</table>

Domains and Referential Integrity

Introduction

This topic describes various domain rules for primary and foreign keys.

Definition: Domain

A domain is a data type. A data type is a well-defined set of values. For example, the data type INTEGER represents all the possible integer numbers.

Table Definition for Examples

An Employee table is used for the examples described in this topic. The definition of the Employee table is given by the following table.

Employee

EmployeeNumber	LastName	SupervisorEmployee Number
PK, SA		**FK**
1	Smith	null
2	Brown	1
3	White	1
4	Gibson	2
5	Black	2
6	Jones	3
7	Mason	3

The following domain rule applies to EmployeeNumber.

EmployeeNumber

Data Type	Domain Description
INTEGER	Value must be greater than zero.

Domain Rules for Primary Keys

The following rules apply to primary key domains and referential integrity.

- New primary key values must be drawn from the domain set of all valid values for the column on which the primary key value is defined.
- You cannot delete a primary key that references foreign keys because that violates the Referential Integrity Rule.

  The rules for deleting a primary key that has foreign key references are as follows.

Rule Name	Rule Description
Prevent	• Do not delete a PK if it references an existing FK. • Do not change the value of PK if it references an existing FK.
Reassign	• Change the value of FK to a different PK value before deleting the old PK row. • Change the value of FK to a different PK value before changing the old PK value.
Nullify	• Change the FK value to NULL before deleting the old PK row. • Change the FK value to NULL before changing the old PK value.
Cascade	• Delete the FK row before deleting the old PK row. • Delete the FK row before changing the old PK value.

Each of these rules preserves the referential integrity of a PK-FK relationship.

Questions and Answers for Primary Key Domains

The following questions and answers about primary keys indicate some examples of how domain integrity is enforced. The questions refer to the table defined in "Table Definition for Examples" on page 5-45.

Question	Answer	Explanation
Is it valid to add Employee 12345?	Yes	12345 is a positive integer.
Is it valid to delete the row for Employee 5?	Yes	Employee 5 is not referenced by any other row.
Is it valid to add Employee -23.67?	No	-23.67 is a negative decimal number.
Is it valid to delete the row for Employee 2?	Yes	Cascade rule.

Domain Rule for Foreign Keys

Foreign key values can be added only when they are drawn from the set of existing primary key values.

Questions and Answers for Foreign Key Domains

The following questions and answers about foreign keys indicate some examples of how domain integrity is enforced. The questions refer to the table defined in "Table Definition for Examples" on page 5-45.

Question	Answer	Explanation
Is it valid to have Employee 6 report to Employee 4?	Yes	Employee 4 is drawn from the set of existing employee numbers.
Is it valid to delete the row for Employee 7?	No	Employee 7 has an assigned foreign key value.
Is it valid to have Employee 6 report to Employee 8?	No	Employee 8 is not drawn from the set of existing employee numbers.

Denormalized Forms

Introduction

The remainder of this chapter describes several forms of denormalization, including the following types.

- Star- and snowflake schemas (dimensional modeling)
- Prejoins
- Derived columns (summary columns)
- Repeating groups

What Does Denormalization Mean?

The term denormalization is commonly used to describe any number of physical implementation techniques that enhance performance. The result of these operations is usually a violation of the design goal of making databases application-neutral. In other words, a "denormalized" database favors one or a few applications at the expense of all other possible applications.

Strictly speaking, these operations are not denormalization at all. The concept of normalization is logical, not physical. Logical denormalization should always be avoided at all cost. Develop a fully normalized design and then, if necessary, adjust your physical implementation to provide the desired performance enhancement. Finally, use views to hide the implementation tricks and present the database to users as if it is fully normalized.

Denormalization, Data Marts, and Data Warehouses

The following quote is taken from the web site of Bill Inmon (http://www.billinmon.com), the originator of the concept of data warehousing. It supports the position argued here: the more general the analyses undertaken on the warehouse data store, the more important the requirement that the data be normalized. The audience size issue he raises is a reflection of the diversity of analysis anticipated and the need to support any and all potential explorations of the data.

“The generic data model represents a logical structuring of data. Depending on whether the modeler is building the model for a data mart or a data warehouse the data modeler will wish to engage in some degree of denormalization. Denormalization of the logical data model serves the purpose of making the data more efficient to access. In the case of a data mart, a high degree of denormalization can be practiced. In the case of a data warehouse a low degree of denormalization is in order.

"The degree of denormalization that is applicable is a function of how many people are being served. The smaller the audience being served, the greater the degree of denormalization. The larger the audience being served, the lower the degree of denormalization."

Problems to Consider Regarding Denormalization

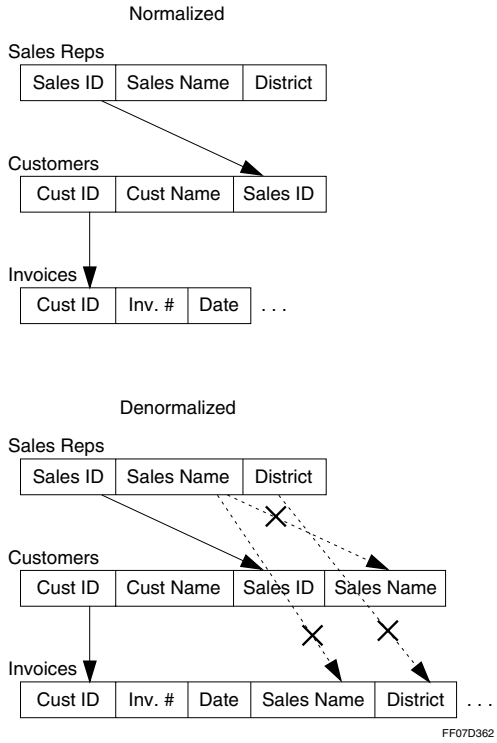
The effects of denormalization on database performance are unpredictable: as many applications can be affected negatively by denormalization as are optimized. If you decide to denormalize your database, make sure you always complete your normalized logical model first. Document the pure logical model and keep your documentation of the physical model current as well.

Denormalize the implementation of the logical model only after you have thoroughly analyzed the costs and benefits, and only after you have completed a normalized logical design.

Consider the following list of effects of denormalization before you decide to undertake design changes:

- Denormalization can increase, as well as hold down, hardware costs.
- While denormalization benefits the applications it is specifically designed to enhance, it often decreases the performance of other applications, thus contravening the goal of maintaining application neutrality for the database.
- A corollary to this observation is the fact that a denormalized database always makes it more difficult to implement new, performant applications.
- Because of the previous two effects, denormalization often increases the cost and complexity of programming.
- Denormalization always reduces the flexibility of the database. Remember, the goal of a Teradata data warehouse is to be able to answer any question on any data at any time.
- Denormalization introduces update anomalies to the database. Remember that the original impetus behind normalization theory was to eliminate update anomalies.

The following graphic uses a simple database to illustrate some common problems encountered with denormalization:



Consider the denormalized schema. Notice that the name of the salesman has been duplicated in the Customers and Invoices tables in addition to being in the Sales Reps table, which is its sole location in the normalized form of the database.

This particular denormalization has all of the following impacts.

- When a sales person is reassigned to a different customer, then all accounts represented by that individual must be updated, either individually or by reloading the table with the new sales person added to the accounts in place of the former representative.

  Because the Customers, or account, table is relatively small (fewer than a million rows), either method of updating it is a minor cost in most cases.
- At the same time, because the ID and name for the sales person also appear in every Invoice transaction for the account, each transaction in the database must also be updated with the information for the new sales person. This update would probably touch many millions of rows in the Invoice table, and even a reload of the table could easily take several days to complete. This is a very costly operation from any perspective.
- Denormalized rows are always wider rows. The greater the degree of a table, the larger the impact on storage space. This impact can be severe in many cases.

  Row width also affects the transfer rate for all I/O operations; not just for disk access, but also for transmission to the requesting client.

Evaluate all these factors carefully before you decide to denormalize large tables. Smaller tables can be denormalized with fewer penalties in those cases where the denormalization significantly improves the performance of frequently performed queries.

Commonly Performed Denormalizations

The following items are typical of the denormalizations that can sometimes be exploited to optimize performance.

- Repeating groups
- Prejoins
- Derived data (fields) and summary tables (column aggregations)

Alternatives to Denormalization

Teradata continues to introduce functions and facilities that permit you to achieve the performance benefits of denormalization without sullying your logical model. Among the available alternatives are the following.

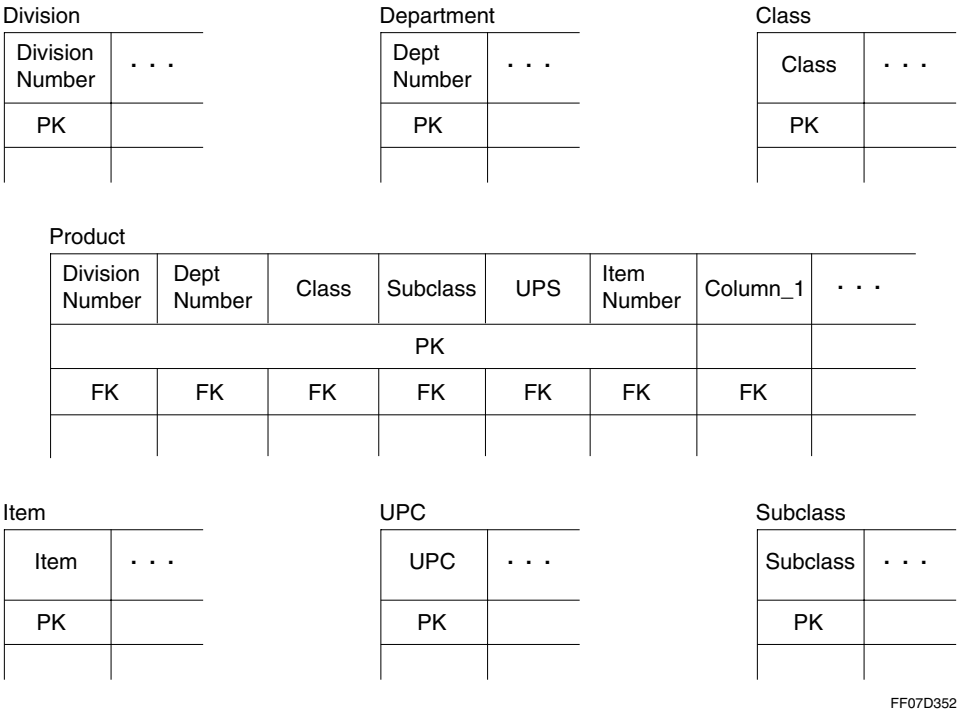
- Views
- Hash and join indexes
- Aggregate join indexes
- Global temporary tables

Dimensional Modeling, Star, and Snowflake Schemas

Definition: Dimensional Modeling

According to Ralph Kimball, the leading advocate of dimensional modeling in the data warehouse environment, "DM is a logical design technique that seeks to present the data in a standard, intuitive framework that allows for high-performance access. It is inherently dimensional, and it adheres to a discipline that uses the relational model with some important restrictions. Every dimensional model is composed of one table with a multipart key, called the fact table, and a set of smaller tables called dimension tables. Each dimension table has a single-part primary key that corresponds exactly to one of the components of the multipart key in the fact table" (Kimball, 1997).

The graphic indicates a simplified example of a fact table and its associated dimension tables.



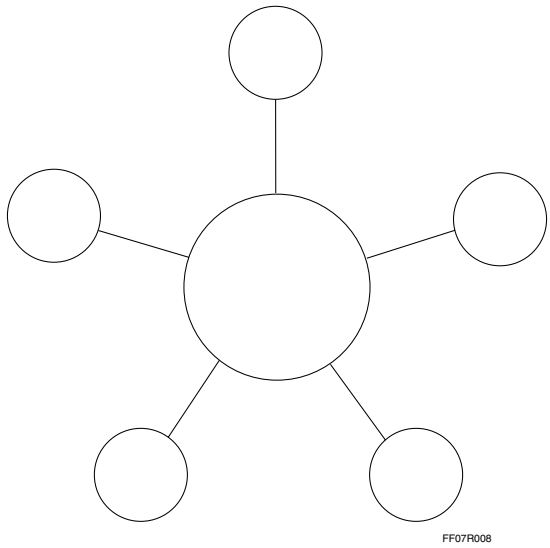
The morphology of a dimension model resembles that of a star or snowflake (see the graphics "Star Schema" on page 5-53 and "Snowflake Schema" on page 5-54).

In a dimensional model, fact relations always represent `M:M` relationships (see "Many-to-Many Relationships" on page 4-13). According to the model, a fact relation should contain one or more numerical measures (the "facts" of the fact relation) that occur for the combination of keys that define each tuple in the relation.

Dimension tables typically contain textual information that describes the fact attributes of the fact table.

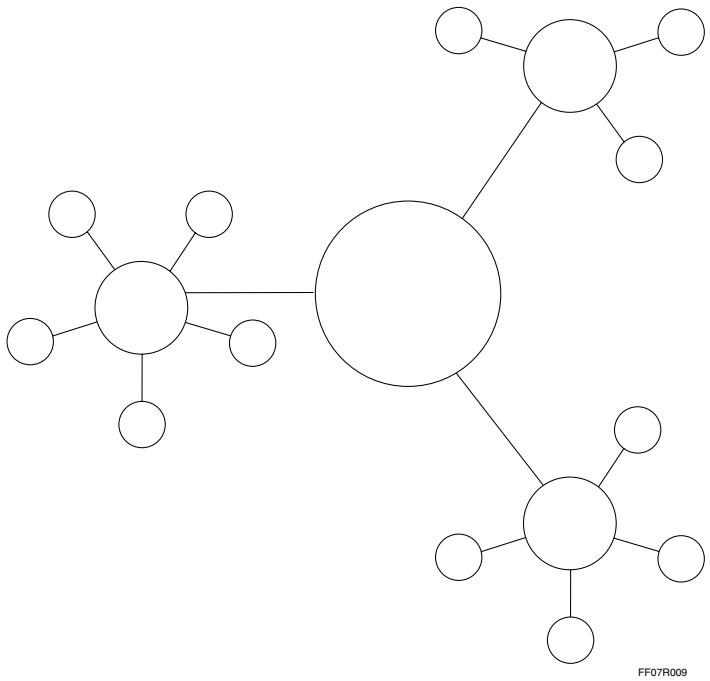
Star Schema

The following graphic illustrates the classical star schema.



Snowflake Schema

The following graphic illustrates the classical snowflake schema.



Contrasting the E-R Model With the DM Model

While a table in a fully normalized E-R-derived database represents an entity and all its relevant descriptors, thus representing *every possible business process in the enterprise* (more on this later), tables in a DM-derived database are said to represent dimensions of the business rules of the enterprise.

An E-R model for an enterprise is always more complex than a DM model for the same enterprise. While the E-R model might have hundreds of individual relations, the comparable DM model probably has somewhere between 10 to 25 superficially similar star join schemas, the dimension relations of which are largely shared among the various fact relations.

Strengths of Dimensional Modeling

Advocates of dimensional modeling list five advantages of DM over E-R models. Those advantages are provided in the following list.

- The dimensional model is a predictable, standard framework.
- The predictable framework of a star join schema is tolerant of ad hoc changes in user behavior.
- The dimensional model accommodates unanticipated new data elements and design decisions because it is readily extensible.
- Dimensional models are supported by a few simple, standard approaches to dealing with common modeling situations in the business universe.
- Aggregate processing is an increasingly important aspect of business data processing and dimensional models easily support such processing.

A Closer Look at Dimensional Modeling

You might be asking yourself the following question: "If dimensional modeling is such a terrific method, why does Teradata advocate the E-R model so strongly?" The reasons are numerous.

Consider the argument that the dimensional model is a predictable, standard framework. This contention is demonstrably false. When you normalize a logical schema, you proceed through a number of logically-derived rules and do not stop the process until you reach the highest possible normal form: BCNF for tables with simple keys and 5NF for tables with composite keys. Dimensional modeling proceeds in exactly the opposite direction and without any formal basis. At what point do you *stop* denormalizing your logical design? If you follow the inverted reasoning of this concept to its logical conclusion, do you stop the denormalization process only when you have reached the lowest possible normal form? If not, what *is* the goal, and what principles are used to derive it?

The answer to this question, of course, is that there are no principles for deriving a dimensional design other than improvisation. To understand the origins of dimensional modeling, it is important to consider the database processing environment that spawned the dimensional model. Quite simply, it was an environment designed to support OLTP processing, not decision support or tactical analysis. As a result, the query optimizers supporting these systems were designed with very specific goals in mind: specifically, to support ultrafast and extremely predictable updates of tables. The approach to decision support in such systems is to limit the questions that can be asked and to closely tailor the design for the questions you know will be asked: in other words, the goal is to develop application-specific databases. This is exactly what DM does, and it violates the core goal of logical modeling, which is to develop application-*neutral* databases.

Principal Motivations for Dimensional Modeling

The fundamental motivation behind the near-global advocacy of DM is to develop application-specific databases that minimize the performance problems presented by most vendor databases when forced to deal with fully normalized tables. While DM attempts to make a virtue of those deficiencies, as often as not it creates new, offsetting problems that, at best, offset any performance benefits achieved by introducing the many integrity and updatability problems introduced by such a severely denormalized logical model.

The most difficult tasks for any relational database management system to perform are the following:

- Joining tables
- Aggregating data
- Sorting data
- Scanning large volumes of data

Interestingly enough, these are also the four most requested operations in the decision support environment.

Teradata does not have the problems dealing with these issues that virtually every other relational system does. Because the majority of vendors cannot support these functions well, they advocate four workarounds to them.

In respective order, the workarounds are these:

- Denormalize the model.

  Translation: avoid having to join tables.

  Method: de-emphasize or ignore the functional and multivalued dependencies in the data and group keys and attributes arbitrarily.
- Use summary tables.

  Translation: avoid aggregation.

  Method: dispense with all detail data and store aggregates only, losing significant granularity.
- Store data in sorted order.

  Translation: avoid real time sorting.

  Method: store physical base table data in a particular sorted order that facilitates some applications, but worsens the response time of others.
- Rely heavily on indexes.

  Translation: avoid scanning large volumes of base table data.

  Method: index a significant number of columns in the database, forcing extensive index maintenance and administration.

The problem with these workarounds is not that they avoid facing the processing limitations of the products that require them; rather, it is that data stored in this way is highly biased toward processing predetermined queries,

and, as a result, is also highly biased against all other queries made against the data. The design is application- specific.

In other words, such a design achieves the polar opposite of the primary goal of building a data warehouse, which is to be able to ask any question at any time and receive a valuable answer in return. The goal of the workarounds is to maximize response time at the expense of utility and universality.

Keep in mind that the value of an answer is almost always inversely proportional to the time it takes to receive it: tough questions take time to answer. On the other hand, if you already know all the questions you are ever going to require of your database, then by all means use dimensional modeling.

Shortcomings of DM: Advantages of E-R

Dimensional models bias queries toward what is already known. They possess no potential for adapting to new relationships that might be mined from the database and, in fact, discourage exploratory data analysis.

To perform queries against a DM database, you must know all the dimensions of the facts you wish to analyze. The underlying database management system responds very suboptimally if all the required dimensions are not supplied in the query. Why is this? Because the database was designed not to support unexpected questions but only to support *expected* questions.

On the other hand, a fully normalized database permits deep, finely grained exploration of all possible relationships to be found in the data. When a database is designed to facilitate complex analyses of detail data, then responding to the sorts of questions for which DM databases are designed becomes trivial. This is the point at which the realization should hit you that the critique of normalized models raised by DM advocates - that they represent every possible business process in the enterprise - is an asset, not a liability.

William Inmon, the father of data warehousing, has gone as far as to describe the universal advocacy of dimensional modeling as selling users a “seven course meal of Hershey bars” …which is not to say that DM has no uses.

Where Dimensional Models Are Valuable

From the analysis of DM presented thus far, it should be fairly obvious where the strengths of dimensional modeling lie: in supporting rapid turnaround of canned, non-detail-oriented queries against the database. The principal environment for this sort of processing is the dependent data mart, particularly where OLAP analyses are a major component of the workload.

DM is an appropriate approach to data modeling when summarized data, which emphasizes the importance of response time over all else, is deemed more important than access flexibility and pinpoint precision detail.

The point to be made is that dimensional modeling should never be the first choice in building a relational database to support an enterprise data warehouse. The decision to use DM should always be a well-reasoned step that is taken only after other methods such as indexing and “denormalized” views on a normalized database have been ruled out and a cost analysis supports the decision to use DM.

A Compromise Between E-R and DM

One approach to integrating the entity-relationship and dimensional models that seems to offer a promising compromise between the two is the so-called dimensional view. “Dimensional Views” on page 5-62 describes the concept of the dimensional view in more detail.

Denormalizing With Repeating Groups

Introduction

Repeating groups are attributes of a non-1NF relation that would be converted to individual tuples in a normalized relation.

Example

For example, this relation has six attributes of sales amounts, one for each of the past six months.

Sales History

EmployeeNumber	Figures for the Last Six Months					
	Sales	Sales	Sales	Sales	Sales	Sales
PK						
FK						
UPI						
2518	32 389	21 405	18 200	27 590	29 785	35 710

When normalized, the Sales History relation has six tuples that correspond to the same six months of sales expressed by the denormalized relation.

Sales History

EmployeeNumber	SalesPeriod	SalesAmount
PK		
FK		
NUPI		
2518	20011031	32 389
2518	20011130	21 405
2518	20011231	18 200
2518	20010131	27 590
2518	20010228	29 785
2518	20010331	35 710

Reasons You Might Want to Denormalize With Repeating Groups

The following items are all possible reasons for denormalizing with repeating groups.

- Saves disk space
- Reduces query and load time
- Makes comparisons among values within the repeating group easier
- Many 3GLs and third party query tools work well with this structure

Reasons Not to Denormalize

The following items all mitigate the use of repeating groups.

- Makes it difficult to detect which month an attribute corresponds to
- Makes it impossible to compare periods other than months
- Changing the number of columns requires both DDL and application modifications

Denormalizing Through Views

Introduction

Views can be used to provide the appearance of denormalizing base relations without materializing the apparent denormalizations they simulate.

This can be a particularly useful solution to the dimensional modeling problem because it is possible to deliver both normalized and virtual multidimensional data to users through the use of dimensional views (see "Dimensional Views" on page 5-62).

Prejoin With Aggregation

The following example creates a prejoin view with aggregation. Note that you can create the identical object as a materialized join index.

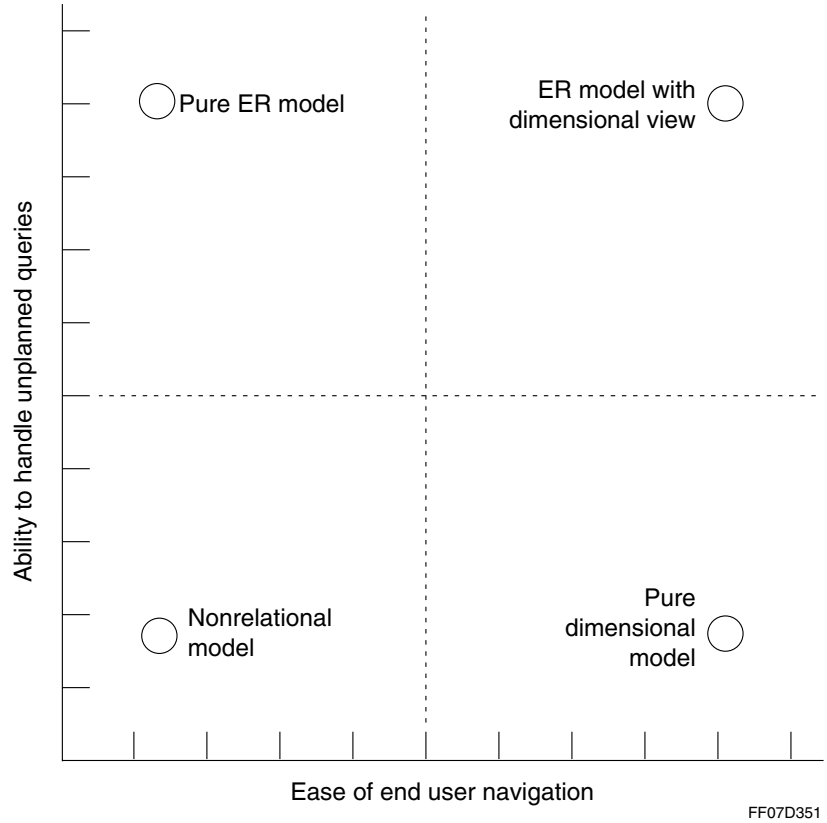
```
REPLACE VIEW LargeTableSpaceTotal
 (DBname,Acctname,Tabname,CurrentPermSum,PeakPermSum,
  NumVprocs)
AS SELECT DatabaseName,AccountName,TableName,
SUM (CurrentPerm)(FORMAT '---,---,---,--9'),
SUM (PeakPerm)(FORMAT '---,---,---,--9'),
COUNT(*)(FORMAT 'ZZ9')
FROM DBC.Tablesize
GROUP BY 1, 2, 3
HAVING SUM (currentperm) > 10E9;
```

```
SELECT DatabaseName (CHAR(10), TITLE 'DbName'),
 AccountName (CHAR(10),TITLE 'AcctName'),
 TableName (CHAR(16),TITLE 'TableName'), Vproc,
 CurrentPerm (FORMAT '---,---,---,--9'),
 CurrentPerm * 100.0 / CurrentpermSum (AS PctDist, TITLE '  %
//Distrib',FORMAT          'ZZ9.999'),PctDist * NumVprocs (AS
PctofAvg,TITLE '%         of//AVG ', FORMAT 'ZZ9.9')
FROM LargeTableSpaceTotal, DBC.Tablesize
WHERE DBname   = Tablesize.DatabaseName
AND   AcctName = Tablesize.AccountName
AND   TabName  = Tablesize.TableName
AND   PctofAvg > 125.0
ORDER BY 1, 2, 3, 4;
```

Dimensional Views

A dimensional view is a virtual star, or snowflake, schema layered over fully normalized physical detail data. Not only does such a view provide high performance, but it does so without incurring the update anomalies caused by denormalized data.

The following illustration graphs the capability of various data modeling approaches to solve ad hoc and data mining queries as a function of ease-of-navigation of the database.



The dimensional view leverages the strengths of both the E-R and dimensional models. Access to the data through standard applications, or by unsophisticated end users, can be accomplished by means of the dimensional view. More sophisticated applications, such as ad hoc tactical and strategic queries and data mining explorations can analyze the fully normalized data either directly or by means of normalized views.

Denormalizing Through Prejoins

Introduction

A prejoin moves frequently joined attributes to the same base relation in order to eliminate join processing. Some vendors refer to prejoins as materialized views.

Example

The following example first indicates two normalized relations, Job and Employee, and then shows how the attributes of the minor relation Job can be carried to the parent relation Employee in order to enhance join processing.

Job

Job Code	Job Description
PK	NN, ND
UPI	
1015	Programmer
1023	Analyst

Employee

Employee Number	Employee Name	Job Code
PK, SA		FK
UPI		
22416	Jones	1023
30547	Smith	1015

This is the denormalized, prejoin form of the same data. This relation violates 1NF.

Employee

Employee Number	Employee Name	Job Code	Job Description
PK, SA		FK	
UPI			
22416	Jones	1023	Analyst
30547	Smith	1015	Programmer

Reasons You Might Want to Denormalize Using Prejoins

The following items are all possible reasons for denormalizing with prejoins.

- Performance can be enhanced significantly.
- The method is a good way to handle situations where there are tables having fewer rows than there are AMPs in the configuration.
- The minor entity is retained in the prejoin so anomalies are avoided and data consistency is maintained.

Reasons Not to Denormalize Using Prejoins

The following items are all possible reasons for not denormalizing with prejoins.

You can achieve the same result without denormalizing your design by using any of the following methods.

- Views with joins (see "Denormalizing Through Views" on page 5-61)
- Join indexes (see "Denormalizing Through Join Indexes" on page 5-66)
- Global temporary tables (see "Denormalizing Through Global Temporary Tables" on page 5-69)

Denormalizing Through Join Indexes

Introduction

Join indexes provide the performance benefits of prejoin tables without incurring update anomalies and without denormalizing your logical database design.

Example

Consider the prejoin example in "Denormalizing Through Prejoins" on page 5-64. You can obtain the same performance benefits this denormalization offers without incurring any of its negative effects by creating a join index.

```
CREATE JOIN INDEX EmployeeJob
 AS SELECT (JobCode, JobDescription), (EmployeeNumber,
 EmployeeName)
FROM Job JOIN Employee ON JobCode ;
```

This join index not only eliminates the possibility for update anomalies, it also reduces storage by compressing redundant Job table information.

Reasons to Denormalize Using Join Indexes

The following items are all reasons to use join indexes to "denormalize" your database by optimizing join processing.

- Update anomalies are eliminated because the system handles all updates to the join index for you, ensuring the integrity of your database.
- Aggregates are also supported for join indexes and can be used to replace base summary tables.

For Further Information

See Chapter 11: "Join and Hash Indexes" for a full description of join indexes.

Derived Data

Introduction

Derived attributes are attributes that are not atomic. Their data can be derived from other, atomic attributes in the database. Because they are derived, they violate the laws of normalization.

Types of Derived Attributes

Derived attributes fall into two basic types.

- Summary (aggregate) data
- Data that can be directly derived from other attributes

Approaches to Handling Standalone Derived Data

Storing derived data violates the law of normalization related to attribute atomicity. There are occasions when you might want to denormalize standalone calculations for performance reasons. The decision to denormalize should be based on the following demographic information, all of which is derived through the ATM process described in Chapter 7: "The Activity Transaction Modeling Process."

- Number of tables and rows involved
- Access frequency
- Data volatility
- Data change schedule

Guidelines for Handling Standalone Derived Data

As a general rule, using an aggregate join index or a global temporary table is preferable to denormalization.

The following table provides guidelines on handling standalone derived data attributes by denormalization. The decisions are all based on the demographics of the particular data. When more than one recommended approach is given and one is preferable to the other, the entries are ranked in order of preference.

Access Frequency	Change Rating	Update Frequency	Recommended Approach
High	High	Dynamic	1 Use an aggregate join index or global temporary table. 2 Denormalize the physical implementation of the model.
High	High	Scheduled	Use an aggregate join index or global temporary table.
High	Low	Dynamic	Use an aggregate join index or global temporary table.
High	Low	Scheduled	• Use an aggregate join index or global temporary table. • Produce a batch report that calculates the aggregates whenever it is run.
Low	Unknown	Unknown	Calculate the information on demand rather than storing it in the database.

Any time the number of tables and rows involved is small, calculate the derived information on demand.

Reasons Not to Denormalize Using Derived Data

The following items deal with the issues of derived data without denormalizing base tables.

- Aggregate join index (see "Aggregate Join Indexes" on page 11-26)
- Global temporary table with derived column definitions
- View with derived column definitions

Denormalizing Through Global Temporary Tables

Introduction

Global temporary tables have a persistently stored definition just like any base table. The difference is that a global temporary table is materialized only when it is accessed by a DML statement.

Global temporary tables, like join indexes, are not part of the logical model. Because of this, they can be denormalized to any degree desired, enhancing the performance of targeted applications while not affecting the normalization of the underlying database schema. The logical model is not skewed, but all the benefits of denormalization are accrued.

Using Global Temporary Tables to Avoid Denormalization

You can use global temporary tables to avoid the following denormalizations you might otherwise consider.

- Prejoins
- Summary tables and other derived data

This final point is important as an alternative for applications that do not require persistent storage of summary results as offered, for example, by aggregate join indexes.

Using Global Temporary Tables to Enhance Performance

You can use global temporary tables to enhance performance in the following ways.

- Simplify application code
- Reduce spool usage
- Eliminate large numbers of joins

This final point is important as an alternative for applications that do not require persistent storage of prejoin results as offered, for example, by join indexes.

Example: Simple Denormalization for Batch Processing

The following temporary table serves 500 different transactions that create the output it defines. These transactions collectively run over one million times per year, but 95 percent of them run only on a monthly batch schedule.

With the following table definition stored in the dictionary, the table itself, which violates 2NF, is materialized only when one of those batch transactions accesses it.

TemporaryBatchOutput

Department Number	Employee Number	Department Name	Last Name	First Name
PK				
FK	FK			

Example: Aggregate Summary Table

The following table definition, if used infrequently, might be a good alternative to using an aggregate join index to define the equivalent summary table.

DepartmentAggregations

Department Number	Period	Sum Salary	Avg Salary	Employee Count
PK				
FK				
NUPI				

Example: Prejoin

Prejoins are a form of derived relationship among tables. The following table definition, if used infrequently, might be a good alternative to using a join index to define the equivalent prejoin table.

This particular table saves the cost of having to join the Order, Location, and Customer tables.

OrderCustomer

Order Number	Customer Number	Order Cost
PK	FK	
FK	NUPI	

Normalization References

Topic	Reference
Why Relational Theory is of Prime Importance to Real World Databases	Fabian Pascal, *Practical Issues in Database Management: A Reference for the Thinking Practitioner*, Boston, MA: Addison-Wesley, 2000. Passionately emphasizes the need to understand and apply the scientific principles of relational database theory to practical issues in database management. Equally importantly, Pascal highlights the departures of the SQL language from the relational model and demonstrates the negative ramifications of those departures, providing suggestions for optimal workarounds where possible and identifying examples of problems to avoid when a workaround is not possible.
Axiomatization for Functional and Multivalued Dependencies	Catriel Beeri, Ronald Fagin, and John H. Howard, “A Complete Axiomatization for Functional and Multivalued Dependencies in Database Relations,” *Proceedings of SIGMOD 1977*, 47-61, 1977. Provides a complete set of inference axioms for functional and multivalued dependencies.
• First Normal Form • Second Normal Form • Third Normal Form	E.F. Codd, "Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks," *IBM Research Report* RJ 599(# 12343), August 19, 1969. E.F. Codd, “A Relational Model of Data for Large Shared Data Banks,” *Communications of the ACM*, 13(6):377-387, 1970. These papers present the initial version of the relational model for database management. Among other things, both papers formally define the first three normal forms. Note that the definition of 3NF is incomplete.
• Third Normal Form • Boyce Codd Normal Form	E.F. Codd, “Recent Investigations into Relational Data Base Systems,” *Information Processing 74: Proceedings of the IFIP Congress*, pp. 1017-1021, 1974. Completes the definition for 3NF and names it “Boyce Codd Normal Form.” Carlo Zaniolo, "A New Normal Form for the Design of Relational Database Schemata," *ACM Transactions on Database Systems*, 7(3): 489-499, 1982. Formally demonstrates the relationship between 3NF and BCNF with an elegant proof that BCNF is strictly stronger than 3NF.

Topic	Reference
Fourth Normal Form	Ronald Fagin, “Multivalued Dependencies and a New Normal Form for Relational Databases,” *ACM Transactions on Database Systems*, 2(3):262-278, 1977. Formally defines 4NF.
Fifth Normal Form	Ronald Fagin, “Normal Forms and Relational Database Operators,” *Proceedings of SIGMOD 1979*, 153-160. Formally defines 5NF and names it “Projection/Normal Form,” or PJ/NF.
Conditional Formal Relationships Between 3NF and 5NF and Between BCNF and 4NF	C.J. Date and Ronald Fagin, "Simple Conditions for Guaranteeing Higher Normal Forms in Relational Databases," *ACM Transactions on Database Systems*, 17(3):465-476, 1992. Proves the two following relationships: • If a relational schema is in 3NF and *every* key is simple, then that schema is also in 5NF. • If a relational schema is in BCNF and at least one key is simple, then that schema is also in 4NF, but not necessarily in 5NF.
Overview of the Five Normal Forms	William Kent, “A Simple Guide to Five Normal Forms in Relational Database Theory,” *Communications of the ACM*, 26(2):120-125, 1983. An informal tutorial on the normal forms written by one of the primary early researchers in the field. This paper is the origin of the following definition for 3NF: “Every fact is a fact about the key, the whole key, and nothing but the key.”
Domain/Key Normal Form	Ronald Fagin, “A Normal Form for Relational Databases That is Based on Domains and Keys,” *ACM Transactions on Database Systems*, 6(3):387-415, 1981. Formally defines a sixth normal form and names it “Domain/Key Normal Form,” or DK/NF. Strictly a historical curiosity.
Inclusion Dependency Normal Form	Mark Levene and Millist W. Vincent, “Justification for Inclusion Dependency Normal Form,” *IEEE Transactions on Knowledge and Data Engineering*, 12(2):281-291, 2000. Formally defines IDNF and describes its usefulness.
Relational Model, Version 2	E.F. Codd, *The Relational Model for Database Management, Version 2*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1990. Extensively updates the original twelve rules supporting the relational model (RM/V2 defines over 200 rules) and rebuts all criticisms of the relational model in exhaustive detail.

Topic	Reference
Dimensional Modeling	Ralph Kimball, "A Dimensional Modeling Manifesto: Drawing the Line Between Dimensional Modeling and ER Modeling Techniques," *DBMS Magazine*, 10(9):59ff, 1997. A forceful and influential argument in favor of using DM exclusively to design databases to support the enterprise data warehouse. Ralph Kimball, Laura Reeves, Margy Ross, and Warren Thornthwaite, *The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing, and Deploying Data Warehouses*. New York: John Wiley and Sons, 1999. Advocates the use of dimensional modeling exclusively in designing relational databases to support enterprise data warehouses. Devotes all three of its chapters on data modeling to various aspects of DM without spending any significant time on normalization.

Section 2:

Translating the Logical Model Into a Physical Design

Chapter 6:

Modeling for Joins and Dependency Preservation

It is often true that logical relations can be decomposed into either BCNF or 4NF and still satisfy the lossless join property (see "Lossless Joins" on page 6-5). This chapter describes the lossless join and advocates dependency preservation (see "Dependency Preservation" on page 6-20) as a method for selecting the best set of relations for satisfying the lossless join property from competing BCNF and 4NF normalized relation sets.

The chapter describes how to design tables in ways that ensure lossless joins. By following the suggested guidelines, the likelihood of performing a lossy join is virtually nil.

The related topics of semantic disintegrity and dependency preservation are also described.

The set and dependency theory notations used in this chapter are defined in Appendix A: "Notation Conventions."

Relational Joins

Introduction

The relational join is an operation that combines data from two or more different relations. Setting aside the degenerate case of the Cartesian product join, you can join any two relations that share a common attribute. A row in relation R is joined to a row in relation S whenever the value of the common attribute is identical in both relations.

The expression binary join is often used to describe the process of joining two relations, while the expression *n*-ary join is used to describe joins of three or more tables. This is never more than a theoretical convenience because all relational systems perform binary joins no matter how many tables are joined in the process. The difference is that projection and restriction are done at each intermediate binary join stage until a single relation is returned in the end.

Law of Conservation of Relations

Notice that, like all other operations on relations, the result of a join is always another relation. This property of relations derives from what could be called the law of conservation of relations, or somewhat more formally, the closure property of relational systems: that the result of any operation on a relation is always another relation.

Example: Binary Single Column Join

Consider the following binary join of two abstractly attributed relations R_1 and R_2.

R_1

A	B	C
1	5	3
2	4	5
8	3	5
9	3	3
1	6	5
5	4	3
2	7	5

R_2

B	D	E
4	7	4
6	2	3
5	7	8
7	2	3
3	4	2

Relations R_1 and R_2 can be joined on column B. The following relation is the result.

(R_1 JOIN R_2)

A	B	C	D	E
1	5	3	7	8
2	4	5	7	4
8	3	5	4	2
9	3	3	4	2
1	6	5	2	3
5	4	3	7	4
2	7	5	2	3

Example: Binary Multicolumn Join

When relations share more than one attribute in common, then any join between them must be made on both attributes. Consider the following example.

R_1

A	B	C
6	1	4
8	1	4
5	1	2
2	7	1

R_2

B	C	D
1	4	9
1	4	2
1	2	1
7	1	2
7	1	3

The relations can be joined on columns B and C. The following relation is the result.

(R_1 JOIN R_2)

A	B	C	D
6	1	4	9
6	1	4	2
8	1	4	9
8	1	4	2
5	1	2	1
2	7	1	2
2	7	1	3

Lossless Joins

Introduction

The set theoretic basis for the normalization process results in any number of nice properties of relational databases. Among them is the characteristic that you can always recreate unnormalized tables that had been decomposed during the normalization process if the database has been completely and properly normalized. When such a join can be made, it is referred to as a lossless join.

By contrast, it is also possible to create relations that are not valid by means of the join operation. If a join can create a row that was not derived from an original unnormalized relation, then that join is not valid.

Definition: Lossless Join

Relations R_1 and R_2 are derived from projections of relation R such that $R_1 \cup R_2 = R$. Define any valid subset, or instance, of R as r.

The join of R_1 and R_2 is lossless iff the cardinality of their join is equal to the cardinality of r.

If the cardinality of the join of R_1 and R_2 is greater than the cardinality of r, then nonvalid rows were created by the join and it is lossy.

More formally, the join of R_1 and R_2 is lossless iff both of the following statements are true.

$R_1 \cap R_2 \neq \varnothing$

Either $(R_1 \cap R_2) \rightarrow ((R_1 - (R_1 \cap R_2))$

Or $(R_1 \cap R_2) \rightarrow ((R_2 - (R_1 \cap R_2))$

Or both

Example: Nonvalid Join

Consider the following unnormalized relation.

StudentName	Class	TrainerName	Time	RoomNumber	Grade
Chan	Physical Database Design	Klein	8:00	16-201	P
Iwahashi	Teradata Basics	Davis	8:00	14-205	P
Fields	Teradata Basics	Davis	8:00	14-205	P
Trott	Relational Database Modeling Workshop	Klein	9:00	20-110	P

The following set of functional dependencies is defined for this relation.

Time, Room Number → Class

Class, Student Name → Grade

Trainer Name, Time → Room Number

Class → Trainer Name

Time, Student Name → Room Number

Given this information, we can decompose the unnormalized relation into four individual relations, each achieving 4NF. These relations are the following.

Student-Time

StudentName	Time	Class
PK		

Student-Class

StudentName	Class	Grade
PK		

Trainer-Time

TrainerName	Time	RoomNumber
PK		

Class-Trainer

Class	TrainerName
PK	

An important observation to make is that not all the dependencies we have identified are accounted for by this decomposition. The reason for this is that the goal of the decomposition was to make lossless joins over these relations possible. This is an objective that can be met without converting each identified dependency into its own individual relation. Our desire is to make it possible to return correct answer sets to ad hoc queries made against these relations and nothing more. Many valid combinations and permutations of decompositions can satisfy the property of valid lossless joins.

Example: Lossy Join

The set of relations derived in the previous example can be populated as follows.

Student-Time

StudentName	Time	Class
Chan	8:00	Physical Database Design
Iwahashi	8:00	Teradata Basics
Fields	8:00	Teradata Basics
Trott	9:00	Relational Database Modeling Workshop

Student-Class

StudentName	Class	Grade
Chan	Physical Database Design	P
Iwahashi	Teradata Basics	P
Fields	Teradata Basics	P
Trott	Relational Database Modeling Workshop	P

Trainer-Time

TrainerName	Time	RoomNumber
Klein	8:00	16-201
Klein	9:00	20-110
Davis	8:00	14-205

Class-Trainer

Class	TrainerName
Physical Database Design	Klein
Teradata Basics	Davis
Relational Database Modeling Workshop	Klein

First Attempt at a Lossless Join

Suppose we need to produce a report that lists the students, trainers, classes, and the numbers of the rooms in which the classes are provided. The Student-Time and Trainer-Time relations contain the data we need, so we can try joining them over the attribute Time. The new relation formed by this join looks like this.

StudentName	Class	TrainerName	Time	RoomNumber
Chan	Physical Database Design	Klein	8:00	16-201
Chan	***Teradata Basics***	***Davis***	***8:00***	***14-205***
Iwahashi	***Physical Database Design***	***Klein***	***8:00***	***16-201***
Iwahashi	Teradata Basics	Davis	8:00	14-205
Fields	***Physical Database Design***	***Klein***	***8:00***	***16-201***
Fields	Teradata Basics	Davis	8:00	14-205
Trott	Relational Database Modeling Workshop	Klein	9:00	20-110

A cursory glance at this report reveals nonvalid information (notated in bold italic typeface). The indicated rows do not appear in our original unnormalized relation. As you can see, three of the four students are shown to have taken two classes simultaneously from two different trainers in two different rooms.

Second Attempt at a Lossless Join

We attempt to produce the same report, this time by joining the Class-Trainer and Trainer-Time relations over the attribute Trainer Name. The new relation formed by this join looks like this.

Class	TrainerName	Time	RoomNumber
Physical Database Design	Klein	8:00	16-201
Physical Database Design	***Klein***	***9:00***	***20-110***
Teradata Basics	Davis	8:00	14-205
Relational Database Modeling Workshop	***Klein***	***8:00***	***16-201***
Relational Database Modeling Workshop	Klein	9:00	20-110

Once again a join produces spurious rows. Klein teaches both Physical Database Design and Relational Database Modeling Workshop at the same time in the same room twice.

Third Attempt at a Lossless Join

We again attempt to produce the desired report, this time by making the following joins in the order indicated.

Step	Joined/Projected Relations	Join Column Set
1	Student-Time JOIN Class-Trainer	Class
2	Step 1 result relation JOIN Trainer-Date	• Trainer Name • Time
3	Step 2 result relation PROJECT Class and Room	None

The new relation formed by the first join looks like this.

StudentName	Class	TraineName	Time
Chan	Physical Database Design	Klein	8:00
Iwahashi	Teradata Basics	Davis	8:00
Fields	Teradata Basics	Davis	8:00
Trott	Relational Database Modeling Workshop	Klein	9:00

The next join, which produces the following, intermediate relation, looks like this.

StudentName	Class	TrainerName	Time	RoomNumber
Chan	Physical Database Design	Klein	8:00	16-201
Iwahashi	Teradata Basics	Davis	8:00	14-205
Fields	Teradata Basics	Davis	8:00	14-205
Trott	Relational Database Modeling Workshop	Klein	9:00	20-110

We project the Class and Room Number attributes from this relation to produce the report we originally set out to produce, which looks like this.

Class	Room Number
Physical Database Design	16-201
Teradata Basics	14-205
Relational Database Modeling Workshop	20-110

This result was only realized once it was recognized that the necessary sequence of joins was (Student-Time x Class-Trainer) x Trainer-Date.

Virtually any query can be handled correctly by a database in 4NF that also has the lossless join property. Correct responses to such ad hoc queries can often be obtained only after a careful analysis of the possible join options. Many sequences of joins can produce answers to such queries, but some sequences are preferable to others because they do not produce nonvalid data at any stage of the process. When an original unnormalized parent relation is correctly decomposed using both functional and multivalued dependencies, then there is always at least one sequence of joins that can recompose the original relation.

One More Example of Achieving a Valid Result From This Set of Relations

Suppose we decide that we need a report that lists student grades by room number. This is not a serious query for this particular set of relations because grades are given on a pass/fail basis only. Nevertheless, the process of attaining a correct response illustrates that more than one path can be taken to achieve the desired result when you have decomposed your relations carefully.

Either of the following join sequences produces a correct result without creating nonvalid data along the way.

Step	Join Sequence
1	Student-Date JOIN Class-Trainer
2	(Student-Date JOIN Class-Trainer) JOIN Student-Class
3	((Student-Date JOIN Class-Trainer) JOIN Student-Class) JOIN Trainer-Date

Step	Join Sequence
1	Student-Date JOIN Class-Trainer
2	(Student-Date JOIN Class-Trainer) JOIN Trainer-Date
3	((Student-Date JOIN Class-Trainer) JOIN Trainer-Date) JOIN Student-Class

The Lossless Join Relation

Introduction

A major problem with Entity-Attribute theory is its lack of a formal foundation. What one individual identifies as an entity another can identify as a relationship between entities. And sometimes even things that were identified as relationships in the initial stage of logical design must be materialized as relations once further normalization of the database has been performed.

The so-called lossless join relation is a case in point.

Definition

A lossless join relation is a table whose only reason for existence is to ensure that any ad hoc query against a set of relations can be made using only the standard operators of the relational algebra. Such a relation is not an identifiable entity. It often contains attributes that have no logical relationship. The purpose of this relation is to serve as a bridge between other relations in the database.

Example 1: Creating a Lossless Join Relation

Consider the relation R(A,B,C,D) having the following dependencies.

$A \rightarrow B$

$C \rightarrow D$

Using the standard procedures of taking projections to normalize the schema, you can produce the following two relations, both of which are in 4NF.

R_1	
A	**B**
PK	

R_2	
C	**D**
PK	

The lossless join property cannot be maintained with just these relations, however. To achieve a lossless join, you must also create the following lossless join relation.

R_3	
A	**C**
PK	

You can now join attributes B and D by joining R_1 and R_3, then joining the result with R_2. These joins do not create non-valid data, so R_3 is demonstrated to be the lossless join relation for the design of this database.

Because the lossless join relation is not created by combining attributes that modify an object or entity but by materializing a relationship between several relations, it can be difficult to populate and maintain because its attributes are often not codetermined.

Lossless join relations also tend to contain multivalued dependencies. Whenever this happens, you can decompose the relation into 4NF. This action not only removes the multivalued dependencies, but also creates smaller relations that are easier to populate and maintain than the original lossless join relation.

Definition: Lossless Join

Suppose you have relations R_1, R_2, and R such that the following property is true.

$$R_1 \cup R_2 = R$$

Define *r* to be any valid instance of R.

R_1 and R_2 have a lossless join if the number of rows in the join of their projections equals the number of rows in *r*. This is obvious because the binary join of instances of R1 and R2 builds all the rows in R. That is $R_1 \cup R_2 = R$, as is already known. If no spurious rows are created by this join operation, then the join of R_1 and R_2 is lossless.

You can sometimes determine whether the join of relations in a decomposition of a relation is lossless by the functional dependencies in the parent relation. For example, consider the following instance of relation R with functional dependency BC → D.

R

A	B	C	D
1	5	8	9
6	5	8	9
2	3	7	4

Suppose this instance of R is decomposed into the projections R_1(A,B,C) and R_2(B,C,D).

R_1

A	B	C
1	5	8
6	5	8
2	3	7

R_2

B	C	D
5	8	9
3	7	4

You can join these projections to produce the original relation R without loss. All valid instances of R can be projected and joined without loss because any row pair identical in BC is also identical in D, which proves the original assertion that the functional dependency BC → D ensures that R1 and R2 can be joined without loss.

More formally, (R1 JOIN R2) is lossless

IF

$R_1 \cap R_2 \neq \varnothing$

ANDR1∩R2→(R_1 - $R_1 \cap R_2$)

ORR1∩R2→(R_2 - $R_1 \cap R_2$)

ELSE IF

$R_1 \cap R_2 \neq \varnothing$

AND R1∩R2→(R_1 - $R_1 \cap R_2$)

AND R1∩R2→(R_2 - $R_1 \cap R_2$)

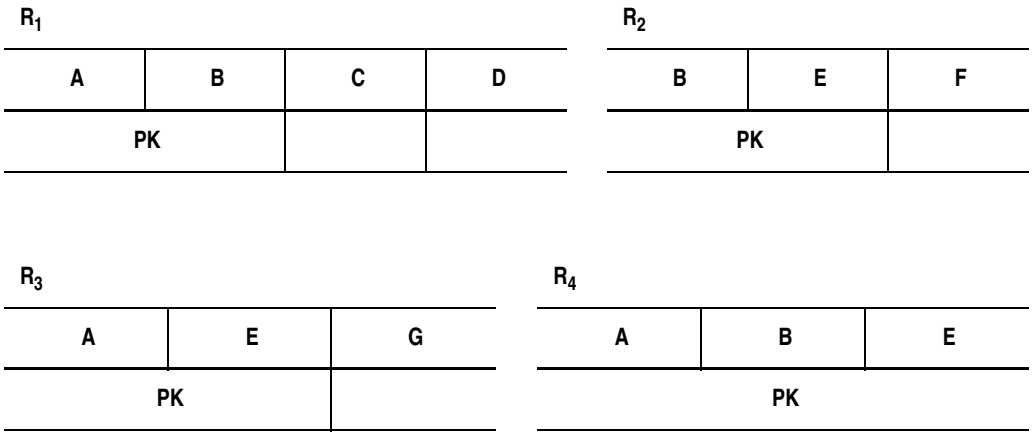
As an illustration of this, consider the following list of functional dependencies and the set of relations generated from them.

AB → C

AB → D

BE → F

AE → G



Relations R_1, R_2, and R_3 satisfy the functional dependencies from the bulleted list, while relation R_4 is a lossless join relation used to join relations R_1, R_2, and R_3 correctly.

Example 2: Lossy Join

Suppose you want to join attributes C, E, and F. The attributes are carried in R_1 and R_2. When this join is made, the following equalities are true.

$R_1 \cap R_2 = B$

$R_1 - R_1 \cap R_2 = ACD$

$R_2 - R_1 \cap R_2 = EF$

If either or both of the following dependencies is true, then the join is lossless.

$B \rightarrow ACD$

$B \rightarrow EF$

Because neither dependency is valid, (R_1 JOIN R_2) might not be lossless.

Example 3: Lossless Join

Suppose you join the lossless join R_4 with R_1. When this join is made, the following equalities are true.

$R_1 \cap R_4 = AB$

$R_1 - R_1 \cap R_4 = CD$

$R_4 - R_1 \cap R_4 = E$

If either or both of the following dependencies is true, then the join is lossless.

$AB \rightarrow CD$

$AB \rightarrow E$

Visual inspection of R_1 shows that AB → CD is true because AB is the primary key for the relation and C and D are its attributes, so the join is lossless.

Example 4: Lossless Join

(R_1 JOIN R_4) creates a new relation R_5. Suppose you join R_2 with R_5. When this join is made, the following equalities are true.

$R_2 \cap R_5 = BE$

$R_5 - R_2 \cap R_5 = ACD$

$R_2 - R_2 \cap R_5 = F$

If either or both of the following dependencies is true, then the join is lossless.

BE → ACD

BE → F

Visual inspection of R_2 shows that BE → F is true because BE is the primary key for the relation and F is its attribute, so the join is lossless.

This two join sequence produces relation R_6 with attributes A, B, C, D, E, and F, which carries the information required (attributes C, E, and F) while not containing any nonvalid information.

The conditions illustrated in these examples indicate that proof of nonloss joinability is sufficient, but not necessary, to ensure a nonloss join. In fact, it is possible to make nonloss joins even when the conditions examined here are not met.

Multivalued Dependencies and Lossless Joins

Suppose you have relation R(A,B,C,D) with the multivalued dependency BC ⇒ D. An instance of this relation could look like the following.

R

A	B	C	D
1	1	1	1
1	1	1	2
2	1	1	1
2	1	1	2

Take projections on R to normalize it. This decomposition results in relations R_1(A,B,C) and R_2(B,C,D), which look like this.

R_1

A	B	C
1	1	1
2	1	1

R_2

B	C	D
1	1	1
1	1	2

If you join R_1 and R_2, you recover the original relation R without introducing spurious rows.

However, because neither of the following dependencies is true, the sufficiency condition explained at the end of "Definition: Lossless Join" on page 6-5 is not satisfied.

$BC \rightarrow D$

$BC \rightarrow A$

Nonetheless, the join is nonloss because a multivalued dependency is both a necessary and a sufficient condition for two relations to be joined without loss. In this example, the multivalued dependency $BC \Rightarrow D$ is the minimal condition necessary to ensure that (R_1 JOIN R_2) is nonloss for any join of projections of R into R_1 and R_2.

More formally, if either of the following multivalued dependencies is true, then the join is lossless.

$R_1 \cap R_2 \Rightarrow (R_1 - R_1 \cap R_2)$

$R_1 \cap R_2 \Rightarrow (R_2 - R_1 \cap R_2)$

Additionally, if (R_1 JOIN R_2) is non-lossy, then at least one of the listed multivalued dependencies must be true. Furthermore, if (R_1 JOIN R_2) *is* lossy, then *neither* of the multivalued dependencies is true.

Design Decision

Nonkey attributes of a lossless join relation can contain multivalued dependencies. If you identify any such dependencies, you should consider decomposing the relation into a set of 4NF relations.

Semantic Disintegrity

Definition

The expression semantic disintegrity describes the problem of lossy joins. The study of semantic disintegrity approaches lossy joins from a different perspective than that pursued in "Lossless Joins" on page 6-5 in that it examines the sequence of mappings from one attribute to another in the course of join processing.

Properties of Attribute Mapping

The following properties and rules apply to the analysis of semantic disintegrity by means of attribute mapping.

- Mappings are either one-to-one or one-to-many.
- A one-to-one mapping from attribute X to attribute Y is equivalent to the functional dependency $X \rightarrow Y$.
- The expression one-to-one is defined much more loosely for attribute mapping than is generally seen in mathematical analysis. For example, The same value of Y can map into *different* values of X in a one-to-one mapping.
- If a mapping is not one-to-one, then it is one-to-many.
- To avoid semantic disintegrity, a sequence of attribute mappings must meet the lossless join property.

Example

Suppose we have the following set of relations and the stated functional dependencies.

R

W	X

S

X	Y

T

Y	Z

$W \overline{\rightarrow} X$

$X \rightarrow Y$

$Y \rightarrow Z$

We can use these dependencies to determine the attribute mappings.

Dependency	Mapping
$W \overline{\rightarrow} X$	One-to-many
$X \rightarrow Y$	One-to-one
$Y \rightarrow Z$	One-to-one

With respect to join orders in order to recompose the parent relation, the mappings and structures of the relations R, S, and T imply the following join order.

```
((R JOIN S ON X) JOIN T ON Y)
```

R JOIN S ON X is valid if either of the following relationships is true.

$X \rightarrow W$

$X \rightarrow Y$

Because the join satisfies the functional dependency $X \rightarrow Y$, it is valid.

The joined relation RS must then be joined with relation T on attribute Y. For this join to be valid, either or both of the following dependencies must be met.

$Y \rightarrow WX$

$Y \rightarrow Z$

Because the join satisfies the functional dependency $Y \rightarrow Z$, it is valid.

The entire join is now proven to be valid, so it does not cause semantic disintegrity.

Examples of Avoiding Semantic Disintegrity

The following table provides a nonexhaustive list of examples that indicate attribute mappings for which semantic disintegrity is never a problem.

Situation	Example
All mappings are one-to-one.	$W \rightarrow X$ $X \rightarrow Y$ $Y \rightarrow Z$
The only one-to-many mapping is the first mapping in the sequence.	$W \overline{\rightarrow} X$ AND $X \rightarrow Y$ AND $Y \rightarrow Z$
The only one-to-many mapping is the second mapping in the sequence, but $X \rightarrow W$.	$W \rightarrow X$ AND $X \overline{\rightarrow} Y$ AND $Y \rightarrow Z$
The only one-to-many mapping is the second mapping in the sequence, but $X \overline{\rightarrow} Y/W$ or $X \overline{\rightarrow} W/Y$.	$W \rightarrow X$ AND $X \overline{\rightarrow} Y$ AND $Y \rightarrow Z$
There are multiple one-to-many mappings. Requires analysis of the rules for lossless joins as they apply to the individual cases. Three representative cases are presented here.	
Mappings	$W \rightarrow X$ AND $X \overline{\rightarrow} Y$ AND $Y \overline{\rightarrow} Z$
Lossless join criteria	$X \Rightarrow W/Y$ AND $Y \Rightarrow WX/Z$ or $X \Rightarrow W/Y$ AND $Y \Rightarrow Z/WX$
Mappings	$W \overline{\rightarrow} X$ AND $X \rightarrow Y$ AND $Y \overline{\rightarrow} Z$
Lossless join criteria	$Y \Rightarrow WX/Z$ or $Y \Rightarrow Z/WX$
Mappings	$W \overline{\rightarrow} X$ AND $X \overline{\rightarrow} Y$ AND $Y \overline{\rightarrow} Z$
Lossless join criteria	$X \Rightarrow W/Y$ AND $Y \Rightarrow WX/Z$ or $X \Rightarrow Y/W$ AND $Y \Rightarrow WX/Z$ or $X \Rightarrow W/Y$ AND $Y \Rightarrow Z/WX$ or $X \Rightarrow Y/W$ AND $Y \Rightarrow Z/WX$

Dependency Preservation

Definition

A decomposition of a relation has dependency preservation if all the functional dependencies in the original relational schema can be implied from the functional dependencies in the decomposed relations.

Why This Is an Important Property to Maintain

You should aim to preserve dependencies because this property reduces many update problems. Dependency preservation does *not* guarantee the elimination of update problems, however, as illustrated by the following simple example.

We have the following relational schema and the stated dependencies.

R

X	Y	Z

$X \rightarrow Y$

$X \rightarrow Z$

$Y \rightarrow Z$

We decompose this relation into the two following normalized relations that preserve all the dependencies in the original relation.

S

X	Y

T

Y	Z

In spite of that preservation, this decomposition does not permit direct updates of XZ combinations.

Example 1: Dependency Preservation Maintained

Consider the relational schema R and its functional dependencies.

R

X	Y	Z

$X \rightarrow Y$

$X \rightarrow Z$

$Y \rightarrow Z$

This schema can be decomposed to the following two relations, each of which is in 3NF.

S

X	Y
PK	

T

Y	Z
PK	

If the dependencies X → Y, X → Z, and Y → Z can be implied from the functional dependencies in S and T, then the decomposition has dependency preservation. The following functional dependencies are easily derived from relations S and T, therefore the decomposition has dependency preservation.

Dependency	How Derived
X → Y	Inspection of relation S.
Y → Z	Inspection of relation T.
X → Z	Transitivity axiom (see "Inference Axioms for Functional Dependencies" on page 5-9).

Example 2: Dependency Preservation Not Maintained

Consider the same relational schema and functional dependencies given for "Example 1: Dependency Preservation Maintained" on page 6-20.

This time we decompose the original schema into two different relations, also in 3NF.

S

X	Y
PK	

T

X	Z
PK	

Once again, if the dependencies X → Y, X → Z, and Y → Z can be implied from the functional dependencies in S and T, then the decomposition has dependency preservation. Analysis of functional dependencies results in the following derivations.

Dependency	How Derived
X → Y	Inspection of relation S.
$Y \overline{\rightarrow} Z$	This dependency cannot be inferred either from relation S or from relation T using the Transitivity axiom (see "Inference Axioms for Functional Dependencies" on page 5-9).
X → Z	Inspection of relation T.

Because we cannot imply the required dependency Y → Z from this decomposition, it does not maintain dependency preservation.

Dependency Preservation Rules for Decomposition to 3NF or BCNF

The following decomposition rules apply to maintaining dependency preservation and lossless joins.

- Relations can always be decomposed into a set of 3NF relations that maintain both the lossless join property and dependency preservation.
- Relations *cannot* always be decomposed into a set of BCNF relations that maintain both the lossless join property and dependency preservation.

Example 3: Lost Dependency Preservation by BCNF Decomposition

Consider the following 3NF relation with the noted functional dependencies.

R

X	Y	Z
PK		

- XY → Z
- Z→ A

This relation cannot be decomposed to BCNF without losing dependency preservation. Consider the following decomposition.

S

X	Z
PK	

T

Y	Z
PK	

The dependency XY → Z cannot be derived from these relations; therefore, dependency preservation is lost.

Suppose you have the following unnormalized Employee- Education relation with the stated functional dependencies.

Employee-Education

Employee Number	Highest Degree	School Code	School Name	Major
PK				

Employee Number, Highest Degree, Degree → School Code

School Code → School Name

You decide to normalize this relation by decomposing it into a set of 3NF relations. The following two 3NF relations are the result of that decomposition. Do they preserve the necessary dependencies?

Employee-Education-School Code

Employee Number	Highest Degree	Major	School Code
PK			

Employee-Education-School Name

Employee Number	Highest Degree	Major	School Name
PK			

The answer is no, they do not, and the dependency they do not satisfy is School Code → School Name. In order to add a new school code-school name pair to the database, at least one employee must have graduated from the school.

A second decomposition of the Employee-Education relation yields the following set of 3NF relations that satisfy all the necessary dependencies.

Employee-Education-School Code

Employee Number	Highest Degree	Major	School Code
PK			

Schools

School Code	School Name
PK	

Chapter 7:

The Activity Transaction Modeling Process

Introduction

This chapter first describes and defines some of the concepts underlying the Activities and Transaction Modeling (ATM) process and then describes the ATM process and its forms, providing examples of how to fill them out and apply them to the physical database model.

Purpose of the ATM Process

You use the early ATM process forms as input to more complex forms later in the process. You use the final ATM process forms when you create the physical objects in your database, ensuring that the universe of domains, constraints, table accesses, and reports and queries are understood well in advance of the process of creating the physical database objects that derive from those constructs.

The ATM process is the first step in the transition from your logical data model to the implementation of your physical data.

The second and final step in that transition is to collect or prototype data demographics.

Goals of the ATM Process

The goals of the ATM process are as follows.

1. Define all domains and constraints.
2. Identify all applications.
3. Model application processing activities including their transactions and run frequencies.

4 Model each transaction using the following information.

   1 Identify tables used.

   2 Identify columns required for value and join access.

   3 Estimate qualifying cardinalities.

5 Summarize value and join access information across all transactions.

6 Add data demographics to the Table Access Summary by Columns report.
   - Table cardinalities
   - Column value distributions (histograms)
   - Column change ratings

The following table indicates the activities of this process and the forms required to complete each activity.

Step	Action	Form Used
1	Define all domains in the system.	Domains
2	Define all constraints for the system.	Constraints
3	Identify all applications in the system.	System
4	Model each identified application.	Application
5	Model each identified transaction.	Report/Query Analysis
6	Summarize all value and join accesses.	
7	Transfer access information.	Table
8	Compile or estimate data demographic information.	
9	Identify column change ratings.	

Terminology

The terms in the following list are used throughout the activities modeling process.

Term	Definition
Domain	A well-defined, closed set of values from which column data can be drawn. Domain is really just another term for data type. Note that this is the classic definition of a domain and differs from the definition used by the ANSI SQL standard. Note that Teradata does not support the *physical* concept of domains.
Constraint	A well-defined physical restriction that can be defined for a column or table. Constraints include the following: • UNIQUE • PRIMARY KEY • FOREIGN KEY • CHECK *expression* • REFERENCES The following rules and recommended practices apply to constraints. • *Always* name constraints. • Constraint names must be 30 or fewer characters. • A constraint name must be unique among all other constraint names defined for a table. • You can specify constraints both at the column and table levels. • The system does not assign names to constraints you do not name. For detailed information about the various table constraints, see "CREATE TABLE" in the chapter “SQL Data Definition Language Statement Syntax” in *Teradata RDBMS SQL Reference, Volume 4*. For design-related information about column and table constraints, see Chapter 12: “Designing for Database Integrity.”

<table>
<tr><th>Term</th><th>Definition</th></tr>
<tr><td>Table</td><td>A two dimensional representation of an entity constructed from the following components:
• Rows, representing tuples
• Columns, representing attributes
Tables are sometimes referred to as relations, though the correspondence between relations and tables is not always direct.
The following example shows the structure of the Location entity before it is fully attributed.
<table><tr><th>LocationNum</th><th>CustNum</th><th>State</th><th>Country</th></tr><tr><th>PK, SA</th><th>FK, NN</th><th>FK</th><th>FK, NN</th></tr><tr><td>2</td><td>10</td><td>CA</td><td>USA</td></tr><tr><td>7</td><td>0</td><td>CA</td><td>USA</td></tr><tr><td>1</td><td>2</td><td>NY</td><td>USA</td></tr><tr><td>4</td><td>7</td><td>PR</td><td>USA</td></tr></table></td></tr>
<tr><td>Row</td><td>An instance of an object in a relational table. Rows in relational tables are not ordered. The expression is a synonym for record, a term not used for relational database systems.
Rows are sometimes referred to as tuples, though the formal term for the corresponding concept is n-tuple.
The number of rows in a table is referred to as its cardinality.
The following example indicates a randomly selected row from the Location entity.
<table><tr><th>LocationNum</th><th>CustNum</th><th>State</th><th>Country</th></tr><tr><th>PK, SA</th><th>FK, NN</th><th>FK</th><th>FK, NN</th></tr><tr><td>2</td><td>10</td><td>CA</td><td>USA</td></tr><tr><td>7</td><td>0</td><td>CA</td><td>USA</td></tr><tr><td>1</td><td>2</td><td>NY</td><td>USA</td></tr><tr><td>4</td><td>7</td><td>PR</td><td>USA</td></tr></table></td></tr>
</table>

<table>
<tr><th>Term</th><th>Definition</th></tr>
<tr><td>Column</td><td>A unique, atomic attribute of a relational entity. Columns in relational tables are not ordered logically, though they are ordered physically and can be referred to by their column number.

Columns are sometimes referred to as attributes.

The number of columns in a table is referred to as its degree or arity.

The following example indicates the CustNum column from the Location entity.

<table>
<tr><th>LocationNum</th><th>CustNum</th><th>State</th><th>Country</th></tr>
<tr><th>PK, SA</th><th>FK, NN</th><th>FK</th><th>FK, NN</th></tr>
<tr><td>2</td><td>10</td><td>CA</td><td>USA</td></tr>
<tr><td>7</td><td>0</td><td>CA</td><td>USA</td></tr>
<tr><td>1</td><td>2</td><td>NY</td><td>USA</td></tr>
<tr><td>4</td><td>7</td><td>PR</td><td>USA</td></tr>
</table></td></tr>
<tr><td>Primary Key</td><td>The primary key is a column set that uniquely identifies a tuple within a relation. Every relation must have one and only one primary key defined during logical design, but a primary key need not be formally defined for a table corresponding to a logically defined relation.

A table can have multiple candidate primary keys, but only one defined primary key. A candidate primary key not selected as the primary key for a table is referred to as an alternate key.

A primary key cannot be null and must be unique.

The following example indicates the primary key column for the Location entity.

<table>
<tr><th>LocationNum</th><th>CustNum</th><th>State</th><th>Country</th></tr>
<tr><th>PK, SA</th><th>FK, NN</th><th>FK</th><th>FK, NN</th></tr>
<tr><td>2</td><td>10</td><td>CA</td><td>USA</td></tr>
<tr><td>7</td><td>0</td><td>CA</td><td>USA</td></tr>
<tr><td>1</td><td>2</td><td>NY</td><td>USA</td></tr>
<tr><td>4</td><td>7</td><td>PR</td><td>USA</td></tr>
</table></td></tr>
</table>

<table>
<tr><th>Term</th><th>Definition</th></tr>
<tr><td>Identity Column</td><td>A column for which the values are unique and system-generated. Values for identity columns can be generated by the system in all cases or only in those cases for which users do not provide a value.

Identity columns are frequently used to generate surrogate keys.

See “Identity Columns” on page 14-13 and the detailed description of identity columns and their use documented by "CREATE TABLE" in the chapter "SQL Data Definition Language Statement Syntax" in Teradata RDBMS SQL Reference, Volume 4 for more information about identity columns.</td></tr>
<tr><td>Foreign Key</td><td>The foreign key is a column set that identifies a relationship between the table for which it is a foreign key and one or more other tables in the database.

Foreign keys are used both as join conditions and to maintain referential integrity between tables.

A foreign key must be the primary key for the table it references and it can be null unless you define the foreign key column set for the table to exclude nulls.

The following example indicates the three foreign key for the Location entity. Notice that the CustNum column, which is constrained to be not null, relates Location to Customer, the State column relates Location to State, and the Country column, which is also constrained to be not null, relates Location table to Country.

<table>
<tr><th>LocationNum</th><th>CustNum</th><th>State</th><th>Country</th></tr>
<tr><td>PK, SA</td><td>FK, NN</td><td>FK</td><td>FK, NN</td></tr>
<tr><td>2</td><td>10</td><td>CA</td><td>USA</td></tr>
<tr><td>7</td><td>0</td><td>CA</td><td>USA</td></tr>
<tr><td>1</td><td>2</td><td>NY</td><td>USA</td></tr>
<tr><td>4</td><td>7</td><td>PR</td><td>USA</td></tr>
</table>
</td></tr>
<tr><td>Normalization</td><td>A method for segregating the attributes of a database into individual tables in such a way that those attributes uniquely modify (or depend upon) the primary key for that table.

Somewhat more formally, a relation is said to be fully normalized if all its nonkey attributes are functionally dependent on its primary key.</td></tr>
</table>

Domains

Introduction

The concept of domain as used in relational database theory derives directly from the more formal definition given in relational set theory and function theory. A variable is defined as a set of points having both a domain and a range, where the domain defines the specific type of data represented by the set and the range defines the bounds on that type. A more concise way of expressing this is to say that domains are data types. It is frequently a good idea to extend this definition to include constraints defined for a column. Extended domains of this type are frequently referred to as business rules. A simple example would be to define EmployeeNumber as an INTEGER type and extend the definition with a constraint to disallow negative integer numbers as valid employee numbers.

Domains have been a fundamental concept supporting the theory of relational databases since the inception of that theory. The variables described by a domain in a relational database are the valid values an attribute, or column, can have.

Unfortunately, the ANSI SQL standard does not support a rigorous atomic definition of domain, so it is up to the database designer to define the domains for a database and their ranges rigorously by creating various constraints on table columns and by ensuring that applications do not make nonsensical cross-domain data manipulations such as subtracting an integer part number from an integer inventory count (see "Column Comparisons" on page 7-9).

Properties of Domains in a Teradata Database

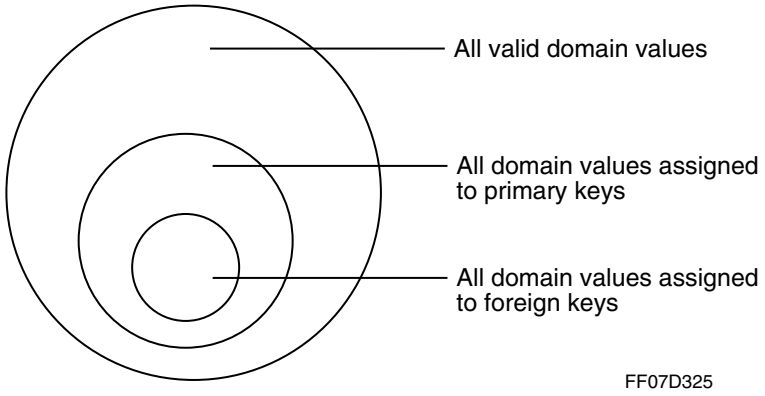
Domains in a Teradata database are defined to have the following properties.

- A domain defines the set of possible values that can be written to a table column.
- The values of a domain cannot be decomposed to component values; they are atomic.

  Note that this is not true for domains having INTERVAL or TIMESTAMP data types, which are not atomic, but the concept generally holds true.
- A domain must have a name.
- A domain must have an assigned data type.

Domains and Keys

The following Venn diagram indicates the relationship among primary and foreign keys drawn from the same domain.



The following rules apply to these relationships.

- All primary key values for a table are drawn from the domain that defines all possible values for that column.
- All foreign key values that reference a primary key value in another table are drawn from the same domain as the values for that primary key.

Teradata Data Types

The following table lists the valid Teradata data types you can use to define a domain.

Data Type	Definition
BYTE VARBYTE	A binary integer used to store digital images.
CHARACTER VARCHAR LONG VARCHAR GRAPHIC	Any glyph from a supported language. For the English language, this type is often referred to as alphanumeric.
DATE	A valid 24-hour named epoch from the Gregorian calendar.
DECIMAL	Any base-10 real number, including those with a fractional part.
FLOAT REAL DOUBLE PRECISION	Any base-10 rational number expressed in exponential format.
INTEGER BYTEINT SMALLINT	Any natural number.

Data Type	Definition
INTERVAL	A time duration with optional fractional precision. Interval data is not implemented atomically, though it is generally treated logically as if it were atomic.
TIME	A valid time expressed using 24-hour notation with optional fractional precision.
TIMESTAMP	A valid date and time expressed using 24-hour notation with optional fractional precision.

Domains in the Logical Data Model

You should always define and assign domains to your table attributes during the logical design phase of your database design.

Domains can be bounded specifically, nonspecifically, or both simultaneously. For example, you can define the employee number domain to be both not null (a nonspecific boundary) and constrained within a restricted range such as greater than 100 and less than 1 000 000.

Column Comparisons

The necessity for naming your domains becomes clear when you examine the topic of column comparisons.

According to the relational model, columns can be compared if and only if their values are drawn from the same domain. If an application is to compare data from different columns in any way, all of the following statements must be true for all of the columns being compared.

- They must share the same set of values.
- The same value represents the same real world object in all cases.
- The values can be compared, added, subtracted, and joined.

In terms of domains, these rules can be stated as follows.

- Columns from the same domain always have the same defined domain name.
- Columns from the same domain always have the same defined constraints.
- Columns from the same domain always have the same data type.

Commercial relational database management systems relax these comparison restrictions to a greater or lesser degree. For example, you can compare INTEGER and DECIMAL values in commercially available systems because, the reasoning goes, both are numeric types. The more strict domain comparison rules of the relational model do not permit this comparison.

Column Names and Constraints

Introduction

The topic of domains leads naturally to the topic of naming columns and assigning constraints to them.

Column Names Are Important

Always use a consistent method to define the names of the columns in your databases.

The following set of columns illustrates an unsophisticated example of why it is important to name your columns carefully.

Case Study, Part 1

Suppose you have defined the following two entities.

PrimaryKeyColumn	ColumnA	Date	Date	Date
PK, SA				
		03 Apr 1960	17 Apr 1981	29 Nov 1987
		31 Jan 1937	03 Apr 1960	30 Jun 1981
		05 Mar 1930	21 Aug 1950	03 Apr 1960

PrimaryKeyColumn	Date	Date	Date
PK, UA			
	03 Apr 1960	10 Apr 1960	13 Apr 1960
	29 Mar 1960	03 Apr 1960	06 Apr 1960
	24 Mar 1960	31 Mar 1960	03 Apr 1960

Can you tell if the highlighted dates in these tables all refer to the same point in time? Can you tell if *any* of them do? If not, then you should not compare them. But how do you know whether the dates are drawn from the same domain or not?

Case Study, Part 2

Now assume that you have identified and named all the domains in your database. The next step is to apply those domains to the identified entities in your logical model.

Consider the same two entities we examined before, only now their columns have been assigned names that represent the domains from which those column values are drawn.

Employee

EmployeeNumber	LastName	BirthDate	HireDate	TerminationDate
PK, SA				
		03 Apr 1960	17 Apr 1981	29 Nov 1987
		31 Jan 1937	03 Apr 1960	30 Jun 1981
		05 Mar 1930	21 Aug 1950	03 Apr 1960

Order

OrderNumber	OrderDate	ShippingDate	BillingDate
PK, UA			
	03 Apr 1960	10 Apr 1960	13 Apr 1960
	29 Mar 1960	03 Apr 1960	06 Apr 1960
	24 Mar 1960	31 Mar 1960	03 Apr 1960

Guidelines for Naming Columns

Introduction

As far as it has been defined so far, this column naming convention fails to define the Date domain with enough rigor to prevent an application developer from comparing a shipping date with a birth or hire date, but it provides a sufficient example for illustration as well as a first approximation to how you might define a domain rigorously.

Because ANSI SQL does not have facilities for defining domains with this level of rigor, your only course of action is to do so textually. You can record these distinctions in several ways.

- DBC.TVFields.CommentString

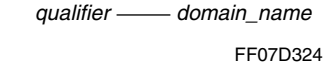
  Create this comment string using the COMMENT SQL statement.

  View the comment you create using the Columns system view.
- Column naming conventions
- Metadata repositories
- ATM Domains form

After you have rigorously defined your domains textually, your next step should be to name your columns by qualifying those domain names.

Column Naming Convention

Teradata recommends the following syntax for naming columns.



FF07D324

where:

Syntax element …	Specifies …
qualifier	a unique name to differentiate one use of a domain from all the other uses of that domain.
domain_name	the name of the domain being qualified.

Using this convention, you could define date domains such as the following example names.

- HRDate
- HR_Date
- OrderDate
- Order_Date

The more finely defined domain names naturally lead to tables having columns with names like those in the following tables.

Employee

EmployeeNumber	LastName	BirthHRDate	HireHRDate	TerminationHRDate
PK, SA				
		03 Apr 1960	17 Apr 1981	29 Nov 1987
		31 Jan 1937	03 Apr 1960	30 Jun 1981
		05 Mar 1930	21 Aug 1950	03 Apr 1960

Order

OrderNumber	OrderOrdDate	ShippingOrdDate	BillingOrdDate
PK, UA			
	03 Apr 1960	10 Apr 1960	13 Apr 1960
	29 Mar 1960	03 Apr 1960	06 Apr 1960
	24 Mar 1960	31 Mar 1960	03 Apr 1960

With this convention, an application programmer would know never to compare an employee birth date with an order shipping date because they do not represent the same thing even though they share the same set of valid values.

Note, too, that superficially "identical" values are not necessarily drawn from the same domain. The following dollar amounts, for example, do not refer to the same real world objects.

- $1.00 US
- $1.00 Canadian
- $1.00 Australian

In the physical design phase, you can further limit column values with constraints imposed on them using the CREATE TABLE or ALTER TABLE statements.

For example, the HRDate domain would probably exclude all weekend and holiday dates, and if your company only ships goods on Fridays, then you would place a Fridays-only constraint on the ShippingOrdDate column.

Naming Foreign Key Columns

You should also name foreign key columns to match their names in their primary tables. For example, the foreign key columns in the following relation are not only indicated by the name of their primary table (Order) but are also demarcated with the characters FK to redundantly indicate their origin in another table.

Lineitem

Lineitem _Number	Order_FK_Number	Order_FK_YearDate
PK		
	FK	FK

Metadata Definitions

All domain definitions should be stored in a metadata database and those definitions should be made available to all users of the database.

ATM Domains Form

As part of the transition from the logical model of your database to its physical implementation, you should record all domains that you identify on a Domains form. See "Domains Form" on page 7-17.

Key Values and Relationships Among Tables

Introduction

One result of database normalization is the identification of relationships among the tables defined in the database. These relationships are defined on the basis of primary key values shared between tables.

Example

Consider the following basic case. The Employee table was defined as a major entity during the logical design phase. Its columns are defined as follows.

Employee

EmpNum	EmpLastName	EmpFirstName	EmpMiddleInitial	EmpPhoneNum
PK				
…	…	…	…	…

The EmployeePhone table was defined as a minor entity related to the Employee table. Its columns are defined as follows.

EmployeePhone

EmpNum_FK_Employee	EmpPhoneNum	EmpPhoneComment
PK, FK	PK	
…	…	…

Note that the EmployeePhone table has a foreign key. That foreign key, EmpNum, is the primary key of the Employee table. The two tables relate to one another through this primary key relationship.

In this particular relationship, the minor entity, EmployeePhone, is said to be the Child table because it references another table. That referenced table, Employee, is said to be the Parent table in the relationship.

As a sidebar, note that because the referenced column is a primary key, it is by definition uniquely constrained, so it must be defined physically as either of the following two index types.

- Unique primary index, not null
- Unique secondary index, not null

Certain database semantics derive from such relationships, and those semantics require a particular type of database constraint to maintain their integrity.

This constraint is referred to as a referential constraint and it is said to maintain referential integrity.

See "The Referential Integrity Rule" on page 5-37 and "Domains and Referential Integrity" on page 5-45 for more information about referential integrity.

Domains Form

Purpose

Lists and describes all the domains for a database.

This form supports the first step in the ATM process (see the table on page 7-3).

Information Recorded

The Domains form records the following information about each domain.

Information Recorded	Description
ELDM Page	Page of the Extended Logical Data Model to which this domain pertains.
System	Name of the system on which this domain is defined.
Domain Name	Name of the domain.
Domain Description	Full text description of the domain.
Data Type	Encoded data type for the domain.
Max Bytes	Maximum number of bytes a column value drawn from this domain can occupy in disk storage. Note that the number of bytes per character does not necessarily map 1:1 to the number of characters defined for the column by the defined data type. Multibyte character sets are often represented by multiple bytes per single character, and you must account for this information when filling out the Domains form.
Print Format	Representation of the FORMAT clause (if any) used in the definition of columns drawn from this domain.
Constraint #	Constraint number (as recorded on the Constraints form) that applies to this domain (if any).

Data Type Codes

The following table lists the valid data types and their codes.

Data Type	Code
BYTE	B(n)
BYTEINT	I1
CHARACTER	C(n)
CHARACTER VARYING	CV(n) Note: Same as VC(n)
DATE	D
DECIMAL	DEC (n[,m])
DOUBLE PRECISION	DP
FLOAT	F(n)
GRAPHIC	G(n)
INTEGER	I
INTERVAL YEAR	IY
INTERVAL YEAR TO MONTH	IYM
INTERVAL MONTH	IMO
INTERVAL DAY	ID
INTERVAL DAY TO HOUR	IDH
INTERVAL DAY TO MINUTE	IDM
INTERVAL DAY TO SECOND	IDS
INTERVAL HOUR	IH
INTERVAL HOUR TO MINUTE	IHM
INTERVAL HOUR TO SECOND	IHS
INTERVAL MINUTE	IMI
INTERVAL MINUTE TO SECOND	IMS
INTERVAL SECOND	IS
LONG CHARACTER VARYING	LVC
LONG GRAPHIC VARYING	LVG
NUMERIC	NUM (n[,m])
REAL	R
SMALLINT	I2
TIME	T
TIMESTAMP	TS
TIME WITH TIME ZONE	TTZ

Data Type	Code
TIMESTAMP WITH TIME ZONE	TSTZ
VARBYTE	VB(n)
VARCHAR	VC(n) Note: Same as CV(n)
VARGRAPHIC	VG(n)

Example

Note that this example assumes you are using a single byte character set to define the Addr, City, and Comment domains.

Domains Page: of:				ELDM Page: System:	
Domain Name	**Domain Description**	**Data Type**	**Max Bytes**	**Print Format**	**Constr #**
Addr	Address	CV(30)	30		
Amt	Dollar Amount	D(10,2)	8	$$$$$$$9.99	
City	City Name	CV(30)	30		
Comment	Comment	CV(100)	100		
…	…	…	…	…	…

Constraints Form

Purpose

The Constraints form has two purposes.

- List all constraints, triggers, and validity checks defined for a database.
- Provide input to the Table form.

This form supports the second step in the ATM process (see the table on page 7-3).

About Database Constraints

Database constraints and validity checks are sometimes loosely referred to as business rules (see “Business Rules” on page 12-11, “Integrity Constraint Types” on page 12-12, and “Constraint Specifications” on page 12-15). They define conditions that must be met before a given value is permitted to be written to a column such as value ranges, equality or inequality conditions, and intercolumn dependencies. Teradata supports constraints at both the column and table levels. Constraints are defined using the CREATE TABLE and ALTER TABLE SQL statements.

Database triggers, too, are often referred to as business rules. They define certain actions that are to be taken when a particular condition occurs during the update of the table on which they are defined.

Information Recorded

The Constraints form records the following information about each constraint, trigger, and validity check.

Information Recorded	Description
ELDM Page	Page of the Extended Logical Data Model to which this constraint pertains.
System	Name of the business system in which the applications using this constraint are defined.
Constraint Number	The number that applies to this constraint. This number is used by other forms as a “foreign key” value to refer to the constraints defined by this form.
Constraint Description	Full text description of the constraint.

Constraint Codes

The following table lists some commonly used constraint codes and their meanings.

Code	Definition
FK	Foreign key
ID	Identity column
NC	No changes permitted
ND	No duplicates permitted
NN	Not null
PK	Primary key
SA	System-assigned
UA	User-assigned

Example

Constraints Page: of:	ELDM Page: System:
Constraint Number	**Constraint Description**
001	ND, NN, NC, PK
002	Must be greater than zero (whole positive integer)
003	Mutually exclusive: One required
004	Mutually exclusive: All required
005	No recursion and no loops
006	Excludes Saturdays, Sundays, and legal holidays
007	Prevent delete rule
008	Reassign delete rule
009	Nullify delete rule
010	Cascade delete rule
011	Copy rows to History table before deleting
012	Trigger update to OrderPart Category table on insert to Order or Part tables.
…	…

System Form

Purpose

The System form has two purposes.

- Identify all applications, grouping them by their business system.
- Provide input to the Application and Table forms.

This form supports the third step in the ATM process (see the table on page 7-3).

About Applications

A system is a set of application programs, or applications, grouped by business function. Applications are grouped into systems because each cluster of applications typically serves a different user community within the enterprise.

In terms of ATM, an application is a major business function. Each application is implemented by a set of transactions (see "Application Form" on page 7-24).

Ensuring that applications are clustered into the appropriate systems is the most important activity undertaken during this stage of the ATM process. The information described by the System form is generic and applies irrespective of the eventual physical implementation of the database.

Information Recorded

Information Recorded	Description
ELDM Page	Page of the Extended Logical Data Model to which this system pertains.
System Name	Name of the business system in which the applications using this constraint are defined.
System Description	Brief description of the function of the system.
Application ID	Unique identifier for the application being documented.
Application Name	Full text name of the application being documented.
User Contacts	Names and other useful information pertaining to principal users of the application.

Example

System **ELDM Page: 1** **System:** PTS
Page: of

System Name: Part Tracking System (PTS)

System Description: Tracks current status and history of each part over its lifetime.

Application ID	Application Name	User Contacts
Cust	Customer tracking	Joy Calder X829
PCat	Part categories	Bill Alpert
Rpt	Parts reporting	Hall Werts
…	…	…
…	…	…
…	…	…

Application Form

Purpose

The Application form has two purposes.

- Identify the transactions for each application in the system to estimate run frequencies and volumes.
- Provide input to the physical design of the database.

This form supports the fourth step in the ATM process (see the table on page 7-3).

About Transactions

The term *transaction* is used very loosely in the context of the Application form. Any of the following activities equates to a transaction.

- Report
- Single-row SELECT query
- Batch load job

Estimating Run Frequency and Duration

A typical business year has the following characteristics.

Attribute	Count
Number of business days per week	5
Number of hours per business day	8
Number of business weeks	52
Number of business days per year	260
Number of business hours per year	2 080

For each transaction, estimate the typical and peak run frequencies and identify when the action is performed and the duration of its performance.

The task is to estimate five parameters for each identified transaction. The parameters to be estimated are the following.

- Normal (nonpeak period) run frequency
- Peak period run frequency
- Point in the business cycle when the peak period occurs
- Duration of the peak period

Example

The following example form models some of the transactions for the Order Tracking application.

Application **ELDM Page:** 1 **Page:** of **System:** PTS

Application ID: Ord **Application Name:** Order Entry

Application Description: Generates and tracks part orders

		Run Frequencies				
Transaction ID	**Transaction Name**	**Typical**	**Peak**	**When**	**Length**	**Total**
OrdCl	Close an order	50/day	75/day	EOQ	2 weeks	14 800
			100/day	EOY	4 weeks	
OrdCn	Cancel an order	1/quarter	N/A	N/A	N/A	4
OrdD	Delete closed orders	1/month	N/A	First day of month	N/A	12
…	…	…	…	…	…	…

Report/Query Analysis Form

Purpose

The Report/Query Analysis form has two purposes.

- Capture a simple category of data demographics for each individual transaction within an application within a system.
  - Lists the tables accessed during a transaction.
  - Lists columns used for valued access (if any).
  - Lists columns used for join access (if any).
- Provide input to the Table form.

This form supports the fifth and sixth steps in the ATM process (see the table on page 7-3).

Terminology

The following terms apply to identifying Report/Query Analysis form components.

Term	Definition
Access column	A column name that would be specified in the WHERE clause of a query. In the following WHERE clause, *column_name* is the name of the access column. `WHERE column_name = 'xyx'`
Join column	A column name that would be specified in the ON clause of a join query. In the following ON clause, *table_1.column_name* and *table_2.column_name* are the names of join columns. `ON table_1.column_name = table_2.column_name`

Procedure

Perform the following procedure to complete the Report/Query Analysis form for each transaction.

Step	Action
1	Begin the analysis with standard reports, load requirements, and similar data.
2	Identify the tables involved and enter the names of each on a Report/Query Analysis form.
3	Identify which columns are printed and enter the names of each on the same Report/Query Analysis form that documents their tables.
4	Identify whether rows are accessed by specific column values. If so, enter the names of each column for which access values must be obtained and mark them with a V. Note that these must specify equijoin conditions matching on a single value.
5	Identify whether tables must be joined. If so, enter the names of the join columns and mark them with a J. Note that these must specify equijoin conditions.
6	Identify the number of rows that qualify for processing. This is not the number of rows in the answer set, but the number of rows that must be processed in order to obtain the answer set. Use these guidelines to determine what values to use. **1** Identify the number of rows, if any, that qualify for value access. **2** Identify the number of rows, if any, that qualify for join access. **3** Select the smaller of **1** and **2** and use it as the value for the number of rows processed. **4** If there are no value access rows and no join access rows, then use the number of rows in the table.

Examples

Note on the hospital database: There are over 1 000 patients in the database. Each patient is hospitalized long-term (≥ one year).

Example 1

The Patient Detail by Room Assignment report derives from the Patient and Bed tables. Values and selected demographics for those tables are modeled as follows.

Patient **Cardinality: 1 000**

Patient_ID	Bed_Number	Nurse_ID	Doctor_ID
PK	FK, ND	FK	FK
701	A	201	501
702	B	201	502
703	C	202	502
…	…	…	…

Bed **Cardinality: 1 500**

Bed_Number	Room_Number
PK, ND	FK
A	101
B	102
C	103
…	…

Patient Detail by Room Assignment

Room_ Number	Patient_ID	Bed_Number	Nurse_ID
101	701	A	201
	716	F	233
	723	T	205
102	702	B	201
	725	Q	201
	748	L	222
…	…	…	…

Frequency or Importance Ranking: _____

Report/Query Description: Patient detail by room assignment

Table: Patient **Number of Output Rows:** 1 000

Input Value Or Source	Bed Bed Number	
Access Column(s)	Bed Number	

Table: Bed **Number of Output Rows:** 1 500

Input Value Or Source	Patient Bed Number	
Access Column(s)	Bed Number	

Table: **Number of Output Rows:**

Input Value Or Source		
Access Column(s)		

Table: **Number of Output Rows:**

Input Value Or Source		
Access Column(s)		

Example 2

The Billing Report for Patient 705 is more complex than the report analyzed in "Example 1" on page 7-28. The cardinality of the Patient Service History table is based on the assumption that there are one year of service data is kept in the history file, each patient receives an average of two services per day, and there are roughly 1 000 patients in the hospital at any time.

The calculation is as follows.

```
1000 patients * 365 days * 2 services = 730 000 rows.
```

Service **Cardinality: 125**

Service_ID	Description	Cost
PK		
801	Appendectomy	$ 500.00
802	Tonsillectomy	$ 350.00
803	Anesthesia	$1 000.00
…	…	…

Patient_Service_History **Cardinality: 730 000**

Patient_ID	Service_ID	Timestamp
	PK	
FK	FK	
701	801	1992-08-05 11:39
701	803	1992-08-05 23:59
705	801	1992-08-05 8:23
705	802	1992-08-05 13:43
705	803	1992-08-05 18:32
705	814	1992-08-05 23:56
…	…	…

Billing Report for Patient 705, August 2000

Date	Service_ID	Description	Cost
08/05/1992	801	Appendectomy	$ 500.00
	802	Tonsillectomy	$ 350.00
	803	Anesthesia	$1 000.00
	Total for 08/05/1992		$1 850.00
08/28/1992	814	Face lift	$3 200.00
	Total for 08/28/1992		$3 200.00

Example 3

This report is based on the same tables as that of "Example 2" on page 7-29. It reports summary information from those tables for all patients on a particular date. The value for the Date column is obtained using the EXTRACT function on the Timestamp value for a performed service.

Billing Report for August 5, 1992

Patient_ID	Number_of_Services	Total_Cost
701	2	$1 500.00
705	4	$2 100.00
713	1	$ 320.00
783	12	$2 400.00
795	2	$ 890.00

Frequency or Importance Ranking:

Report/Query Description: Billing report for a specific date

Table: Patient_Service_History **Number of Output Rows:** 2 000

Input Value Or Source	Current_Date	Service Service_ID
Access Column(s)	Current_Date	Service_ID

Table: Service **Number of Output Rows:** 2

Input Value Or Source	Patient_Service_History Service_ID	
Access Column(s)	Service_ID	

Table: **Number of Output Rows:**

Input Value Or Source			
Access Column(s)			

Table: **Number of Output Rows:**

Input Value Or Source			
Access Column(s)			

Table Form

Purpose

The Table form has two purposes.

- Extend the fully attributed table model created through the logical database design by collecting table- and column-level data demographic information and adding it to the model.
- Provide input to the physical design of the database.

This form introduces data demographics to the ATM process. Supporting analysis requires you to determine various cumulants and then to add that data to information derived from the Column Names and Constraints and Report/Query Analysis forms.

The form supports the seventh, eighth, and ninth steps in the ATM process (see the table on page 7-3).

Approaches to Filling Out the Table Form

Because the determination of data demographics for columns that will be treated as multicolumn indexes is far more complex than the determination of similar data for single column index columns, the sample analysis and the way you fill out the Table form differs slightly for the single column and multicolumn situations.

The following table points to the Table form topics that apply to the two situations.

Topic	Applies to Single Column Data?	Applies to Multicolumn Data?
"Table Form"	Yes	Yes
"Table Form: Basic Information"	Yes	Yes
"Table Form: Column-Level Information"	Yes	Yes
"Table Form: Miscellaneous Column-Level Information"	Yes	Yes
"Table Form: Access Information"	Yes	Yes
"Table Form: Data Demographics for Single Column Objects"	Yes	No
"Maximum and Typical Column Value Frequencies"	Yes	Yes
"Table Form: Data Demographics for Multicolumn Objects"	No	Yes

Types of Information Collected for the Table Forms

Table form information falls into three broad categories.

- Miscellaneous
- Access
- Data demographics

Miscellaneous Information

The following Table form information can be described as miscellaneous.

- Column name
- Identity column
- Primary Key/Foreign Key
- Constraint number
- Primary Index/Secondary Index
- Sample Data

Transcribe this information from the completed Column Names and Constraints form to the Table form.

Access Information

The following Table form information describes value and join accesses.

- Value access frequency
- Join access frequency
- Join access rows

Transcribe this information from the completed Report/Query Analysis form to the Table form.

Data Demographics Information

The following Table form information describes the demographics of your data.

- Distinct values
- Maximum rows/value
- Rows/null
- Typical rows/value
- Change rating

This information applies only to those columns that have been identified by the Report/Query Analysis activity as taking part in value and join accesses.

The Table form introduces data demographics to the ATM process for the first time. The sources for this information are varied and you might have to use a variety of resources to obtain the data.

Details of this activity are provided in "Table Form: Data Demographics for Single Column Objects" on page 7-42.

Filling Out the Table Form

Because the Table form is both complicated and extensive, the next several topics provide a brief overview of its components as well as instructions for how to derive the information requested.

The following list itemizes the topics that apply to filling out the Table forms.

- "Table Form Example" on page 7-37
- "Table Form: Basic Information" on page 7-38
- "Table Form: Miscellaneous Column-Level Information" on page 7-40
- "Table Form: Access Information" on page 7-41
- "Table Form: Data Demographics for Single Column Objects" on page 7-42
- "Maximum and Typical Column Value Frequencies" on page 7-45
- "Table Form: Data Demographics for Multicolumn Objects" on page 7-50

Table Form Example

Introduction

This example is the first page of the Table form for the Patient_Service_History table. A continuation page for this table, which collects data for multicolumn index candidates, is illustrated in "Table Form: Data Demographics for Multicolumn Objects" on page 7-51.

Maximum Rows/Value **Typical Rows/Value**

Table **Page: of**		**ELDM Page:** **System:** Patient Tracking	
Table Name: Patient_Service_History	**Table Type:** History	**Cardinality:** 730 000	**Data Protection:** RAID 5
Column Name	Patient_ID	Service_ID	Timestamp
PK/FK/ID	PK		
	FK	FK	
Constraint Number			
Value Access Frequency	1 000	52	365
Join Access Frequency		232	
Join Access Rows		1 400 000	
Distinct Values	1 000	125	365
Maximum Rows/Value	3 650	15 000	4 000
Rows/Null	0	0	0
Typical Rows/Value	720	5840	2000
Change Rating	0	0	0
PI/SI			
Sample Data	701 702 703	801 801 802	19920101 19920105 19920105

Table Form: Basic Information

Introduction

This topic explains how to fill out the general information requests named at the top of the Table form.

Sources for the Required Information

The following table points to the sources for the data requested at the top of the Table form.

<table>
<tr><th>Information</th><th>Source</th></tr>
<tr><td>System</td><td>System form</td></tr>
<tr><td>Table name</td><td>Entity name from your logical data model</td></tr>
<tr><td>Table type</td><td>Entity type from your logical data model</td></tr>
<tr><td>Row count</td><td>Varies.
<table>
<tr><th>IF there is …</th><th>THEN …</th></tr>
<tr><td>an existing database that contains the information</td><td>query that database for the cardinality of the table in one of two ways.
• Examine current statistics for the table.
• Perform a simple <code>SELECT COUNT (*)</code> statement against the table.</td></tr>
<tr><td>no existing electronic database that contains the information</td><td>consult with your users to make a best guess cardinality estimate.
Base your estimates on known maxima, not averages.</td></tr>
</table></td></tr>
<tr><td>Data protection</td><td>Varied, depending on how your company operates. The following items are all likely sources for this information.
• Users
• Enterprise data model
• IT policies and procedures
The following methods for protecting data are available. Note that while you can specify fallback at the table and join index level, RAID protection applies to your entire hardware configuration.
• Fallback
• RAID 1
• RAID 5
• RAID S</td></tr>
</table>

Table Form: Column-Level Information

Introduction

Column-level information derives from several different sources. This topic parses column-level information into three types. Additional topics examine each of those three types in greater detail.

Column-Level Information Types

Column-level information separates into three categories.

Category	Topic to See for More Detailed Information
Miscellaneous	"Table Form: Miscellaneous Column-Level Information" on page 7-40
Value and join access	"Table Form: Access Information" on page 7-41
Data demographics	"Table Form: Data Demographics for Single Column Objects" on page 7-42

Table Form: Miscellaneous Column-Level Information

Introduction

This topic explains how to fill out all the column-level information not categorized as either access or data demographics data.

Sources for the Required Information

The following table points to the sources for the miscellaneous column-level data requested by the Table form.

Information	Source
Column name	Column name from your logical data model.
PK/FK/ID	Primary and foreign key designations from your logical data model. Identity column designation identified by this step.
Constraint number	Constraints form.
PI/SI	Primary and secondary index designations identified by this step.
Sample data	Sample data from your logical data model.

Table Form: Access Information

Introduction

This topic explains how to fill out the column access information requests named in the Table form.

Sources for the Required Information

The following table points to the sources for the access data requested by the Table form.

Information	Source
Value access frequency	Report/Query Analysis form
Join access frequency	
Join access rows	

Table Form: Data Demographics for Single Column Objects

Introduction

This topic explains how to derive column demographics and how to add that information to the Table form.

Capture demographic data only for access and join column sets and primary key column sets.

By collecting this information, you are emulating some of the fundamental data collection activities performed by the COLLECT STATISTICS statement. The data is used for similar purposes as well: to help you to model your physical database in a way that its performance across all systems and applications is optimal.

Sources for the Required Information

The following table points to the sources for the data demographics requested by the Table form.

<table>
<tr><th>Information</th><th>Source</th></tr>
<tr><td>Distinct values</td><td>Varies.
This parameter determines how many unique values (including nulls) exist for the specified column.
<table><tr><th>IF there is …</th><th>THEN …</th></tr><tr><td>an existing electronic database that contains the information</td><td>query that database for the number of distinct values in the column in one of two ways.
• Examine current statistics for the table.
• Perform a simple <code>SELECT COUNT DISTINCT</code> statement over the required rows.</td></tr><tr><td>no existing electronic database that contains the information</td><td>consult with your users to make a best guess estimate.</td></tr></table></td></tr>
</table>

<table>
<tr><th>Information</th><th>Source</th></tr>
<tr><td>Maximum rows/value</td><td>Varies.
This parameter determines the maximum value for the specified column.<table><tr><th>IF there is ...</th><th>THEN ...</th></tr><tr><td>an existing electronic database that contains the information</td><td>query that database for the value that occurs most frequently in the column in one of two ways.
Examine current statistics for the table.
Perform a simple <tt>SELECT MAX column_name</tt> statement over the required rows.</td></tr><tr><td>no existing electronic database that contains the information</td><td>consult with your users to make a best guess estimate.
Base your estimates on known maxima, not averages.</td></tr></table></td></tr>
<tr><td>Rows/null</td><td>This parameter determines how many rows are null for the specified column.<table><tr><th>IF there is ...</th><th>THEN ...</th></tr><tr><td>an existing electronic database that contains the information</td><td>query that database for the number of rows having a null in this in one of two ways.
Examine current statistics for the table.
Perform a simple <tt>SELECT COUNT column_name WHERE column_name IS NULL</tt> statement over the required rows.</td></tr><tr><td>no existing electronic database that contains the information</td><td>consult with your users to make a best guess estimate.
Base your estimates on known maxima, not averages.</td></tr></table></td></tr>
</table>

<table>
<tr><th>Information</th><th>Source</th></tr>
<tr><td>Typical rows/value</td><td>This parameter determines a typical number of rows per value for the specified column.

The typical number of rows per value for a column is not necessarily the average value, and the frequency can be so widely dispersed that a reasonably typical value cannot be determined.

See “Maximum and Typical Column Value Frequencies” on page 7-45 for examples of how to determine a value for this parameter.</td></tr>
<tr><td>Change rating</td><td>Varied, depending on how your company operates. The following items are all likely sources for this information.
• Users
• Enterprise data model
• IT policies and procedures

Note that the Teradata Index Wizard uses column change ratings as a parameter in determining the candidacy of columns for use as indexes.

See the following topics for more information about how the Teradata Index Wizard uses this information:
• The chapter "Database Query Analysis" in Teradata RDBMS SQL Reference, Volume 2.
• "INITIATE INDEX ANALYSIS" in the chapter "SQL Data Definition Statement Language Syntax" in Teradata RDBMS SQL Reference, Volume 4. The heading "How ChangeRate Is Evaluated by INITIATE INDEX ANALYSIS" documents the column change ratings used by the Teradata Index Wizard explicitly.

<table>
<tr><th>Code</th><th>Description</th></tr>
<tr><td>0</td><td>Data for this column never changes.

Examples include primary key columns and columns that contain historical information.</td></tr>
<tr><td>1</td><td>Data for this column rarely changes.</td></tr>
<tr><td>2 - 8</td><td>User-determined.

These codes cover anything not covered by codes 0, 1, and 9.</td></tr>
<tr><td>9</td><td>Data for this column frequently changes.</td></tr>
</table>
</td></tr>
</table>

Maximum and Typical Column Value Frequencies

Introduction

This topic compares the concepts of maximum value and typical value and indicates methods for estimating the typical value for a column.

Variables are often distributed in such a way that they have no “typical” value, in which case the average, which represents the maximum likelihood value, is the only reasonable choice.

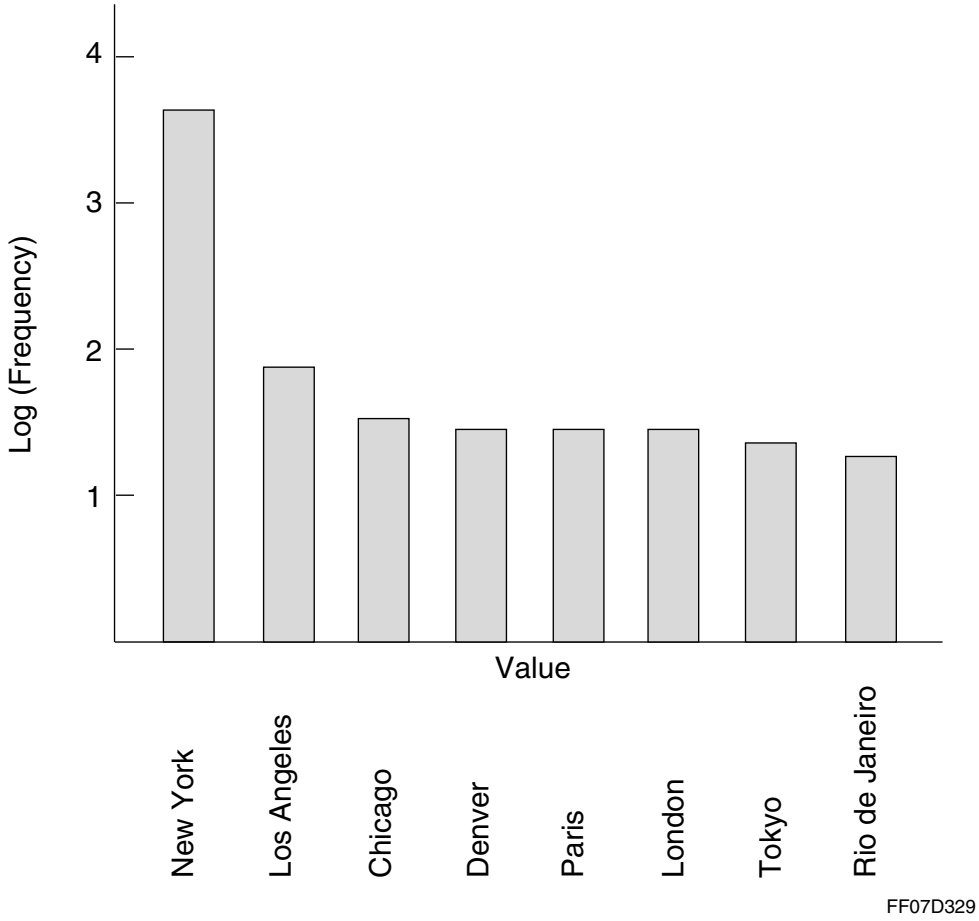
Each of these case studies examines different demographics for the same column. In each case, the data examined is for a City column.

Case Study 1

The following table indicates the number of distinct occurrences of City column values.

City Value	Frequency of Value	Log Frequency of Value
New York	4 000	3.602
Los Angeles	80	1.903
Chicago	35	1.544
Denver	30	1.477
Paris	30	1.477
London	30	1.477
Tokyo	25	1.398
Rio de Janeiro	20	1.301

The following histogram graphs the logarithm of the number of rows as a function of row values.



The maximum value for this set is 4 000, but what is the typical value?

It is easy to see by visual inspection of the table on page 7-45 that the typical value for a variable is about 30. Note that the *average* value for this variable is 531.25, which is by no means typical of the values for the variable.

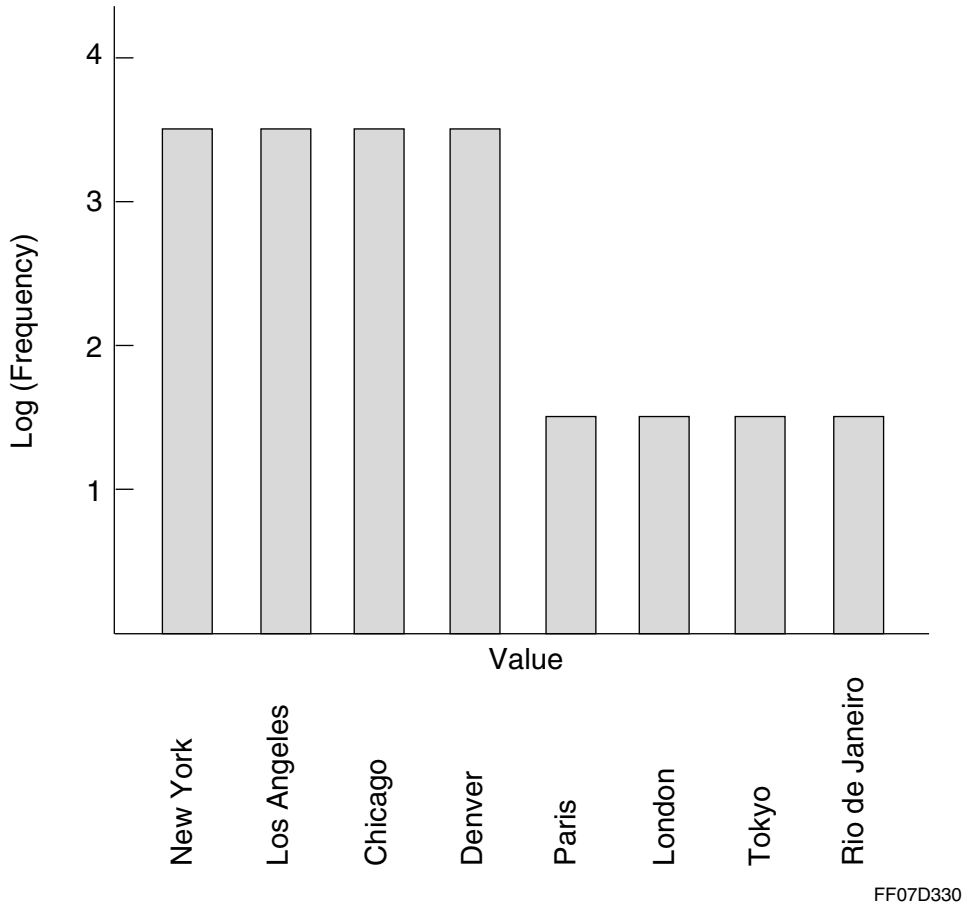
Maximum Value	Typical Value
4 000	30

Case Study 2

The following table indicates the number of distinct occurrences of City column values.

City Value	Frequency of Value	Log Frequency of Value
New York	4 000	3.602
Los Angeles	4 000	3.602
Chicago	4 000	3.602
Denver	4 000	3.602
Paris	30	1.477
London	30	1.477
Tokyo	30	1.477
Rio de Janeiro	30	1.477

The following histogram graphs the logarithm of the number of rows as a function of row values.



The maximum value for this set is 4 000, but what is the typical value?

It is impossible to determine a typical value for the scenario provided by this case history. When you encounter a situation like this, the optimum solution is to use the worst case as your typical value. In this case, that value is 4 000. Note that the *average* value for this variable is 2 015, which is not only not a typical value for the distribution of the variable, it is *never* a value for the variable in this case.

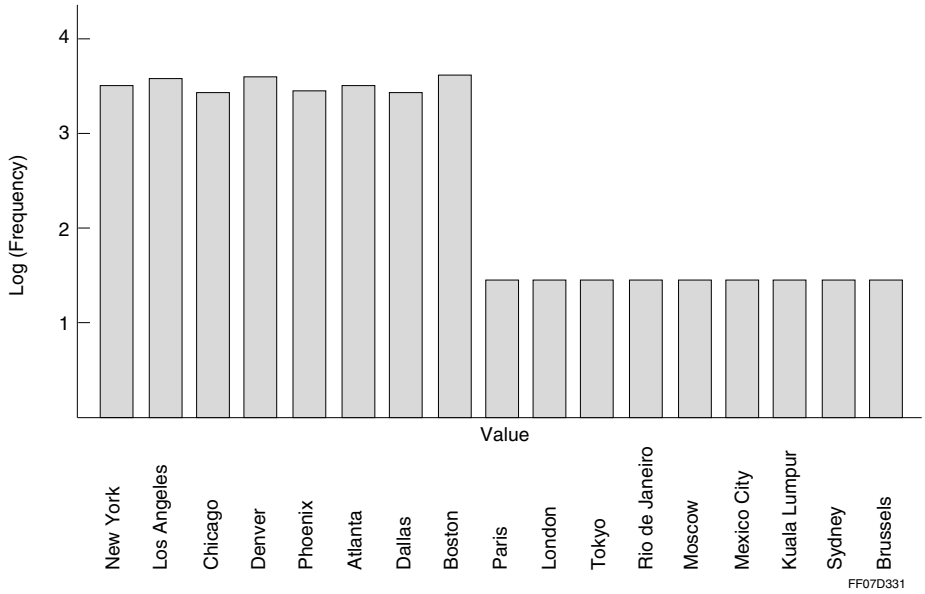
Maximum Value	Typical Value
4 000	4 000

Case Study 3

The following table indicates the number of distinct occurrences of City column values.

City Value	Frequency of Value	Log Frequency of Value
New York	4 000	3.602
Los Angeles	4 100	3.613
Chicago	3 800	3.580
Denver	4 200	3.623
Phoenix	3 900	3.591
Atlanta	4 000	3.602
Dallas	3 800	3.580
Boston	4 150	3.618
Paris	30	1.477
London	30	1.477
Tokyo	30	1.477
Rio de Janeiro	30	1.477
Moscow	30	1.477
Mexico City	30	1.477
Kuala Lumpur	30	1.477
Sydney	30	1.477
Brussels	30	1.477

The following histogram graphs the logarithm of the number of rows as a function of row values.



The maximum value for this set is 4 200, but what is the typical value?

It is impossible to determine an accurate “typical” value for the scenario provided by this case history. Like the scenario presented by “Case Study 2” on page 7-47, the distribution of values has two peaks at widely diverse points in the distribution. Unlike “Case Study 2,” the value set clustered around the value 4 000 is not constant.

When you encounter a situation like this, the optimum solution is to use a value around which the largest values cluster as your typical value. In this case, that value is 4 000.

Maximum Value	Typical Value
4 200	4 000

Table Form: Data Demographics for Multicolumn Objects

Introduction

This topic describes how to determine data demographics for multicolumn objects such as composite indexes.

Sources for the Required Information

The following table points to the sources for the multicolumn data demographics requested by the Table form. With the exception of the information documented here, sources are identical to those documented by "Sources for the Required Information" on page 7-42.

Information	Source
Distinct values	See the following topics. • "Calculating Multicolumn Demographics" on page 7-52 • "Example Cases for These Guidelines" on page 7-53
Maximum rows/value	
Rows/null	
Typical rows/value	

Example

This example form, which continues the form illustrated in "Table Form Example" on page 7-37 indicates two multicolumn index candidates.

Information acquisition for the second of the two multicolumn specifications is made more simple because the specified columns constitute the primary key for the table.

Because the index candidates are both composite, you cannot use an identity column to define them.

This implies certain demographics by default, as described by the following list. The listed implications hold true irrespective of whether the primary key is uni- or multicolumnar.

- All demographics must be unique
- No demographics can be null.

  Therefore, the value for Rows Null is always 0.
- The value for Distinct Values always matches the Cardinality for the table and the value for Maximum Rows/Value is always 1.

Note that the value for Change Rating is also 0 by default because primary key values should never be changed.

Table, continued **(Multicolumn index considerations)** **Page: of**		**ELDM Page:** **System:** Patient Tracking	
Table Name: Patient_Service_History	**Table Type:** History	**Cardinality:** 730 000	**Data Protection:** RAID 5
Column Name	Patient_ID Service_ID	Patient_ID Service_ID Timestamp	
PK/FK/ID	FK	PK	
Constraint Number			
Value Access Frequency		120 000	
Join Access Frequency	2 200	300 000	
Join Access Rows	200 000	300 000	
Distinct Values	8 000	730 000	
Maximum Rows/Value	365	1	
Maximum Rows Null	0	0	
Typical Rows/Value	90	1	
Change Rating	0	0	
PI/SI			

Demographic Trends as a Function of the Number of Columns

Certain characteristics of the demographics of a candidate index change in predictable ways as more columns are added to the index definition.

Selectivity, defined in more detail in "Selectivity of Indexes" on page 8-3, is a measure of the ability of an index to return a highly selective subset of rows from a table. The higher the selectivity, the fewer rows retrieved.

- The number of distinct values increases because each additional index column further enhances the singularity of the row.
- The number of rows per value decreases because the number of values increases proportionately.
- The selectivity of the candidate index increases because any set of index column values points to fewer rows.

The following table summarizes these trends.

Index Columns	Number of Distinct Values	Number of Rows Per Value	Selectivity
State	Least	Most	Lowest
State + Zip Code	?	?	?
State + Zip Code + Last Name	Most	Least	Highest

Calculating Multicolumn Demographics

You must have an understanding of the underlying data before you can undertake a legitimate calculation of the demographics for multicolumn situations.

The following table provides some calculation guidelines for determining the number of distinct values in a binary (two column) situation. The guidelines scale as more columns are added to the candidate index.

IF distinct column_1 values ...	THEN the total number of distinct values in the two-column index equals ...	ELSE the number of distinct values is ...
can be paired with distinct values from column_2	the product of the number of distinct values in column_1 and the number of distinct values in column_2	approximately equal to the product of the number of distinct values in column_1 and the average number of values in column_2 that can be paired with a single column_1 value.

Example Cases for These Guidelines

The following examples illustrate the guidelines stated in "Calculating Multicolumn Demographics."

- Consider the following table fragment. The candidate index for which multicolumn demographics are to be estimated is ProductCode, ColorCode.

...	ProductCode	ColorCode	...
…	…	…	…
…	…	…	…

  The product (ProductCode) * (ColorCode) is true iff every product is available in every color.

- Consider the following table fragment The candidate index for which multicolumn demographics are to be estimated is StateCode, ZipCode.

...	StateCode	ZipCode	...
…	…	…	…
…	…	…	…

  - The product (StateCode) * (ZipCode) is not an accurate estimate of the number of distinct values in the multicolumn index because the resulting product has fewer distinct values than the product of distinct StateCode and distinct ZipCode values.
  - If you cannot pair every value in the StateCode column with every value in the ZipCode column, then the number of distinct values in the multicolumn index is approximately equal to (Number of distinct StateCode values) * (Average number of ZipCode values that can be paired with a single StateCode value).

    For example, the number of distinct values is the product of the number of states in the US and the average number of zip codes per state, which we estimate to be 20.

    Therefore, the number of distinct values for the StateCode + ZipCode index would be roughly 50 * 20, or 1 000.

- The same considerations hold most of the other rows per value calculations.

  For example, if every value of column_1 can be paired validly with every value for column_2, then the result of the division is a fairly accurate estimate.

  For each value estimated, the minimum is never less than the number of column_1 rows per value divided by the number of distinct column_2 values.

The exception is the count of Rows Null (labeled Rows/Null). See "How to Treat Rows Null Counts for Composite Indexes" on page 7-54 for information on handling multicolumn indexes that are wholly or partially null.

How to Treat Rows Null Counts for Composite Indexes

An index can be partly or wholly null. Consider the following example:

EmployeeNumber	DepartmentNumber
PK	
FK	FK
103794	7012
?	7012
105378	?
?	?

- The first row has both Employee and Department numbers.
- The second row has a null Employee number and a Department number.
- The third row has an Employee number and a null Department number.
- The fourth row has nulls for both its Employee number and its Department number.

All four rows are valid primary index rows. Should they all contribute to the count for the Rows Null value or should they be treated differently?

The following table provides recommended treatment guidelines for partially and wholly null composite indexes.

IF a composite index value is ...	THEN you should ...
wholly null	add it to the Rows Null count.
partially null	*not* add it to the Rows Null count.

Section 3:

Physical Database Design, Part 1: Indexing

Chapter 8:

Teradata Indexes: The Basics

Teradata provides several indexing options for optimizing the performance of your relational databases. This chapter describes Teradata indexing in general as well introducing the individual types of indexes you can specify.

Each index type is described briefly here as follows:

- "Primary Indexes" on page 8-16
- "Secondary Indexes" on page 8-18
- "Join Indexes" on page 8-20
- "Hash Indexes" on page 8-23
- "Reference Indexes" on page 8-25

Separate chapters describe more detailed information about the various index types and how they are used.

FOR more information about this topic ...	Refer to this chapter ...
Primary indexes	Chapter 9: "Primary Indexes"
Secondary indexes	Chapter 10: "Secondary Indexes"
Join indexes	Chapter 11: "Join and Hash Indexes"
Hash indexes	

Teradata primary and secondary indexes are not really indexes, but hash keys. Definitions of indexes and hash keys are provided in "Indexing and Hashing" on page 8-26 and "Tradeoffs Between Hashing and Indexing" on page 8-28.

Teradata hashing is described in the following topics.

- "Teradata Hashing Algorithm" on page 8-32
- "Hash Collisions" on page 8-34
- "Hash Maps" on page 8-35
- "Hash-Related Functions" on page 8-42
- "Hashed Row Access" on page 8-45

Like primary and secondary indexes, hash and join indexes are not true indexes; instead, they are user-inaccessible base tables that the Optimizer can use to resolve queries covered by the join index without accessing the underlying base tables.

Definitions for Parsing Engine Software

The following table defines some of the PE software terminology used in this manual.

Component	Definition
Parser	Software that checks SQL statements for the following properties: • Proper SQL syntax • Proper use of the SQL lexicon • Access permissions for requested database objects • Existence of requested database objects
Syntaxer	Component of the Parser that checks SQL statements for proper syntax. If syntax is incorrect, an appropriate error message is returned to the requestor.
Lexer	Component of the Parser that checks SQL statements for proper use of the SQL lexicon. If lexical use is incorrect, an appropriate error message is returned to the requestor.
Resolver	Component of the Parser that checks SQL statements for access permissions to, and existence of, requested database objects. If the requestor does not have appropriate access permissions or if a requested database object does not exist, then an appropriate error message is returned to the requestor.
Optimizer	Software that takes error-free output from the Parser and performs all the following tasks in order to optimize the cost of an SQL query. • Restructures SQL statements to make them more efficient • Determines an optimal cost method for retrieving data rows • Determines an optimal cost method for joining data tables • Creates an access plan for the query • Creates a join plan for the query when required • Compiles the access and join plans into machine code
Generator	Software that takes the access plan produced by the Optimizer and converts it into a set of discrete tasks for the Database Manager software to perform. These tasks are referred to as AMP steps. The hashing algorithm is also a component of the Generator.
Dispatcher	Software that manages the AMP steps it receives from the Generator. Each AMP step is dispatched to the appropriate AMP set over the BYNET. The next step in sequence is not dispatched until its predecessor has completed its task.

Indexes

Definition

An index is a mechanism that can be used by the SQL query optimizer to make table access more performant. Indexes enhance data access by providing a more-or-less direct path to stored data and avoiding the necessity to perform full-table scans to locate the small number of rows you typically want to retrieve or update.

The Teradata parallel architecture makes indexing an aid to better performance, not a crutch necessary to ensure adequate performance. Full-table scans are not something to be feared in the Teradata environment. This means that the sorts of unplanned, ad hoc queries that characterize the data warehouse process, and that often are not supported by indexes, perform very effectively for Teradata using full-table scans.

The classic index for a relational database is itself a file made up of rows having two parts.

- A (possibly unique) data field in the referenced table.
- A pointer to the location of that row in the base table (if the index is unique) or a pointer to all possible locations of the row (if the index is nonunique).

Because Teradata is a massively parallel architecture, it requires a more efficient means of distributing and retrieving its data. That more efficient method is hashing. With the exception of hash and join indexes, all Teradata indexes are based on row hash values rather than raw table column values, even though secondary, hash, and join indexes can be stored in order of their values to make them more useful for satisfying range conditions.

Selectivity of Indexes

An index that retrieves many rows is said to have low selectivity. By definition, an index with low selectivity typically accesses more than ten percent of the rows in the table on which it is defined.

An index that retrieves few rows is said to be highly selective. A highly selective index typically selects fewer than ten percent of the rows in the table on which it is defined. The more highly selective an index is, the more useful it is for enhancing performance

In some conditions, it is possible to link several nonunique secondary indexes with low selectivity together by bit mapping them. The result is effectively a highly selective index and a dramatic reduction in the number of table rows that must be accessed. See "NUSI Bit Mapping" on page 10-20 for further discussion of linking secondary indexes with low selectivity into a highly selective unit using bit mapping.

Row Hash and RowID

Teradata table rows are self-indexing with respect to their primary index and so require no additional storage space. When a row is inserted into a table, the file system stores the 32-bit row hash value of the primary index in place with the column data for the row.

Because row hash values are not necessarily unique, the Generator also produces a unique 32-bit numeric value (called the Uniqueness Value) that it appends to the row hash value, forming a unique RowID. This RowID makes each row in a table uniquely identifiable.

The first row inserted into a table having a specific row hash value is always assigned a uniqueness value of 0. Additional table rows having the same row hash value are assigned uniqueness values in a numerically increasing order. The rows are stored on disk, sorted in ascending order of RowID.

To determine the next uniqueness value for a row hash, the relational database manager reads through all the rows in the table belonging to that row hash to find the current high value, which is then incremented by one.

Uniqueness values are not reused except for the special case in which the highest valued row within a row hash is deleted from a table.

Index Hash Mapping

Rows are distributed across the AMPS using a hashing algorithm (see "Teradata Hashing Algorithm" on page 8-32). The hashing algorithm computes a row hash value based on the primary index. The row hash is a 32-bit value containing a 16-bit Destination Selection Word and a 16-bit remainder.

Teradata relational databases have 65 536 hash buckets. The hash buckets are distributed as evenly as possible among the AMPs on a system (see "Hash Maps" on page 8-35).

The BYNET interface board on each AMP maintains a hash map—an index of which hash buckets are assigned to which AMPs—that it uses to determine whether rows belong to it based on their row hash values. Row assignment is performed in a manner that ensures as equal a distribution of table rows as possible among all the AMPs in the configuration.

Advantages of Indexes

The intent of indexes is to lessen the time it takes to retrieve rows from a database. The faster the retrieval, the better.

Disadvantages of Indexes

Perhaps not so obvious are the disadvantages of using indexes.

- Index subtables must be updated each time an indexed column value in the base table is updated or deleted, or whenever a new row is inserted.

  This is only a consideration for secondary, hash, and join indexes in the Teradata environment. The more secondary, hash, and join indexes you have defined for a table, the larger the potential update maintenance downside becomes.

  Because of this, secondary, hash, and join indexes are generally inappropriate for OLTP situations.
- All Teradata secondary, hash, and join indexes are stored in subtables, so they exert a burden on system storage space.
- When fallback is defined for a table, a further storage space burden is created because secondary index subtables are also duplicated whenever fallback is defined for a table.

  You can optionally specify fallback on hash and join indexes as well. Hash and join indexes do *not* default to fallback if their underlying base tables are defined with fallback.

For these reasons, it is extremely important to use EXPLAIN statements to determine optimum data manipulation statement syntax and index usage before putting those statements (and indexes) to work in a production environment.

Teradata Indexes

Teradata provides four different index types.

- Primary index

  All Teradata tables require a primary index because the system distributes table rows to the AMPs based on their primary index values. Primary indexes are defined over two orthogonal dimensions: unique-nonunique and partitioned-nonpartitioned.
  - Unique primary index (UPI)
  - Nonunique primary index (NUPI)
  - Nonpartitioned primary index (NPPI)
  - Partitioned primary index (PPI)

  These can be mixed in any cross-dimensional combination, so the following four complete specifications for a primary index can all be defined in Teradata:
  - Unique nonpartitioned primary index
  - Nonunique nonpartitioned primary index
  - Unique partitioned primary index
  - Nonunique partitioned primary index.

- Secondary index
  - Unique secondary index (USI)
  - Nonunique secondary index (NUSI)
- Join index
  - Multitable join index
  - Single-table join index
- Hash index

Unique Indexes

A unique index column set, like a primary key column set, has a unique value for each row in a table.

Teradata defines two different types of unique index.

- Unique primary index (UPI)

  UPIs provide optimal data distribution and are typically assigned to the primary key for a table. When a NUPI makes better sense for a table, then the primary key is frequently assigned to be a USI.
- Unique secondary index (USI)

  USIs guarantee that each complete index value is unique, while ensuring that data access based on it is always a two-AMP operation.

You should define at least one unique index for any table you create. Any time you create a table having a nonunique primary index, you should also create a unique secondary index on the primary key for that table.

Nonunique Indexes

By definition, a nonunique index does not require its values to be unique. There are occasions when a nonunique index is the best choice as the primary index for a table.

How evenly a NUPI distributes the rows for a table across the AMPs depends on the relative singularity of the data comprising the primary index. As a rule of thumb, you should never assign a NUPI to a column set for which much more than 100 rows in the table are likely to have the same NUPI value.

NUSIs are also very useful for many decision support applications.

Partitioned and Nonpartitioned Indexes

A partitioned index assigns rows to a particular partition within an AMP based on a user-defined partitioning expression. Only primary indexes can be partitioned. Any index other than a partitioned primary index is, by default, a nonpartitioned index.

Join Indexes

There are two types of join index, each of which can also be simple or aggregated.

- Multitable join indexes are defined for join queries that are performed frequently enough to justify defining a prejoin on the joined columns.
- Single-table join indexes are defined to facilitate joins by hashing a frequently joined subset of base table columns to the same AMP as the table rows to which they are frequently joined. This enhanced geography eliminates BYNET traffic as well as often providing a smaller sized row to be read and joined.

For additional information, see "Join Indexes" on page 8-20 and Chapter 11: "Join and Hash Indexes."

Hash Indexes

Hash indexes are similar to single-table join indexes, though their definition is more similar to that of secondary indexes.

For additional information, see "Hash Indexes" on page 8-23 and Chapter 11: "Join and Hash Indexes."

Using EXPLAIN Statements to Determine the Usefulness of Indexes

One important thing to remember is that the selection of indexes to support a query is *not* under user control. The Teradata query optimizer neither requires nor accepts pragmas or hints. Additionally, you cannot specify resource options for Teradata indexes nor can you control index locking.

The only references made to indexes in the Teradata SQL dialect concern their definition, *not* their use. SQL data manipulation language statements do not provide a method for specifying indexes.

The implications of this behavior are severalfold.

- It is critically important to collect statistics regularly on all indexed and frequently joined columns to ensure that the Optimizer has access to current information on the demographics of the database so it can know how to best optimize any query or update made to the database. Maintaining fresh statistics is the only way to tune index requests in Teradata.

  For additional information concerning collecting and maintaining accurate database statistics, see "COLLECT STATISTICS (Optimizer Form)" in *Teradata RDBMS SQL Reference, Volume 4*.

  You can use the Teradata Statistics Wizard client utility to recommend which columns and indexes you should collect statistics on for particular query workloads. The Statistics Wizard also recommends when statistics should be recollected on those columns. See *Teradata Statistics Wizard User Guide* for further information.

- It is equally important to build your SQL queries and updates in such a way that you know their performance will be most optimal.

  Apart from good logical database design, there are two ways to ensure that you are accessing your data in the most efficient manner possible:

  - Use the EXPLAIN modifier (or the Visual Explain tool - see "Comparing EXPLAIN Reports Graphically" on page 8-8) to test various candidate queries or updates and to note which indexes are used by the Optimizer in their execution (if any) as well as examining the relative length of time required to complete the operation.
  - Use the Teradata Index Wizard client utility to recommend and validate a set of secondary indexes for a particular query workload. See *Teradata Index Wizard User Guide* for further information.

  EXPLAIN statement output provides you with the following basic information:

  - The step-by-step access method the Optimizer would use to execute the specified data manipulation statement given the current set of table statistics it has to work with.
  - The *relative* and worst case time estimates to perform the specified data manipulation statement.

  While you cannot rely on the reported statement execution time as an absolute, you *can* rely on it as a relative means for comparison with other candidate data manipulation statements against the same tables with the same statistics defined.

Even within the bounds of a constant system configuration a query plan can change significantly because of shifting data demographics. The only way to track these shifts and to ensure that the Optimizer is building the best possible query plans is to collect statistics frequently on your index and other frequently joined columns and then run new EXPLAINs.

Extensive information about how to use the EXPLAIN modifier is contained in "EXPLAIN Modifier" in *Teradata RDBMS SQL Reference, Volume 6*.

Information about using the Visual EXPLAIN utility is contained in *Teradata Visual Explain Getting Started Guide*.

Comparing EXPLAIN Reports Graphically

Teradata provides a sophisticated tool for analyzing and comparing the EXPLAIN reports generated by semantically identical, but syntactically different, queries or by the same query performed on different hardware platforms and configurations or under different software releases. This tool, called Visual EXPLAIN, provides facilities for displaying and comparing EXPLAIN statement outputs graphically.

Analyzing EXPLAIN Reports Using SQL

SQL provides another powerful facility for analyzing the query plans the Optimizer creates for various data manipulation statements. This feature, called the Query Capture Facility, permits a DBA to capture access plan information from the query optimizer using the utility statements DUMP EXPLAIN or INSERT EXPLAIN and then analyze the captured data using standard SQL DML statements.

The captured query plans are stored in a relational database called the Query Capture Database (QCD) provided for that purpose. QCD is a user database whose space is drawn from available permanent disk space: it is not a set of system tables.

Note that the Visual Explain tool also uses these tables for its analyses.

See *Teradata RDBMS SQL Reference, Volume 2* for more information on the Query Capture facility.

See the online help files for the Visual Explain tool for more information about its capabilities.

Target Level Emulation

A related facility, Target Level Emulation, permits you to capture the environmental cost data and random AMP sampled statistics of a production system and then analyze various queries in a test environment that emulates your production system.

See *Teradata RDBMS SQL Reference, Volume 2* for more information on this tool.

Further Information

For additional performance-related information about how to use the access and join plan reports produced by the EXPLAIN statement to optimize the performance of your databases, consult the following manuals.

- *Teradata RDBMS SQL Reference, Volume 6*
- *Teradata RDBMS Performance Optimization*

Keys Versus Indexes

Definition: Key

There are several different meanings for the term key in relational database management. When reading this manual, you should always assume that if the term key is used without a qualifying adjective, the reference is to the primary key for a table.

The following table defines the various uses of key in the context of relational database management. Note that in every case, a key is an identifier and not a retrieval mechanism. In every case, you should read the word *field* to mean "one field or several fields in combination." The expression "column set" is used throughout this manual in the same way; to indicate that the table object under discussion, whether it be a key or index, can be made up of one or more columns.

Term	Definition
Alternate key	Any candidate key not selected to be the primary key for a table.
Candidate key	A field that uniquely identifies a row. Every normalized table has at least one candidate key, and most have *only* one. When multiple candidate keys are identified during logical design, one is selected to be the primary key and the candidate keys not selected to be the primary key are then referred to as alternate keys.
Child key	A foreign key that is a primary key in another table.
Foreign key	A field that corresponds to the primary key of a different, but related, table. Foreign keys are used to maintain referential integrity. They are often useful as NUPIs when you want rows from two or more related tables to hash to the same AMP.
Grouping key	A field used to group the rows of a table in a particular way.
Hash key	A field used to compute the row hash value for a row.
Order key	A field used to store the rows of a table in a particular order.
Parent key	A primary key that is a foreign key in another table.
Primary key	A field that uniquely identifies a row. A primary key can never be null.
Search key	A field used to match a corresponding field in a searched table.
Sort key	A field used to sort the rows of a table. Note that the sorted table in this case is not necessarily a base table; for example, it can be an intermediate result such as a spool file.

Definition: Index

Teradata uses the term *index* in two different ways, neither of which is a true index.

- Primary and secondary indexes are row hash values. See "Indexing and Hashing" on page 8-26.
- Join and hash indexes are denormalized, inaccessible base tables that can substitute for base tables in various queries, particularly queries that require join processing.

  Hash indexes can also be used to directly access a base table to complete a query cover. A join index can be used for this purpose if it has been defined using any of the following components.
  - The UPI of the underlying base table
  - The keyword ROWID[1]
  - The NUPI of the underlying base table and the keyword ROWID
  - The NUPI of the underlying base table and the USI of the underlying base table

    It is preferable to specify NUPI and ROWID over NUPI and USI.

In the most general definition possible, an index is a field or combination of fields in a table used to access its data in the most performant means possible. The primary index for a table is frequently defined on the same column set that is identified as the primary key during logical database design, but nonunique, nonprimary key fields are also used as the primary index for a table.

Summary: Differences Between Keys and Indexes

The following table summarizes the differences between keys and indexes using the primary key and primary index for purposes of comparison.

Primary Key	Primary Index
Component of logical data model.	Component of physical data.
Used to maintain referential integrity.	Used to retrieve data.
Must be unique.	Can be nonunique.
Uniquely identifies each row.	Partitions table rows across the AMPs. Might or might not uniquely identify each row depending on whether it is defined as a UPI or a NUPI.

1 You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement. See "CREATE JOIN INDEX" in *Teradata RDBMS SQL Reference, Volume 4*.

Primary Key	Primary Index
Values can never be changed.	Values can be changed.
Cannot be null.	Can be null.
Does not imply access path.	Defines the most common access path.
Not required for physical table definition.	Required for physical table definition.

SQL and Indexes

Introduction

The five basic SQL data manipulation language statements, SELECT, INSERT, DELETE, UPDATE, and MERGE, all use indexing to manipulate table rows.

The following table summarizes the I/O types required by these four statements.

Statement	Types of I/O Required
SELECT	Read
INSERT	Read and Write
DELETE	
UPDATE	
MERGE	

Role of the WHERE Clause in SQL DML

SELECT, DELETE, and UPDATE statements can all specify a WHERE clause. The WHERE clause filters row selection, limiting access to only those rows or row ranges specified in the clause. It is equivalent to the RESTRICT (or SELECT) operator of the relational algebra.

The MERGE statement uses an ON clause predicate to filter row selection.

Using a carefully constructed WHERE or ON clause to limit the number of rows processed by a DML statement enhances throughput markedly. If no WHERE clause is specified, then all rows participate in the selection, deletion, or update activities specified by a particular DML statement.

Note that the WHERE CURRENT OF clause available for DELETE and UPDATE statements in embedded SQL and stored procedures acts on preselected spool rows and does not affect I/O.

How the Optimizer Uses the WHERE and ON Clauses

The Optimizer uses WHERE or ON clause predicates as the foundation on which to build its access and join plans. Many predicates, or search conditions as they are often called, invoke the selection of indexes by the Optimizer if the appropriate indexes have been defined on the columns specified by the predicate condition.

Specify these conditions whenever possible because they promote optimal performance and are selected by the Optimizer whenever table statistics, environmental cost parameters, and other demographics suggest they would decrease response time.

Predicates That Invoke Indexes

The following search conditions invoke indexes if they are defined on the columns specified in a WHERE clause predicate.

- column_name = value
- column_name IS NULL
- column_name IN (subquery)
- column_name IN (value_list)
- table_1.column_x = table_1.column_y
- table_1.column_x = table_2.column_x
- search_condition_1 AND … search_condition_*n*
- search_condition_1 OR … search_condition_*n*
- column_name = ANY
- column_name = SOME
- column_name = ALL
- any range condition satisfied by a value-ordered index

Predicates That Invoke Full-Table Scans

The following search conditions typically invoke a full-table scan of the base table or of a secondary index subtable when used as a HAVING or WHERE clause condition because they do not specify a specific value or set of values.

- nonequality comparisons
- column_name IS NOT NULL
- column_name NOT IN (subquery)
- column_name NOT IN (value_list)
- column_name BETWEEN value_1 AND value_2
- join_condition_1 OR join_condition_2
- NOT (condition)
- table_1.column_x * expression = value
- table_1.column_x * expression = table_1.column_y
- column_1 || column_2 = value
- column_name LIKE expression
- INDEX (column_name)
- SUBSTRING (column_name)
- SUM (numeric_expression)
- MINIMUM (numeric_expression)
- MAXIMUM (numeric_expression)
- AVERAGE (numeric_expression)
- COUNT (value_expression)
- DISTINCT
- ANY
- ALL
- no WHERE clause specified for query

All Teradata indexes can optionally be stored in the order of their values as well as in the more standard hash order. If a value-ordered index is defined on the column that satisfies a range predicate from this list, then a full-table scan is not necessary to satisfy the condition and the Optimizer chooses to scan the value-ordered index subtable instead.

Primary Indexes

Purpose

Teradata partitions tables horizontally across all AMPs on a system. Rows are assigned to AMPs based on the value of their primary index. The determination of which hash bucket, and hence which AMP the row is to be stored on, is made solely on the row hash value of its primary index.

Many retrievals also use the primary index, though others might use a secondary index, a hash or join index, a full-table scan, or a mix of several different index types.

Mandatory Use

Each Teradata table must have a primary index. If you fail to explicitly define a primary index when you create a table, one of several situation-dependent default strategies is used to define one for the table. See Chapter 9: "Primary Indexes," for details.

Restrictions

Only one primary index can be defined on a table.

No more than 64 columns can be specified for a primary index definition.

Types of Primary Index

A primary index can be defined over two orthogonal dimensions.

- Unique or nonunique
- Partitioned or nonpartitioned

The Unique—Nonunique Dimension

The only physical consideration this choice raises is the possibility of not distributing the rows for a table evenly across the AMPs.

The most likely scenario causing an uneven row distribution would be if you did not select a *nearly* unique column to be a nonunique primary index, in which case the distribution of rows for the table across the AMPs can be uneven (sometimes the words "lumpy" or "skewed" are used to describe this kind of distribution).

Do not interpret this to mean that NUPIs necessarily, or even usually, cause the distribution of rows across the AMPs to be skewed. Particularly when the number of rows in a table is very large, it is possible to achieve an even distribution of rows with a NUPI, even if it is not nearly unique.

Skewed distributions also occur for *unique* primary indexes whenever the number of rows in a table is small relative to the number of AMPs in the configuration, particularly when there are fewer rows than AMPs.

Excessively uneven distribution of rows causes performance problems, particularly for large tables, and should be avoided.

The Partitioned—Nonpartitioned Dimension

Partitioned primary indexes (PPIs) are defined for tables that are frequent targets of range or partition-based queries. A common example of this is queries that involve date ranges.

As is true for nonpartitioned primary indexes (NPPIs), partitioning and its parameters are specified either with the CREATE TABLE statement when the table is created or with the ALTER TABLE statement after the table has already been created. The rows of a table having a PPI are hashed to the same AMPs they would be hashed to if the primary index were not partitioned. Once assigned to an AMP, PPI rows are further assigned to, and stored in row hash order within, their respective partitions.

NPPIs behave identically with the exception that they are not assigned to partitions after they are hashed to their AMP. You might want to conceptualize NPPI table row distribution as being like PPI table row distribution when all rows are assigned to a single partition.

NPPI rows are stored in row hash order on their respective AMPs.

See Chapter 9: “Primary Indexes,” for additional details about partitioned and nonpartitioned primary indexes.

Primary Index Storage

Primary indexes are stored in-line with the row they index.

Because of this, you cannot drop a primary index. Instead, you must drop the entire table definition and then recreate it with the new primary index.

The primary index for a table is stored in hash-order for a standard index or in hash-order within partition for a partitioned index.

Further Information

See Chapter 9: “Primary Indexes” for additional usage information and design tips about primary indexes.

Secondary Indexes

Purpose

Secondary indexes provide faster set selection. The term set selection refers to a query that returns multiple rows.

Not all queries are based on primary index retrievals. It is common for WHERE clause search criteria to be based on columns other than those making up the primary index for a table, and such occasions often benefit greatly if a secondary index is defined for the column defined for the search criterion.

Secondary indexes are frequently selected by the Optimizer when a search condition cannot be satisfied with a primary index retrieval. The Optimizer also selects secondary indexes for query plans when they completely or partially cover a query.

Many retrievals also use the primary index, though others might use a secondary index, a hash or join index, a full-table scan, or a mix of several different index types.

Mandatory Use

Secondary indexes are never required and are rarely recommended for tables that are accessed by OLTP applications. They are highly recommended for decision support applications because they often provide superior performance to full-table scans. See Chapter 10: “Secondary Indexes,” for details.

Restrictions

Excluding the primary index, you can only define 32 indexes on a table. These 32 indexes can be any combination of secondary, hash, and join indexes.

No more than 64 columns can be included in a secondary index definition.

Types of Secondary Index

A secondary index can be defined as unique or nonunique.

Unique secondary indexes are typically defined on the primary key of a table that has a nonunique primary index.

Secondary Index Storage

Secondary indexes are stored in secondary index subtables.

Teradata partitions unique secondary index rows on the row hash for the index. The determination of which hash bucket the row belongs to, and hence which AMP the row is to be stored on, is made on the value of the unique secondary index. Because the row hash for a secondary index is usually different from the row hash for the primary index on the same table, unique secondary indexes are generally stored on a different AMP than the row they point to.

Nonunique secondary indexes are not hashed and are always stored on the same AMP as the rows they point to.

Secondary indexes for a table can be stored either in hash order or in value order, depending on their intended use.

Further Information

See Chapter 10: “Secondary Indexes” for additional usage information and design tips about secondary indexes.

Join Indexes

Purpose

Join indexes have several uses.

- Define a prejoin table on frequently joined columns (with optional aggregation) without denormalizing the database.
- Create a full or partial replication of a base table with a primary index on a foreign key column table to facilitate joins of very large tables by hashing their rows to the same AMP as the large table.
- Define a summary table without denormalizing the database.

You can define a join index on one or several tables. Single-table join index functionality is an extension of the original intent of join indexes, hence the confusing adjective “join” used to describe single-table join indexes.

Join indexes are not indexes in the usual sense of the word. They are base tables that cannot be accessed directly with queries.

The Optimizer includes a multitable join index in a query plan in the following situations.

- The index covers all or part of a join query, thus eliminating the need to redistribute rows to make the join.
- A query requests that one or more columns be aggregated and an appropriate aggregate join index exists, thus eliminating the need to perform the aggregate computation.

The Optimizer includes a join index in a query plan in the following situations.

- The index covers all or part of a join query, thus eliminating the need to redistribute rows to make the join. A join index can be used for a partial cover only if it has been defined using any of the following components.
  - The UPI of the underlying base table
  - The keyword ROWID[2]
  - The NUPI of the underlying base table and the keyword ROWID

2 You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement. See "CREATE JOIN INDEX" in *Teradata RDBMS SQL Reference, Volume 4*.

  - The NUPI of the underlying base table and the USI of the underlying base table

    It is preferable to specify NUPI and ROWID over NUPI and USI.

    In this case, the Optimizer uses the ROWID, base table UPI or NUPI, or base table USI to join the join index rows with their underlying base table rows to pick up the base table columns necessary to complete the cover.
- A query requests aggregation on a column set and an existing single-table aggregate join index covers the specified aggregation, thus eliminating the need to perform the aggregate computation.

Mandatory Use

Join indexes are never required and are rarely recommended for tables that are accessed frequently by OLTP applications. They are highly recommended for decision support applications because they often provide superior performance to large table joins and aggregate computations. See Chapter 11: “Join and Hash Indexes,” for details.

Restrictions

Excluding the primary index, you can only define 32 indexes on a table. These 32 indexes can be any combination of secondary, hash, and join indexes.

No more than 16 columns can be defined per joined table in a join index definition.

No more than 32 columns can be defined for a compressed join index, 16 each for the fixed and variable parts.

There is no limit on how many total columns can be defined in an uncompressed join index other than system restrictions on the amount of SQL text required to define them.

Although the Optimizer substitutes only one multitable join index per referenced table in a query, it also considers additional single-table join indexes for inclusion in the join plan after the optimal multitable join index has been substituted and evaluated for the plan.

Types of Join Index

There are four basic join index types defined over two orthogonal dimensions: single or multitable and simple or aggregate. These are better thought of as ways to group functionality than as different types of join index.

- Multitable simple
- Multitable aggregate
- Single-table simple
- Single-table aggregate

Join Index Storage

For the most part, join index storage is identical to standard base table storage except that join indexes can be compressed. Join index rows are hashed and partitioned on their primary index (which is always defined as nonunique). Join index tables can be indexed, and their indexes are stored just like nonunique primary and secondary indexes for any other base table.

The major difference in storage between join indexes and standard base tables is the manner in which the repeated field values of a join index are stored. Repeated field value storage is too complex to describe in this summary. See Chapter 11: "Join and Hash Indexes" under "Hash and Join Index Storage" on page 11-74 for details about how repeated field values are stored.

A join index can be stored either in hash-order or in value-order, depending on its intended use.

Further Information

See Chapter 11: "Join and Hash Indexes" for additional usage information and design tips about join indexes.

Hash Indexes

Purpose

Hash indexes are used for the same purposes as single-table join indexes. The principal difference between hash and single-table join indexes are listed in the following table.

Hash Index	Single-Table Join Index
Cannot have secondary indexes.	Can have secondary indexes.
Rows are partitioned on the hash of the primary index of its base table unless otherwise explicitly specified.	Rows are partitioned on an explicitly defined primary index.
Column list cannot contain aggregate functions.	Column list can contain aggregate functions.

Hash indexes create a full or partial replication of a base table with a primary index on a foreign key column table to facilitate joins of very large tables by hashing them to the same AMP.

You can define a hash index on one table only. With the exception of storing aggregates, the functionality of hash indexes is a superset to that of single-table join indexes.

Hash indexes are not indexes in the usual sense of the word. They are base tables that cannot be accessed directly by a query.

The Optimizer includes a hash index in a query plan in the following situations.

- The index covers all or part of a join query, thus eliminating the need to redistribute rows to make the join.

  In the case of partial query covers, the Optimizer uses certain implicitly defined elements in the hash index to join it with its underlying base table to pick up the base table columns necessary to complete the cover.
- A query requests that one or more columns be aggregated, thus eliminating the need to perform the aggregate computation.

Mandatory Use

Hash indexes are never required and are rarely recommended for tables that are accessed frequently by OLTP applications. They are highly recommended for decision support applications because they often provide superior performance for ad hoc queries. See Chapter 11: “Join and Hash Indexes,” for details.

Restrictions

Excluding the primary index, you can only define 32 indexes on a table. These 32 indexes can be any combination of secondary, hash, and join indexes.

Hash Index Storage

For the most part, hash index storage is identical to standard base table storage except that hash indexes can be compressed. Hash index rows are hashed and partitioned on their primary index (which is always defined as nonunique). Hash index tables can be indexed explicitly, and their indexes are stored just like nonunique primary indexes for any other base table. Unlike join indexes, hash index definitions do not permit you to specify secondary indexes.

The major difference in storage between hash indexes and standard base tables is the manner in which the repeated field values of a hash index are stored. Repeated field value storage is too complex to describe in this summary. See Chapter 11: “Join and Hash Indexes” under “Hash and Join Index Storage” on page 11-74 for details about how repeated field values are stored.

A hash index can be stored either in hash-order or in value-order, depending on its intended use.

Further Information

See Chapter 11: “Join and Hash Indexes” for additional usage information and design tips about hash indexes.

Reference Indexes

Introduction

A reference index is an internal structure that the system creates whenever a referential integrity constraint is defined between tables using a PRIMARY KEY or UNIQUE constraint on the parent table in the relationship and a REFERENCES constraint on a foreign key in the child table.[3]

The index row contains a count of the number of references in the child, or foreign key, table to the PRIMARY KEY or UNIQUE constraint in the parent table.

Apart from capacity planning issues, reference indexes have no user visibility.

Further Information

For information about reference index sizing issues, see “Reference Index Sizing Equation” on page 14-94.

3 Teradata does not create or maintain REFERENCE indexes for batch referential integrity relationships.

Indexing and Hashing

Introduction

Hashing and indexing are two different methods for facilitating data access.

Teradata does not use indexing. What we refer to as indexes are either row hash values or, in the case of join indexes, denormalized data tables that substitute for base tables in join queries.

How Indexing Works

In the case of indexing, a data value (or values, if the index is compound) from a row acts as an index key to that row. The index associates the index key with a relative row address that reports the location of the row on disk. Indexes are stored in order of their index key values and are said to be value-ordered.

Think of an index on a database table as you would think of the index to a book. A book index typically provides two data values: a character string indicating the indexed topic and a page number or numbers indicating the location of that information in the book.

If the topic is unique, then there is only one page number listed for it. If the topic is nonunique, then there might be several pages listed for it.

To find a piece of information in a book, you can either consult the index, find the topic you need, and then turn directly to the appropriate page (think of this as an indexed retrieval) or you can start reading the book from the beginning and stop only when you find the information you seek (this of this as a full-table scan).

How Hashing Works

In the case of hashing, an index key data value is transformed by a mathematical function called a hash function to produce an abstract value not related to the original data value in an obvious way. Hashed data is assigned to hash buckets, which are memory-resident routing structures that correspond in a 1:1 manner to the relationship a particular hash code or range of hash codes has with an AMP location.

Hashing is similar to indexing in that it associates an index key with a relative row address.

Hashing differs from indexing in the following ways:

- There is no obvious correspondence between a hash code and the location of the row it refers to, even though the hash code identifies the location of the row.
- Different data values can and often do produce identical hash codes. This results in what is called a hash collision, because any rows having the identical row hash value would be stored in the same disk location if there were no means for preventing that from happening.

Rows in Teradata tables are distributed, or horizontally partitioned, among the AMPs based on a hash code generated from the primary index, or primary hash key, for the table. Because the Teradata hash functions are formally mature and mathematically sound, rows with unique primary indexes are always distributed in a uniformly random fashion across the AMPs, even when there is a natural clustering of key values. This behavior both avoids nodal hot spots and minimizes the number of key comparisons required to perform join processing when two rows have the same row hash value.

All primary indexes are hashed for row partitioning, and while the stored value for the primary index is retained as data, it is the hashed transformation of that value that is used for distribution and primary index retrieval of the row.

Similarly, unique secondary indexes are also hashed and partitioned to their appropriate index subtables based on that hash value, where they are stored in RowID order. Rows stored in this way are said to be hash-ordered.
Non-primary index retrievals usually benefit from defining one or more secondary hash keys on a table.

To summarize, although Teradata uses the term *index* for these values, they are really hash keys, not indexes.

Tradeoffs Between Hashing and Indexing

Introduction

This topic compares the relative advantages and disadvantages of hashing and indexing and concludes that hashing is the better methodology for the Teradata massively parallel architecture.

Advantages of Hashing Over Indexing

Why use hashing instead of indexing? The following list of advantages explains why hashing is superior to indexing for the Teradata parallel architecture.

- For the majority of queries, hashing provides consistently better performance because rows are always distributed evenly across the AMPs, providing a consistently balanced workload across parallel units. Because Teradata does not use B-trees to organize row hash values, there is never a need to reorganize as the B-trees become unbalanced and unordered.
- Primary indexes are not stored in an index subtable. Instead, the row hash for an individual row is stored directly as part of the row data. This direct access to row data offers the following advantages over indexing of the same data:
  - Enhanced performance because of fewer I/Os
  - Decreased storage overhead because there are no primary index subtables
- Hashing is far better suited for the parallel database architecture pioneered by Teradata than is indexing. Because each table in a Teradata database is partitioned horizontally across the AMPs on a row-by-row basis, the storage and retrieval algorithms and indexes used by standard file systems for storing and retrieving data are not as efficient as the distributed parallel Teradata architecture.

  Because of its hashed data placement and parallel architecture, Teradata requires only a one-time decision on the primary index for each table: the system takes care of everything else.

  Compare this with other relational products where a DBA must perform tasks such as determining keys and locations and coding space allocation for each partition before even beginning to create a table. Then the DBA must embed the partitioning assignments into the CREATE TABLE statement and forever after monitor and maintain partition sizes and perform periodic table reorganizations.

- By carefully defining primary index columns on frequently used join constraints, you can force rows that are frequently joined to be co-located on the same AMP. This is critical to join performance, because rows must be on the same AMP if they are to be joined. By co-locating join-constrained rows, you avoid the need to redistribute the rows from one or more tables across the BYNET to the AMP where they will be joined with rows located on that AMP.

Disadvantages of Other Row Partitioning Methods

There are several problems with traditional data placement schemes, particularly in a VLDB environment.

- Data set partitioning guarantees hot spots and interconnect log jams.
  - This method requires the DBA to batch data as it arrives in the database and then manually control how it is partitioned. While it is possible to maintain a balanced data load with these scheme, it is *not* possible to maintain a balanced processing load.

    Table rows are received, batched, and partitioned serially. This means that the newest data is always clustered together. Because a high percentage of data warehouse processing involves comparing current data with historical data to detect trends, this results in the majority of users attempting to access the same, co-located, data simultaneously.
  - VLDB data warehouse environments are typically configured as massively parallel or clustered processing units. This requires some sort of network interconnect channel.

    There is no way to know whether rows from tables that are to be joined are co-located or not, and there is no way to ensure that they are. This means that table joins in a data set partitioning situation typically need to transmit vast quantities of data across their interconnect channels in order to join tables, severely reducing system throughput in the process.

- Pure range partitioning is not a useful method for most data warehouse applications. When data access is repeatedly based on the same constraints, then range partitioning is an excellent choice. Data can be partitioned into multiple sets based on the repeatedly used constraints.

  As the following list of potential problems indicates, there are more problems with pure range partitioning in a massively parallel data warehouse environment than there are solutions provided.

  - Business data is rarely well-balanced. Because of this, the distribution of the data must be carefully analyzed in order to determine how best to partition the data across network nodes.
  - Some way must be found to ensure that commonly joined table rows are co-located in order to minimize traffic on the interconnect channels.
  - Some method of balancing table data across multiple nodes must be determined.
  - Because the demographics of data always change as a function of time, these considerations must be revisited repeatedly to ensure continued smooth performance.

  Notice that all these issues require intensive intervention on the part of the DBA. Distribution and other demographic data must be collected and analyzed (and an inexpensive, yet reliable method of doing the collection and analysis must be found), algorithms must be discovered or developed and then tested, and every aspect of the data must be monitored continually.

  The Teradata solution to range partitioning is the partitioned primary index, which hashes table rows to the AMPs using the same row hash method that assigns nonpartitioned primary index rows, but adds the ability to further assign those rows to user-defined range partitions. See "Partitioned and Nonpartitioned Primary Indexes" on page 9-6 for further information about partitioned primary indexes.
- Random partitioning creates more problems than it solves.

  Random, or round-robin, partitioning is formally related to hash partitioning. Unlike hashing, which uses an algorithm with known partitioning properties to distribute table rows, random partitioning uses a random number generator to distribute rows. The resulting distribution is even, but unrepeatable. This method makes it impossible to know where a table row is stored, so it can never be accessed directly.

  Random partitioning also ensures that data must be redistributed for join and aggregate processing, resulting in suboptimal system performance.

- Schema partitioning almost always requires extensive redistribution of table rows for join and aggregate processing in a networked configuration.

  This method, which is a scheme to assign specific table groups (“schemas”) to specific physical processors or nodes, has proven useful for optimizing the retrieval performance of specific tables in small, single node systems.

  When applied to a multiple node parallel environment, however, its deficiencies are readily apparent.
  - Most join cases require all rows to be redistributed across the interconnect for each query, reducing system throughput in the process.
  - Node balancing problems are always made worse when schema partitioning is used for anything other than very small tables.

Advantages of Indexing Over Hashing

There are two categories of query that often perform better with value-ordered indexing,

- Range queries
- Retrievals having selection criteria that involve only part of a multicolumn hash key

First consider range queries. The following SQL SELECT statement requests all rows from the parts table with a part number ranging between 3517 and 3713, inclusive. The primary key for the table is part_number.

```
SELECT part_number, part_description
FROM parts_table
WHERE part_number BETWEEN 3517 AND 3713;
```

Depending on the configuration and the database manager, such a query might be more performant if executed against a table that uses a value-ordered primary index key rather than a hash-ordered hash key.

To enhance the ability of hash keys to retrieve range query data, Teradata provides two mechanisms: the capability to store index rows in the order of their index values and the capability to store base table rows within partitions, including range partitions, after they have been hashed to an AMP. See “CREATE INDEX,” “CREATE JOIN INDEX,” and “CREATE TABLE” in the chapter “SQL Data Definition Language Statement Syntax” in *Teradata RDBMS SQL Reference, Volume 4* and "CASE_N" and "RANGE_N" in the chapter "Arithmetic Operators and Functions/Trigonometric/Hyperbolic Functions" in *Teradata RDBMS SQL Reference, Volume 5*.

The only way to avoid the partial hash key issue is to have a thorough understanding of your applications and data demographics before you define your indexes. In other words, if your applications will use partial key selection criteria, either define the index on those frequently retrieved columns or define a secondary index on them. The optimal solution is very dependent on the individual circumstances and there is no single correct way to design for this particular situation.

Teradata Hashing Algorithm

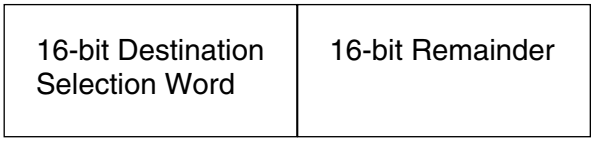
Introduction

The Teradata hashing algorithms are proprietary mathematical functions that transform an input data value of any length into a 32-bit value referred to as a row hash. You can select from among several hashing algorithms to use with your system, depending on what character sets your site uses. The only difference among the available hashing algorithms is how they are optimized to handle different character sets such as those used by double-byte Asian languages and European languages that use various diacritical marks. You *cannot* define multiple hashing algorithms for the same system.

Destination Selection Word

The first 16 bits of the row hash are a Destination Selection Word, which is used to define the hash bucket for the hashed row. The remaining 16 bits are a remainder from the operation of the hash function on the original input value.

The values input to the hashing algorithm are the column set values that constitute an index on the table in question.



FF07D301

A 32-bit row hash value provides 4.2 billion possible values, which reduces hash collisions to a level that eliminates their impact on retrieval performance for all intents and purposes.

Because the number of possible index column values for a given domain can easily exceed that number of values, further information is required to make each row uniquely identifiable. This situation also occurs for tables hashed on a nonunique primary index, where different rows can have the same primary index value, and so the same row hash value.

Uniqueness Value

Teradata uses an additional 32-bit system-generated Uniqueness Value to ensure the uniqueness of any RowID (see "Hash Collisions" on page 8-34).

Base table row distribution is always determined by the row hash for the primary index on the table, while index subtable values are hashed on the value of the secondary index they represent.

Retrieval operations use the primary index value for a row plus any existing secondary indexes that the Optimizer determines would reduce the cost of the activities required to retrieve the row.

Note that primary index operations use only the row hash value, while secondary index operations use the entire RowID.

Hash Collisions

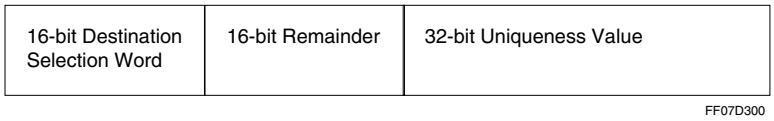
Introduction

The principal problem faced by hashing is hash collisions: situations in which the row hash value for different rows is identical, making it difficult for a system to discriminate among the hash synonyms when one unique row is requested for retrieval from a set of hash synonyms.

Minimizing Hash Collisions

To minimize this problem, Teradata systems define 65 536 hash buckets. The Generator, a PE software component that builds AMP directives from the Optimizer query plan, also adds a system-generated 32-bit Uniqueness Value to the row hash. The resulting 64-bit value is called the RowID, a value that uniquely identifies each row in a system, making a scan of a hash bucket to retrieve a particular row among several having the same row hash a trivial task.

The following graphic illustrates the structure of a Teradata RowID.



Hash Maps

Introduction

A hash map is a mechanism for determining which AMP a row is stored on. Each cell in the map array corresponds to a hash bucket, and each hash bucket is assigned to an AMP.

A hash map defines the correspondence between a hash bucket and an AMP. There are 65 536 buckets in the map, divided as evenly as possible among the AMPs on the system.

Hash map entries are maintained by the BYNET.

Hash Map Consistency Across Configurations

Hash maps are consistent across configurations. Any two systems having the same number of AMPs also have identical primary hash maps. Fallback hash maps can differ somewhat from system to system depending on how their AMPs are clustered.

The Problem of Skew

Each node in a Teradata system has a set of four hash maps, each performing a different function. All Teradata hash maps have 65 536 entries: one for each hash bucket. This number was carefully selected to balance the requirement that the hash maps be memory-resident with the need to have enough buckets to avoid the skewing problems seen when an uneven number of buckets are assigned to each AMP. With 65 536 hash buckets, skew is effectively eliminated even for the very largest tables.

As an example of how a relatively small number of hash buckets quickly leads to significant skew, consider the following case. Suppose there were an imaginary Teradata system that specified only 3 643 hash buckets rather than 65 536 hash buckets. Now suppose this was a large system having 200 AMPs. The result of having so few hash buckets is that some AMPs have 18 buckets and others have 19. This might seem like a small difference, but a quick computation tells a different story. The distribution of hash buckets maintained by this system guarantees a 5.26 percent skew of rows across the AMPs as indicated by the following calculation.

$$\left(\frac{1}{19}\right)(100) = 5.26 \text{ percent skew}$$

The following table shows how the number of hash buckets per AMP varies with the number of AMPs in the configuration. Note that configurations with a number of AMPs that does not divide evenly into 65 536 spread the remainder buckets among the AMPs as evenly as possible.

Number of AMPs in Configuration	Number of Hash Buckets/AMP	Number of AMPs With This Many Hash Buckets	Percent Skew
4	16 384	4	0.0000
5	13 107	4	0.0076
	13 108	1	
11	5 957	2	0.0170
	5 958	9	
60	1 092	11	0.0900
	1 093	4	

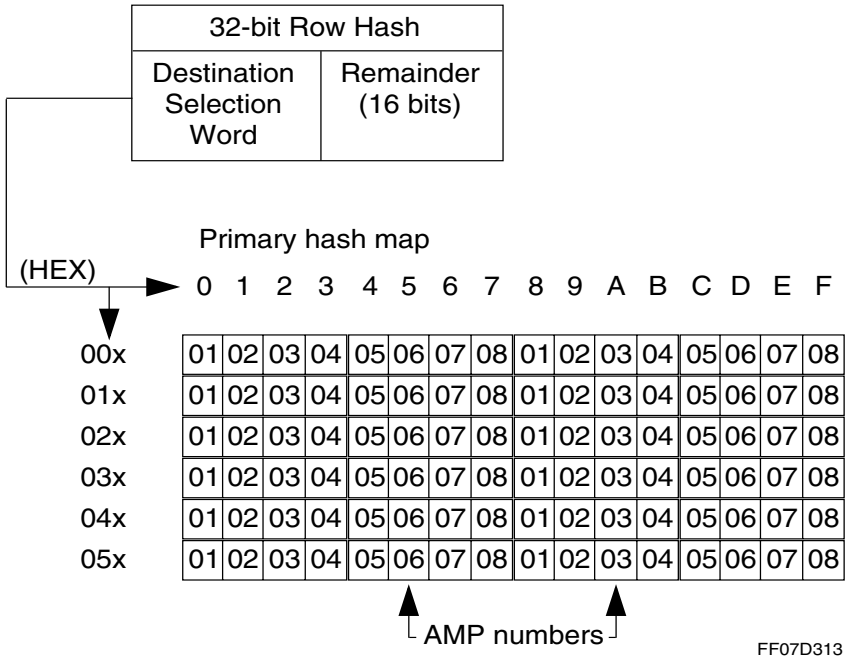
Definitions for the Four Hash Maps

The four hash maps maintained on each node in a Teradata system are defined as follows:

- The current configuration primary map

  This map maintains the hash bucket-AMP correspondences for the primary copies of all rows on the system.

  The following graphic is a logical diagram of the primary hash map. Physical hash maps do not look like this.



- The current configuration fallback map

  This map maintains the hash bucket-to-AMP correspondences for the fallback copies of all rows from tables defined with fallback. The fallback hash map is not identical to the current configuration primary hash map.
- The reconfiguration primary map

  This map maintains the map that indicates which primary rows migrate to which AMPs during a reconfiguration.
- The reconfiguration fallback map

  This map maintains the map that indicates which fallback row copies migrate to which AMPs during a reconfiguration.

Hash Map Lookup

When a row is hashed, the resulting Destination Selection Word points to a cell entry in a hash map. This entry defines the AMP to which the row hashes.

While the BYNET uses only the Destination Selection Word to distribute a row to its assigned AMP, the receiving AMP uses the entire row hash value to determine the logical disk storage location for the row.

Hash-Based Table Partitioning

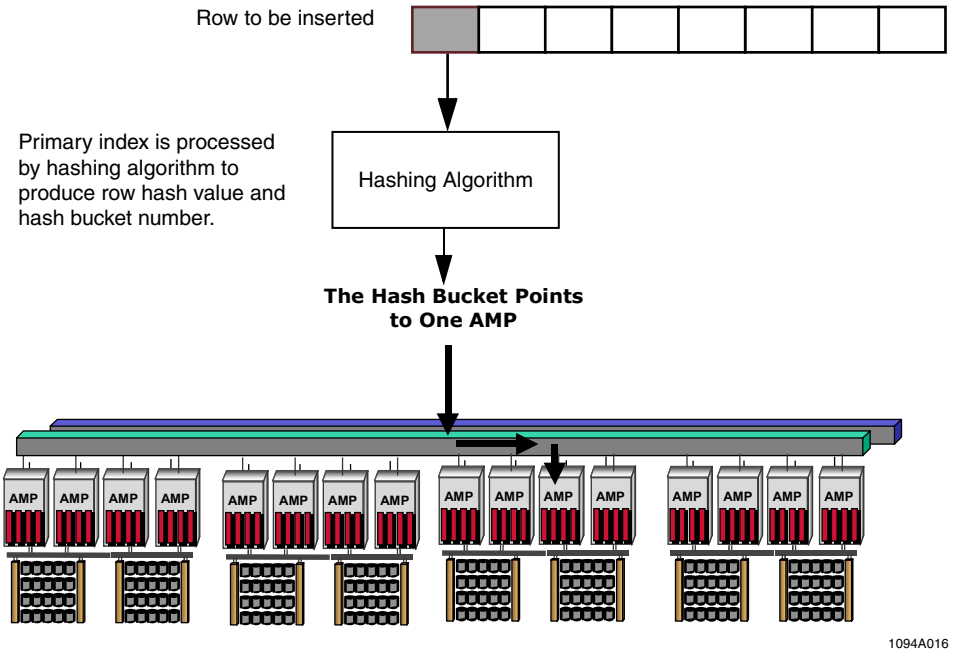
Introduction

All Teradata tables are partitioned horizontally by default. This means that the individual rows of each Teradata table are assigned to different AMPs in order to facilitate parallel processing.

Teradata uses the computed row hash value for each row to determine which AMP it should be assigned to (see "Teradata Hashing Algorithm" on page 8-32). If the primary index for a table is partitioned, then further processing is required to compute which partition it belongs to on its assigned AMP (see "Partitioned Row Allocation" on page 8-40).

Hash Partitioned Row Allocation

The process for assigning a particular row to an AMP is depicted in the following graphic. The logical process involved in this assignment is described in the table following the graphic.



The logical process involved in row assignment for a table insert operation is roughly the following:

<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>1</td><td>The INSERT statement passes its syntactical, lexical, and security constraints.</td></tr>
<tr><td>2</td><td>A row header is prepended to each row that includes information about whether the table into which it is to be inserted has an NPPI or PPI.</td></tr>
<tr><td>3</td><td>The primary index for each row is submitted to the hashing algorithm for processing.</td></tr>
<tr><td>4</td><td>The hashing algorithm produces a row hash value consisting of the Destination Selection Word (DSW) and Uniqueness Value.</td></tr>
<tr><td>5</td><td>The value of the DSW is checked against the hash map and the destination AMP for the row is determined.</td></tr>
<tr><td>6</td><td>The Dispatcher sends the row across the BYNET to its destination AMP.</td></tr>
<tr><td>7</td><td>The file system examines the flag byte in the row header (see “Base Table Row Format” on page 14-4) to determine whether the PI for the row is an NPPI or a PPI.
<table>
<tr><th>IF the primary index is this type ...</th><th>THEN AMP software ...</th></tr>
<tr><td>NPPI</td><td>assigns the row to the appropriate data block and stores it there in hash value uniqueness order.
A determination is made whether each row is uniquely constrained.
<table>
<tr><th>IF the table is of this type ...</th><th>THEN each row to be inserted is ...</th></tr>
<tr><td>SET</td><td>checked for uniqueness.
If the row is not unique, then it is not processed.</td></tr>
<tr><td>MULTISET</td><td>processed.</td></tr>
</table>
This ends the process.</td></tr>
<tr><td>PPI</td><td>uses the partitioning expression for the PPI to calculate the partition number for the row.
The partition number determines the AMP partition to which the row is assigned.
Go to Stage 8.</td></tr>
</table></td></tr>
<tr><td>8</td><td>The file system assigns the row to the appropriate partition.
See “Partitioned Row Allocation” on page 8-40 for an illustration of Stages 7 and 8 for PPIs.</td></tr>
</table>

<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>9</td><td>A determination is made whether each row is uniquely constrained.

<table>
<tr><th>IF the table is of this type ...</th><th>THEN each row to be inserted is ...</th></tr>
<tr><td>SET</td><td>checked for uniqueness.
If the row is not unique, it is not processed.</td></tr>
<tr><td>MULTISET</td><td>processed.</td></tr>
</table>

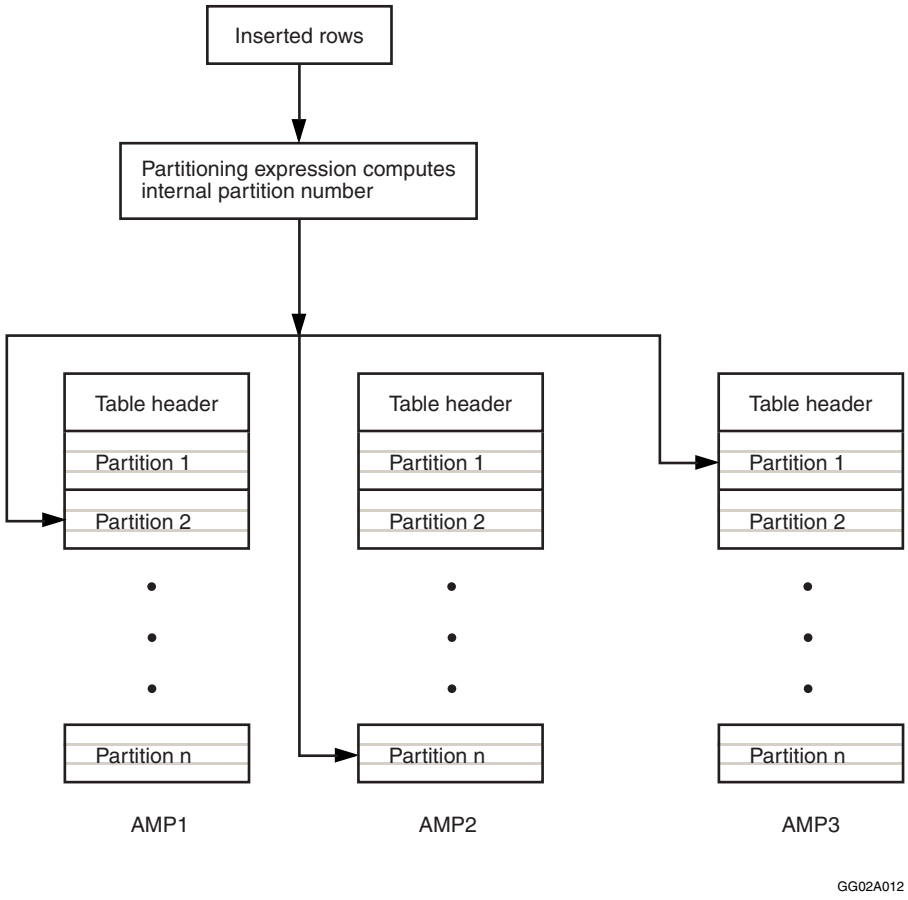
The row is then stored in hash value uniqueness order.</td></tr>
</table>

Partitioned Row Allocation

If the primary index for your table is a PPI, then the following events occur when a row is processed for storage.

Stage	Process
1	The row is hashed to its AMP (see "Hash-Based Table Partitioning" on page 8-38).
2	Its partition assignment is calculated by AMP software stored in the table header using the partitioning expression defined for the table.
3	The row is inserted into the proper partition by the file system in hash value uniqueness order.

The following diagram provides a high-level picture of the process at the point where the rows have arrived at their assigned AMPs.



Hash-Related Functions

Introduction

Teradata SQL provides several functions that you can use to analyze the hashing characteristics of your existing indexes and candidate indexes. These functions are documented fully in *Teradata RDBMS SQL Reference, Volume 5*, in the chapter "Hash-Related Functions."

The following table summarizes the hash-related functions. Note that they can be nested.

This function …	Returns the …
HASHAMP	number of the AMP associated with an expression.
HASHBAKAMP	number of the backup AMP associated with an expression.
HASHBUCKET	number of the hash bucket associated with a hash row expression.
HASHROW	hexadecimal row hash value for an expression.

Using HASHROW

HASHROW is especially useful for evaluating the following things.

- Row distribution efficiency of proposed primary index fields.
- Relative efficiency of various data types for proposed or existing primary index columns.

As shown in the following example set, you can use the output from HASHROW to compute the average number of rows that cause hash collisions.

Example 1

This example uses the HASHROW function on a unique primary index.

Assume that the EmpNo column is the UPI for the Employee table. To find the number of hash collisions generated by the values in EmpNo, type the following statement.

```
SELECT(COUNT (*) (FLOAT))/(COUNT(DISTINCT HASHROW(EmpNo)))
FROM Employee;
```

If there are no hash collisions, the result ratio is close to 1. The larger the returned value, the more hash collisions occur and the less efficient the analyzed column expression analyzed is as a primary index.

If a proposed primary index contains nonunique fields, perform the following check.

Step	Action
1	Submit the statement in the example against the proposed primary index column expression and save the result.
2	Resubmit the statement against a column expression where each index value is unique and save the result.
3	Compare the result step 1 against the result of step 2. The comparison should indicate clearly which index generates fewer hash collisions.

Example 2

The following example uses the HASHROW function against a the same primary index column analyzed in "Example 1" on page 8-42, but retyped as DECIMAL(2,2).

Step	Action

1	Determine how many hash collisions occur with the existing data type by submitting the same query used in "Example 1" on page 8-42.
	`SELECT (COUNT (*) (FLOAT)) / COUNT (DISTINCT HASHROW (EmpNo))`
	`FROM Employee;`

2	Determine whether hash collisions would be reduced by changing the data type to DECIMAL(4,3) by submitting the following two statements (both are required because of the restrictions on the DISTINCT option):
	`SELECT COUNT (DISTINCT EmpNo(DECIMAL(4,3)))`
	`FROM Employee;`
	
	`SELECT COUNT (DISTINCT HASHROW (EmpNo(DECIMAL(4,3)))`
	`FROM Employee;`

3	Divide the numbers returned by the two statements to calculate the average number of hash collisions that occur with the new data type.
4	Compare this average with the result of the first statement to determine if the change reduces the collision count.

Example 3

When used together, HASHAMP, HASHBUCKET, and HASHROW indicate the distribution of primary rows among the AMPs. For example, the following query shows the row distribution of the Employee table based on the primary index EmpNo, from which you can quickly see whether distribution is even or uneven:

```
SELECT HASHAMP (HASHBUCKET (HASHROW(EmpNo))), COUNT (*)
FROM Employee
GROUP BY 1
ORDER BY 1;
```

IF the distribution is ...	THEN ...
uneven	• Change the primary index to a column or columns that uniquely identifies each row. • Add a column to the primary index to force uniqueness. • Change the data type of one or more index columns (see "Hashing and Data Types" on page 8-50).
even	create your production table defining the primary index based on the most performant columns. For example, either of the following examples where preliminary performance results are based on trials on a test system. • A primary index based on a column that provides the best performance. • A primary index based on the same column, but having a particular data type provides the best performance (see "Hashing and Data Types" on page 8-50).

Hashed Row Access

Introduction

This topic describes indexed access to rows on disk. The data and file structures used for this purpose are the following.

- Master index
- Cylinder index
- Data block

Master Index

The master index is a memory-resident file system data structure that contains pointers to every cylinder index on a given AMP in rows referred to as Cylinder Index Descriptors, or CIDs. When a cylinder does not contain data, it is not listed in the master index.

Each master index entry contains the following data sorted on Table ID and row hash:

- Lowest Table ID in the cylinder
- Lowest RowID value on the cylinder (associated with the lowest Table ID)
- Highest Table ID in the cylinder
- Highest row hash (*not* RowID) value on the cylinder (associated with the highest Table ID)
- Cylinder number

The file system uses the Table ID and RowID (or row hash value) for the row being retrieved to scan the master index for the desired cylinder number.

As an example of how a Table ID and row hash are used to scan the master index, consider the following query.

```
SELECT *
FROM Employee
WHERE EmpNo = 7225;
```

For this example, assume that the decimal row hash value for 7225, the primary index value for the row in question, is 266 and the Table ID for the Employee table is 100.

Consider the following set of contrived master index entries.

Lowest Table ID	Starting RowID	Highest Table ID	Highest Row Hash Value	Cylinder Number
…	…			…
…	…			…
…	…			…
078	58234 + 0001	081	58332	204
100	00017 + 0001	100	00073	037
100	00102 + 0001	100	00687	169
100	00785 + 0001	100	00991	777
100	01087 + 0001	100	01116	802
100	03662 + 0003	100	03712	117
100	04123 + 0001	100	04255	888
100	05974 + 0001	100	06785	753
100	07353 + 0001	100	07834	477
123	00543 + 0001	123	00786	529
…	…			…
…	…			…
…	…			…

The process for finding the required cylinder block is as follows.

Stage	Process
1	The file system scans the master index for Table ID 100.
2	When Table ID 100 is found, the file system scans the Starting RowID column until it finds a row hash value that exceeds 266.
3	The previous cylinder number, 169, is the cylinder on which the data block containing the row will be found.
4	The file system creates a new search key on Table ID, rowID, and cylinder index. In this case, the search key is 100, 266, 169.
5	The file system locates cylinder index 169.

Cylinder Index

The cylinder index is a disk-resident file system file structure that contains rows referred to as Data Block Descriptors, or DBDs. Each disk cylinder has its own cylinder index. The structure contains pointers to all the data blocks and free sectors on the cylinder it represents.

If AMP memory is large enough, the cylinder index is cached to ensure that a primary index lookup never takes more than one I/O.

Each cylinder index entry contains the following data sorted on Table ID and row hash:

- Table ID for the table stored in the block (only one table is stored in a data block)
- Lowest RowID value in the block
- Highest row hash value in the block
- Sector number for that block
- Number of sectors in the block

The file system uses the Table ID and RowID value for the row being retrieved to scan the cylinder index for the number of the sector on which the desired block is stored.

Following through with the query from "Master Index" on page 8-45, consider the following set of fictional cylinder index entries for cylinder 169. Note that the + *n* values for the RowIDs indicate the Uniqueness Value.

Table ID	Starting RowID	Highest Row Hash Value	Beginning Sector Number	Sector Count
100	00058 + 0001	00163	1010	9
100	00184 + 0001	00332	0789	6
100	00363 + 0001	00994	0530	4
100	01035 + 0001	01114	0056	5
100	01185 + 0002	01190	1138	8
100	01221 + 0001	02072	0117	16
100	02123 + 0001	02365	0888	7
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…

The process for finding the required data block is as follows.

Stage	Process
1	The file system scans the cylinder index of disk cylinder 169 for Table ID 100.
2	When Table ID 100 is found, the file system scans the Starting Row Hash column until it finds a row hash value that exceeds 266.
3	The previous beginning sector number, 0789, is the sector at which the data block that contains the desired row begins and runs for 6 sectors.
4	The file system creates a new search key on row hash value and sector number and scans the data block for the desired row.

Data Block

The data block is a disk-resident file system file structure that contains one or more rows from the same table. Row storage is atomic: either the entire row fits within a data block or none of it does (in which case it is stored in a different block). In other words, rows cannot be split between blocks.

The Teradata blocking strategy ensures the following things:

- No more than 511 bytes ever remain unused within a block
- No pointer chains are required to locate logically adjacent sectors in other blocks
- No reorganization because of block split fragmentation is ever required.

The data block is the physical I/O unit for the Teradata file system. Block sizes range between 512 and 130 560 bytes (1 to 255 sectors).

You can set the default maximum data block size in several ways.

- Set a system-wide default using the DBSControl utility.
- Set or change the block size at the individual table level by establishing a value using the DATABLOCKSIZE parameter of the CREATE TABLE and ALTER TABLE statements, respectively.

Blocks within an individual table can vary in size, and the file system adjusts their sizes dynamically as required. See *Teradata RDBMS SQL Reference, Volume 4*, for syntax and usage of the ALTER TABLE and CREATE TABLE statements.

Although the rows within a data block are not stored in any particular physical order, a structure called the Row Reference Array, which contains 2-byte pointers to the first word in each row, stores its data in ascending order of RowID.

The logical storage sequence for rows in an Employee file might look something like the following table.

RowID		User Data		
Row Hash	Uniqueness ID	EmpNo	LastName	FirstName
7332	0001	6149	Smith	John
7332	0002	6171	Chang	Wei-hee
7332	0003	7777	Misawa	Mitsuharu
7333	0001	6333	Messiaen	Jean-Luc
7334	0001	5010	Wanz	Hans
7334	0002	6214	Kulthum	Muhammad
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…

Hashing and Data Types

Introduction

Though it might not be apparent at first glance, the data type of a column affects the way it is hashed, and this property can be an important consideration for database design.

Hashing On Bit Patterns, Not Values

The Teradata hashing algorithm operates on the bit patterns of the data on which it operates: the internal representation of the data, not its external value.

For example, a numeric value stored as a decimal number produces a different hash code than the same numeric value stored as an integer or floating point number (see "Data Type Considerations" on page 14-75). Additionally, a number typed as an integer requires only 4 bytes of storage, while its decimal representation might take as many as 8 bytes, depending on the size of the number.

Considerations for Indexing

You should evaluate the following considerations when assigning data types to index columns:

- Suitability of the type family for indexing
  - Numeric types are best because they are compact and often tend toward uniqueness in business data.
  - Character and byte types are poor because they tend not to be compact and because character and byte data naturally tends to be nonunique.
- Hash collisions

  "Example 2" on page 8-43 and "Example 3" on page 8-44 under the topic "Hash-Related Functions" address this issue.
- Storage space

  Greater space requirements for a column translate to fewer data rows per block, which results in slower full-table scans.
- Join and predicate conversions

  Any time tables are joined on a column, whether indexed or not, the columns must have the same data type in both tables. If the types do not agree, one must be converted, which results in a performance hit. The same is true for columnar comparisons made in WHERE clause predicates.

Teradata Index Comparisons

The following table summarizes the similarities and differences among Teradata primary, secondary, join, and hash indexes.

Attribute	Primary Index	Secondary Index	Join Index	Hash Index
Must be defined	Yes	No		
Maximum per table	1	32[a]		
Minimum per table	1	0		
Maximum number of columns	64		64[b]	
Unique type supported	Yes		No	
Nonunique type supported	Yes		No	
Affects base table row partitioning	Yes	No		
Affects index subtable row partitioning	Not applicable[c]	Unique Yes	Not applicable[d]	
		Nonunique No		
Can be partitioned within an AMP	Yes	No		
Can be created and dropped dynamically	No	Yes		
Enhances access performance	Yes			
Supports multiple data types	Yes			
Stored in a separate file structure	No	Yes		
Processing overhead	No	Yes		

a. Combined number of secondary, hash, and join indexes defined on a table cannot exceed 32.
b. You can specify 64 columns per referenced base table in a multitable join index definition when you do not also specify compression for the index.
c. There are no primary index subtables. Primary indexes are stored in-line as row data.
d. Join and hash index rows are distributed on their primary indexes, just like base table rows.

Table Access Summary

The following table characterizes nine access methods available to the Optimizer for accessing tables. Definitions for the access method abbreviations are provided in the following table.

Access Method Abbreviation	Definition
UPI	Unique Primary Index
UPPI	Unique Partitioned Primary Index
NUPI	Nonunique Primary Index
NUPPI	Nonunique Partitioned Primary Index
USI	Unique Secondary Index
NUSI	Nonunique Secondary Index
JI	Join Index
HI	Hash Index
FTS	Full-Table Scan

For many table join operations, a join index is often a more efficient means for accessing data. See "Join Indexes" on page 8-20 and Chapter 11: "Join and Hash Indexes."

Access Method	Relative Efficiency	Number of AMPs Used	Number of Rows Accessed	Spool Space Required?	
UPI	Very efficient.	1	1	No	
UPPI	Very efficient.	1	1	No	
NUPI NUPPI	Efficient when selectivity is high and skew is low.	1	Multiple	Depends	
				IF a query returns ...	**THEN ...**
				a single response	no
				more rows than can be handled by a single response	yes
USI	Very efficient.	2	1	No	

<table>
<tr><th>Access Method</th><th>Relative Efficiency</th><th colspan="2">Number of AMPs Used</th><th colspan="2">Number of Rows Accessed</th><th colspan="2">Spool Space Required?</th></tr>
<tr><td>NUSI</td><td>Efficient when the number of rows accessed is less than the number of data blocks in the table.</td><td colspan="2">All</td><td colspan="2">Multiple</td><td colspan="2">Yes</td></tr>
<tr><td rowspan="5">JI</td><td rowspan="5">Very efficient.</td><td>IF the JI is ...</td><td>THEN the number of AMPs used is ...</td><td>IF the JI is ...</td><td>THEN the number of rows accessed is ...</td><td colspan="2">Depends</td></tr>
<tr><td>accessed by its primary index</td><td>1</td><td>accessed by its primary index</td><td>multiple</td><td>IF the JI ...</td><td>THEN ...</td></tr>
<tr><td>not accessed by its primary index</td><td>all</td><td>not accessed by its primary index</td><td>all (FTS)</td><td>covers the query</td><td>No</td></tr>
<tr><td colspan="2"></td><td colspan="2"></td><td>partially covers the query and its definition specifies either ROWID[a] or the UPI for the base table</td><td>Yes</td></tr>
<tr><td colspan="4"></td></tr>
<tr><td rowspan="4">HI</td><td rowspan="4">Very efficient.</td><td colspan="2" rowspan="4">Depends on how the index is partitioned.</td><td colspan="2" rowspan="4">Multiple</td><td colspan="2">Depends</td></tr>
<tr><td>IF the HI ...</td><td>THEN ...</td></tr>
<tr><td>covers the query</td><td>No</td></tr>
<tr><td>partially covers the query</td><td>Yes</td></tr>
<tr><td>FTS</td><td>Efficient because each row is touched only once.</td><td colspan="2">All</td><td colspan="2">All</td><td colspan="2">Yes
Can require spool as large as the table being scanned.</td></tr>
</table>

a. You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement. See "CREATE JOIN INDEX" in *Teradata RDBMS SQL Reference: Volume 4*.

Index Selection Summary

The following list of factors illustrates how highly multidetermined the selection of Teradata indexes is. These factors are examined in the chapters on individual types of indexes that follow this one.

- Nonspecific Factors
  - Degree of normalization of the database
  - How the Optimizer might use the index
  - Table type indexed
    - Major entity
    - Minor entity
    - Subentity
- Space Utilization Factors
  - How much space does the index occupy?
  - Type of data protection specified
- Demographic Factors
  - Cardinality of the table
  - Number of distinct column values
  - Maximum rows per value
  - Columns most frequently used to access table rows
  - Are rows most commonly accessed by values or by a join?
  - Degree of skew of column values
- Application Factors
  - In which application environment are rows most commonly accessed
    - Decision support
    - OLTP
    - Ad hoc
    - Range queries
    - Batch reporting
    - Batch maintenance
- Transaction Factors
  - How are transactions written?
  - How are transactions parcelled?
  - What levels and types of locking does a transaction require?
  - How long does the transaction hold locks?
- DML Factors
  - Number of DELETEs and when they occur
  - Number of INSERTs and when they occur
  - Number of UPDATEs and when they occur

Chapter 9:

Primary Indexes

This chapter describes the Teradata primary indexes over two orthogonal dimensions: unique-nonunique and partitioned-nonpartitioned.

The process of selecting a primary index is given heavy emphasis. Other topics describe primary index access to rows, the various performance considerations that pertain to primary indexes, special considerations for nonunique primary indexes relating to duplicate checking, and space utilization parameters for primary indexes.

Primary Indexes

Introduction

This topic describes the basic properties of primary indexes.

A primary index is required for all Teradata tables. If you do not assign a primary index explicitly when you create a table, then the system assigns one automatically according to the rules described in "Primary Index Default Definitions" on page 9-9. Use the CREATE TABLE statement to create primary indexes (see "CREATE TABLE" in the chapter "Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*.

Data accessed using a primary index is always a one-AMP operation because a row and its primary index are stored together in the same structure (see "Base Table Row Format" on page 14-4). This is true whether the primary index is unique or nonunique, and whether it is partitioned or nonpartitioned.

Purpose of the Primary Index

The primary index has four purposes:

- To define the distribution of the rows to the AMPs.

  Teradata distributes table rows across the AMPs on the hash of their primary index value (see "Teradata Hashing Algorithm" on page 8-32). The determination of which hash bucket, and hence which AMP the row is to be stored on, is made solely on the value of the primary index.

  The choice of columns for the primary index affects how even this distribution is. An even distribution of rows to the AMPs is usually of critical importance in picking a primary index column set.
- To provide access to rows more efficiently than with full table scan.

  If the values for all the primary index columns are specified in a DML statement, single-AMP access can be made to the rows using that primary index value.

  With a partitioned primary index, faster access is also possible when all the values of the partitioning columns are specified or if there is a constraint on partitioning columns.

  Other retrievals might use a secondary index, a hash or join index, a full table scan, or a mix of several different index types.
- To provide for efficient joins.

  If there is an equality join constraint on the primary index of a table, it may be possible to do a direct join to the table (that is, rows of the table might not have to be redistributed, spooled, and sorted prior to the join).
- To provide a means for efficient aggregations.

  If the GROUP BY key is on the primary index of a table, it is often possible to perform a more efficient aggregation.

Limits

The following restrictions apply to primary indexes:

- Each table in a Teradata database has one and only one primary index.
- A primary index can be defined on anywhere from one to 64 columns.

Primary Index Dimensions

Primary indexes are defined across two orthogonal dimensions:

- Unique-nonunique (see "Unique and Nonunique Primary Indexes" on page 9-4).
- Partitioned-nonpartitioned (see "Partitioned and Nonpartitioned Primary Indexes" on page 9-6).

Unique and Nonunique Primary Indexes

Introduction

A primary index can be either unique or nonunique. The decision to define which of the two choices for a table depends on how its rows are most frequently accessed. This topic describes the differences between the two types as well as their relative advantages and disadvantages.

Unique Primary Indexes

A primary index is defined to be either unique or nonunique.

Major entities and subentities are typically assigned unique primary indexes. When a subentity has been defined in the logical model, it typically uses the same unique primary index as the major entity it is associated with. This ensures that related rows from the two tables always hash to the same AMP.

Consider the following Employee table derived from a major entity in the logical database design. The primary key is also defined as the unique primary index for the table.

Employee

PK **UPI** **Employee_ID**	**Employee_Name**	**Home_Address**
6149	Joe Smith	3 Homestead Way
6171	Wei-hee Chan	44 Fifth Avenue
7049	Yuka Maeda	1000 Chestnut Lane
…	…	…
…	…	…

Nonunique Primary Indexes

Minor entities are typically assigned nonunique primary indexes defined on the same column as the Major entity with which they are associated.

Consider the following Employee-Phone table derived from a minor entity in the logical database design.

The Employee-Phone table is related to the Employee table by its foreign key, Employee_ID, which is the primary key of the Employee table. Note that the primary index for this table is defined only on the Employee_ID attribute,

which is only half the primary key. This makes the primary index nonunique by definition.

The advantage gained by this is that both Employee and Employee-Phone rows have the same primary index and hash to the same AMP. This means that joins on these tables, which are a frequent occurrence, do not require redistribution of table rows across the BYNET.

You could have defined a NUPI on Phone_Number because there should not be many duplicate entries for that field (husband-wife-child groupings and roommates being likely examples), but you would lose the advantage of hashing related rows to the same AMPs.

Employee-Phone

PK	**PK**	
FK		
NUPI		
Employee_ID	**Phone_Number**	**Phone_Remarks**
6149	555-1234	Not after 20:00.
6171	555-5678	Any time.
7049	555-9012	Never call home number.
…	…	…
…	…	…

Partitioned and Nonpartitioned Primary Indexes

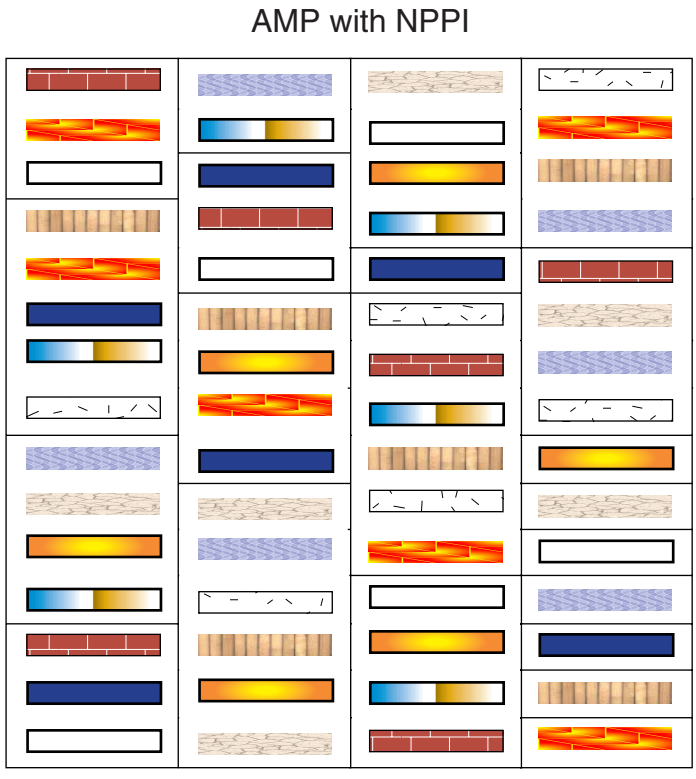
Introduction

A primary index can be either partitioned or nonpartitioned. The decision to define which of the two choices for a table depends on how its rows are most frequently accessed (see "Comparison of Partitioned and Nonpartitioned Primary Index Access for Some Typical Operations" on page 9-11 and "PPI Case Studies" on page 9-30). This topic describes the differences between the two types as well as their relative advantages and disadvantages.

Nonpartitioned Primary Indexes

A primary index is defined to be either partitioned or nonpartitioned.

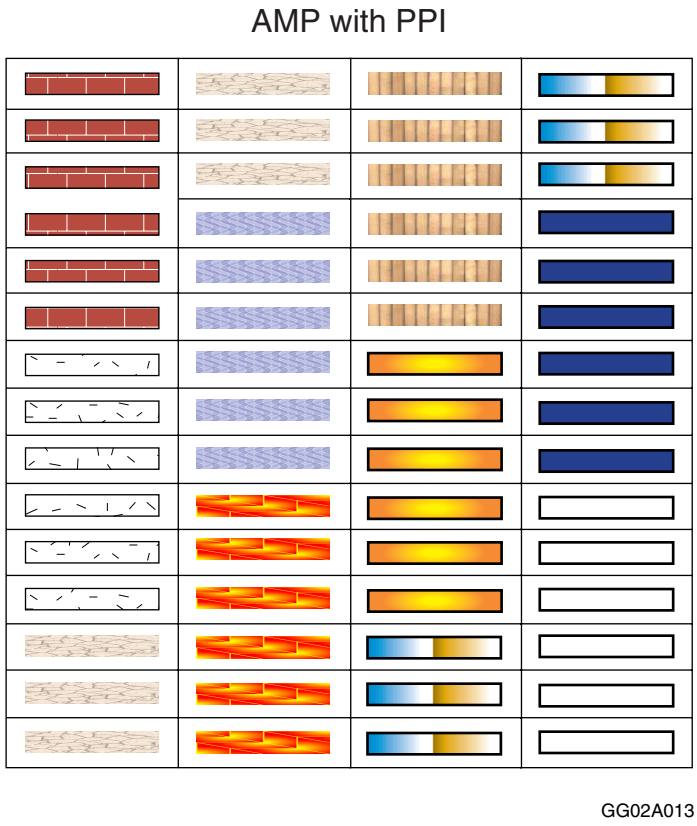
The nonpartitioned primary index (NPPI) is the standard Teradata primary index. When a table is created with an NPPI, its rows are hashed to the appropriate AMPs and stored there in row hash order, as illustrated by the following graphic.



Rows within the same box have the same row hash value.

Partitioned Primary Indexes

The partitioned primary index (PPI) is an extension to the NPPI. When a table is created with a PPI, its rows are hashed to the appropriate AMPs and then assigned to an appropriate partition based on the value of a partitioning expression defined by the user when the table was created or altered. Once assigned to a partition, the rows are stored in row hash order, as illustrated by the following graphic.



Rows are ordered first by their partition number, then by row hash value within each partition.

PPIs are designed to optimize range queries while also providing efficient primary index join strategies. Analyze your range query optimization needs carefully because there are performance tradeoffs between specific range query enhancements and possible decrements for primary index accesses and joins and aggregations on the primary index that occur as a function of the number of active partitions.

The following list of factors characterizes the general performance of PPIs with respect to NPPIs:

- PPI access by means of an equality constraint on *all* PPI and partitioning columns is as efficient as the same access made by means of an NPPI.
- PPI access on an equality constraint on the primary index columns and an equality or other constraint on the partitioning columns that limits access to a single partition, produces performance as efficient as that made with an NPPI.
- PPI access with a constraint on the partitioning columns can approach that of NPPI access depending on the extent of partition elimination (an indirect measure of the number of partitions that must be probed before the desired row is found) required.
- PPI access via an equality condition on the primary index that neither includes the partitioning columns nor makes a constraint on the partitioning columns might not be as efficient as access by means of an NPPI depending on the number of partitions that contain data.

See "PPI Case Studies" on page 9-30 for a set of design scenarios that evaluate the performance effects of several different partitioned primary indexes.

Memory Limitations Related to Partitioned Primary Indexes

You might encounter several different memory limitations when working with PPI tables. The following table addresses these limitations by documenting the relevant error message numbers and providing explanations or workarounds for them:

Message Number	Problem	Possible Cause	Remedy
3708	Table header size limit exceeded (see "Table Header Format" on page 14-7). The limit is 64Kbytes.	Too many columns in the table.	If possible, reduce the number of columns in the table.
		Too many distinct values compressed in the table.	If possible, reduce the number of compressed values.
		Partitioning expression is too complex.	Simplify the partitioning expression. If possible, use RANGE_N instead of CASE_N.

<table>
<tr><th>Message Number</th><th>Problem</th><th>Possible Cause</th><th>Remedy</th></tr>
<tr><td rowspan="2">3710</td><td rowspan="2">Parser memory size limit exceeded.

The limit is variable and is determined by the value of the MaxParseTreeSegs field in DBSControl.</td><td>Current values for DBSControl record fields do not allocate enough parser memory to process the request.</td><td>Change the values for the following DBSControl fields to the specified settings:

<table><tr><th>IF the problem occurs with PPI tables ...</th><th>THEN change the value for ...</th></tr><tr><td>without hash or join indexes</td><td>MaxParseTreeSegs to 2000.</td></tr><tr><td>with hash or join indexes</td><td>• MaxParseTreeSegs to 2000.

and

• OptMaxGlobalJI to 8.</td></tr></table>
You might find that your query workloads are such that these values need to be increased still further.

OptMaxGlobalJI is an internal DBSControl field. You must contact a Teradata support representative to change its value.</td></tr>
<tr><td>Partitioning expression is too complex.</td><td>Simplify the partitioning expression.

If possible, use RANGE_N instead of CASE_N.</td></tr>
<tr><td>3891</td><td>Check constraint text size limit exceeded.

The limit is 8 192 characters.

The check constraint text can be explicit or derived from the partitioning expression for the PPI.</td><td>Text for the partitioning expression in the PPI definition is too long.</td><td>Reduce the partitioning expression text to 8 172[a] or fewer characters.</td></tr>
</table>

a. 20 characters of the 8 192 character limit are allocated to the text string "<partitioning_expression> BETWEEN 1 AND 65535" by default.

Primary Index Default Definitions

A primary index must be specified when a new table is created. If you do not specify a column or set of columns as the primary index for the table, then CREATE TABLE specifies a primary index by default. When the selection of a primary index column set is problematic, consider using an identity column as the primary index for the table (see "Identity Columns" on page 14-13 and CREATE TABLE in *Teradata RDBMS SQL Reference, Volume 4*).

The process and criteria that CREATE TABLE uses for selecting a default primary index column are the following:

<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>1</td><td>
<table>
<tr><th>WHEN …</th><th>THEN Teradata selects the …</th></tr>
<tr><td>a primary key is defined but a primary index is not</td><td>primary key column set to be the primary index and defines it as a UPI.</td></tr>
</table>
</td></tr>
<tr><td>2</td><td>
<table>
<tr><th>WHEN …</th><th>THEN Teradata selects the …</th></tr>
<tr><td>neither a primary key nor a primary index is defined</td><td>first column having a UNIQUE constraint to be the primary index and defines it as a UPI.</td></tr>
</table>
</td></tr>
<tr><td>3</td><td>
<table>
<tr><th>WHEN …</th><th>THEN Teradata selects the …</th></tr>
<tr><td>no primary key, primary index, or uniquely constrained column is defined</td><td>first column defined for the table to be the primary index.
<table>
<tr><th>WHEN …</th><th>THEN Teradata defines the primary index as a …</th></tr>
<tr><td>the table has only one column and its kind is defined as SET</td><td>UPI.</td></tr>
<tr><td>anything else</td><td>NUPI.</td></tr>
</table>
</td></tr>
</table>
</td></tr>
</table>

Comparison of Partitioned and Nonpartitioned Primary Index Access for Some Typical Operations

The following table compares the access mechanisms for the two primary index types over a range of typical operations and conditions.

Operation	NPPI	PPI where PI includes all partitioning columns	PPI where PI does not include all partitioning columns
Selection via an equality (=, IN) constraint on the primary index columns	Single-AMP and direct access to the first data block of rows with the same hash value. Scan all rows with the same hash value checking primary index and residual conditions. Row-hash lock.	Single-AMP and direct access to the first data block of rows with the same hash value for only the partition specified. Scan all rows with the same hash value in this partition checking primary index and residual conditions. Row-hash lock.	Equality constraint on the partitioning columns: • Single-AMP and direct access to the first data block of rows with the same hash value for only the partition specified. • Scan all rows with the same hash value in the affected partition checking primary index and residual conditions. • Row-hash lock.
			Constraint on partitioning columns: • Single-AMP and direct access to the first data block of rows with the same hash value in each of the non-eliminated partitions. • Scan all rows with the same hash value in the affected partitions checking primary index and residual conditions. • Row-hash lock.
			Otherwise: • Single-AMP and direct access to the first data block of rows with same hash value in each active partition. • Scan all rows with the same hash value in each partition checking primary index and residual conditions. • Row-hash lock.

Operation	NPPI	PPI where PI includes all partitioning columns	PPI where PI does not include all partitioning columns
Selection without equality constraint on the primary index columns	All-AMP, full-file scan checking each row for any residual conditions. All-AMP table lock.	Equality constraints on all the partitioned columns: • All-AMP, but only scan a single partition on each AMP. • Scan all rows in the affected partition checking for any residual conditions. • All-AMP table lock.	
		Constraints on partitioned columns: • All-AMP, but only scan non-eliminated partitions. • Scan all rows in the affected partitions checking for any residual conditions. • All-AMP table lock.	
		Otherwise: • All-AMP, full-file scan checking each row for any residual conditions. • All-AMP table lock.	
Joins	There are a wide variety of joins. In many cases, an all-AMPs table lock is applied. However, there are some cases where row-hash locks are applied.	There are a wide variety of joins. These tables can be joined in the same manner as a table with a nonpartitioned primary index with the addition of same partition elimination possibilities being taken advantage of when possible.	
INSERT	Single-AMP and direct access to the first data block of rows with the same hash value. If SET table and no unique indexes, verify no duplicate row within this same hash value. If unique primary index, verify uniqueness within this same hash value.	Single-AMP and direct access to the first data block of rows with the same hash value in the specified partition. If SET table and no unique indexes, verify no duplicate row within this same hash value in this partition. If unique primary index, verify uniqueness within this same hash value in this partition.	Single-AMP and direct access to the first data block of rows with the same hash value in the specified partition. If SET table and no unique indexes, verify no duplicate row within this same hash value in this partition.

Operation	NPPI	PPI where PI includes all partitioning columns	PPI where PI does not include all partitioning columns
INSERT (continued)	Append to the end of the rows with the same hash value. Update secondary indexes (validate uniqueness for USIs). Row-hash lock.	Append to the end of the rows with the same hash value in this partition. Update secondary indexes (validate uniqueness for USIs). Row-hash lock.	Append to the end of the rows with the same hash value in this partition. Update secondary indexes (validate uniqueness for USIs). Row-hash lock.
INSERT … SELECT	Sort inserts as needed and merge into table. All-AMP table lock.		
DELETE	Same as selection.	Same as selection.	Same as selection.
UPDATE (Nonupsert form)	Same as selection unless index columns are updated. If index columns are updated, this is treated as an "unreasonable update." That is for an update of primary index columns, updated rows must be deleted and re-inserted based on the new values (most likely on a different AMP) using an all-AMP table lock.		Same as selection unless index or partitioning columns are updated. If index or partitioning columns are updated, this is treated as an "unreasonable update." That is, for an update of primary index or partitioning columns, updated rows must be deleted and re-inserted based on the new values using an all-AMP table lock. If a primary index column is updated, the row will most likely be re-inserted on a different AMP. If partitioning columns are updated without updating any primary index columns, a row will remain on the same AMP.
UPDATE (Upsert form)	This can be a more complex operation and will not be described here. For a specific UPSERT operation, refer to the appropriate EXPLAIN report.		
MERGE … INTO			

Selecting a Primary Index

Introduction

When assigning columns to be the primary index for a table, there are two essential factors to keep in mind: uniform distribution of rows and optimal access to the data.

Uniform Data Distribution

With respect to uniform data distribution, the following factors should be considered.

- The more distinct the primary index values, the better.
- Rows having the same primary index value are distributed to the same AMP.
- Parallel processing is more efficient when table rows are distributed evenly across the AMPs.

Optimal Data Access

With respect to optimal data access, you should consider the following factors.

- The primary index should be chosen on the most frequently used access path.

  For example, if rows are generally accessed by a range query, you should consider defining a partitioned primary index on the table that creates a useful set of partitions.

  If the table is frequently joined with a specific set of tables, then you should consider defining the primary index on the column set that is typically used as the join condition.
- Primary index operations must provide the full primary index value.
- Primary index retrievals on a single value are always one-AMP operations.

Criteria for Selecting a Primary Index

The following guidelines and performance considerations apply to selecting a unique or a nonunique column set as the primary index for a table.

- Choose columns for the primary index based on the selection set most frequently used to retrieve rows from the table even when that set is not unique (if and only if the values of the selection set are *nearly* unique).
- Choose columns for the primary index that distribute table rows evenly across the AMPs. The more singular the values for a column, the more optimal their distribution.

- Choose as few columns as possible for the primary index to optimize the generality of the index.

  All the columns in an index must be specified in a WHERE clause predicate before the Optimizer can select the primary index as the retrieval mechanism.
- If it is difficult to define a unique primary index for a table that must have one, you can generate arbitrary unique values for a single column if you define it as an identity column (see "Identity Columns" on page 14-13).
- Base the column selection on an equality search. For equality constraints *only*, the system hashes directly to the stored row that satisfies the condition.
-

Tables with this kind of primary key …	Tend to assign the primary index to …
Single column	the primary key. This is referred to as a Unique Primary Index (UPI).
Multicolumn	one of the foreign key components of the primary key. This is referred to as a Nonunique Primary Index (NUPI).

- Generally speaking, if the primary key for a table is not defined as the primary index, then you should define it as a Unique Secondary Index (USI) for the table.
-

A UPI …	WHILE a NUPI …
at most involves one row	can involve multiple rows.
does not require a spool file	often creates a spool file.

- Duplicate NUPI values are always stored on the same AMP and in the same data block if possible.
- NUPI retrieval only requires one I/O operation when the rows are stored in the same data block.
-

This type of value range …	Seen when using this predicate in a WHERE clause …	Results in this kind of retrieval action …
Implicit	BETWEEN	full table scan, irrespective of any indexes defined for the table. The exception is PPI tables, where partition elimination can be exploited.
Explicit	IN	individual row hashing.

Considerations for Choosing Between Partitioned and Nonpartitioned Primary Indexes

Selecting the optimum primary index for a table is often a complex task because some applications might favor one type of primary index, while other applications might perform more optimally using a different primary index. Tables can have only one primary index, however, so you must select one that best suits the majority of the applications that a table serves.

You can always add additional non-primary indexes, such as secondary, hash, and join indexes, to facilitate particular applications. Be aware that these indexes all incur various overhead costs, including the following major costs:

- Storage space.
- System performance decrements that occur whenever base table rows are updated. Keep in mind that each time a base table row is updated, the index values for any indexed columns affected by that update must also be updated.

You should always consider these tradeoffs when planning your indexes, then be sure to test to ensure that the assumptions that lead to your choices are correct. For example, if you design a primary index with even row distribution as your principal criterion, analyze the actual distribution of table rows to ensure that they *are* evenly distributed.

For many applications, a partitioned primary index can provide a better solution to resolving these issues than a nonpartitioned primary index because it provides efficient access both via the primary index columns as well as via a constraint on the partitioning columns. As always, you should confirm that the PPI actually improves query performance by carefully examining EXPLAIN reports and collecting the appropriate statistics. You should also weigh the costs of the index against the benefits. This is particularly important if you have also defined a USI on the table because additional maintenance is required to enforce uniqueness, thus potentially neutralizing or even reducing the overall performance advantage of the index.

Creating a table with a PPI does not guarantee that the index will be used in query plans. A PPI may not be used for any of the following common reasons:

- It is not applicable to the actual queries in the workload.
- The Optimizer cost analysis for a query determines that the index would not make the cost of the query less expensive.
- The query does not conform to any number of restrictions.

In some cases, a query plan with a PPI might not perform as well as one with an NPPI (see “Comparison of Partitioned and Nonpartitioned Primary Index Access for Some Typical Operations” on page 9-11 for specific examples).

Various partitioning strategies are possible:

- For some applications, defining the partitions such that each has approximately the same number of rows might be an effective strategy.
- For other applications, having a varying number of rows per partition might be desirable. For example, more frequently accessed data (such as for the current year) might be divided into finer partitions (such as weeks) but other data (such as previous years) may have coarser partitions (such as months or multiples of months).
- Alternatively, defining each range with equal width, even if the number of rows per range varies, might be important.

The most important factors for PPIs are accessibility and maximization of partition elimination. In all cases, defining a primary index that distributes the rows of the table fairly evenly across the AMPs is critical for efficient parallel processing.

Primary Index Summary

Teradata SQL primary indexes have the following properties.

- Controls data distribution and retrieval using the Teradata hashing algorithm (see "Teradata Hashing Algorithm" on page 8-32).
- Defined with the CREATE TABLE data definition statement (see "CREATE TABLE" in the chapter "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4)*.

  CREATE INDEX is used *only* to create secondary indexes.
- If no explicit primary index is defined, then CREATE TABLE assigns one automatically according to the rules described in "Primary Index Default Definitions" on page 9-9.
- Can be unique or nonunique (see "Unique Primary Indexes" and "Nonunique Primary Indexes" on page 9-4).
- If the primary index is not defined explicitly as unique, then the definition defaults to nonunique (see "Primary Index Default Definitions" on page 9-9).
- Can be partitioned or nonpartitioned (see "Partitioned Primary Indexes" and "Nonpartitioned Primary Indexes" on page 9-6).
- Can be composed of as many as 64 columns (see "Limits" on page 9-3).
- Can be generated automatically if defined on an identity column (see "Identity Columns" on page 14-13).

- A minimum of one must be defined per table (see "Limits" on page 9-3).
- A maximum of one can be defined per table (see "Limits" on page 9-3).
- Improves performance when used correctly in the WHERE clause of an SQL data manipulation statement to perform the following actions (see "Purpose of the Primary Index" on page 9-2):
  - Single-AMP retrievals
  - Joins between tables with identical primary indexes, the optimal scenario.

Further Information

See *Teradata RDBMS Performance Optimization* for more detailed information on using primary indexes to enhance the performance of your databases.

Primary indexes are also described in "CREATE TABLE" in *Teradata RDBMS SQL Reference, Volume 4*.

Principal Criteria for Selecting a Primary Index

Introduction

This topic introduces the three principal criteria you should evaluate when selecting a primary index for a table.

Principal Selection Criteria

You need to evaluate several different criteria when you analyze your logical data model for likely primary index candidates. The principal criteria you must consider when selecting a primary index for a table are the following.

- Access demographics (see "Access Demographics" and "Primary Index Value Retrieval Access" on page 9-19 and "Join Access" on page 9-21)
- Distribution demographics (see "Distribution Demographics" on page 9-22)
- Volatility of indexed column values (see "Volatility of Indexed Column Values" on page 9-23)

You will sometimes encounter situations where the selection criteria conflict. For example, specifying a NUPI instead of a UPI, or specifying an alternate key as the UPI instead of using the primary key. The decision to select a theoretically less pure column set over a formally more appropriate one is almost always made for reasons of performance.

Access Demographics

The topic of access demographics is composed of the following component topics:

- Primary index value retrieval
- Join access

Primary Index Value Retrieval Access

An *access* column is one that is commonly used as a valued predicate in a WHERE clause.

While it is true that the column set chosen to be the primary index for a table is often the same column set that defines the primary key, it is also true that primary indexes are often composed of fields that are neither unique nor components of the primary key for the table.

The primary guideline for selecting a primary index based on access demographics is to choose a single column or, less preferably, a set of several columns, that is most frequently used to access the table. In other words, define

the primary index for a table on a column set that is most frequently equated to discrete values in WHERE clause predicates in your application environment.

The reason for defining the primary index on the smallest possible column set is that you cannot hash or retrieve on a partial index value. Note that this is also true for USIs.

Conversely, if you specify too few columns in the primary index definition, then each primary index value might correspond to a large number of rows, a situation that often not only causes data skew, but also degrades any data maintenance that must touch all rows in a row hash.

The goal of this guideline is to maximize the number of single-AMP (primary index only) operations.

The same guideline applies to PPI access. The principal reason to define a table with a PPI is to facilitate partition elimination[1]. The degree of partition elimination depends both on the partitioning expression specified for the index and on the conditions specified in the query. Increasing the number of active partitions can degrade the performance of primary index access, joins, and aggregations on the primary index, but it also permits finer partition elimination; therefore, it is critical to understand the nature of the applications that are the predominant users of a PPI table. Note that it is not always necessary for all values of the partitioning columns to be specified in a query for partition elimination to occur (see "Comparison of Partitioned and Nonpartitioned Primary Index Access for Some Typical Operations" on page 9-11 for details about the relative performance implications of various access methods on tables with partitioned or nonpartitioned primary indexes).

A PPI provides optimal access to base table rows while also providing efficient join and aggregation strategies on the primary index in other situations. With a constraint on the partitioning columns of a table, PPI access performance can approach the performance of an NPPI depending on the degree to which the Optimizer can eliminate partitions from consideration by the query.

Access via an equality constraint on a primary index that also includes all the partitioning columns is as efficient as with an NPPI. If there is an equality constraint on a primary index that does *not* include all the partitioning columns, but there *is* an equality or other constraint on the partitioning columns that limits access to a single partition, then access is also as efficient as with an NPPI.

1 Partition elimination is analogous to how the Optimizer uses column projection and row restriction in that it eliminates partitions that are not relevant at the earliest possible stage in query processing.

Access via an equality constraint on the primary index that neither includes the partitioning columns nor constrains the partitioning columns, might not be quite as efficient as with an NPPI, depending on the number of active partitions.[2]

Although you can narrow access by specifying a particular partition set using the PARTITION keyword (see "PARTITION Column" on page 14-15), access to particular partitions is generally performed internally and need not be specified explicitly in a query.

IF a query ...	THEN ...
specifies values for all primary index and partitioning columns	a row can be retrieved by single AMP access from a single partition.
specifies values for all primary index columns and also specifies search conditions on the partitioning columns	partition elimination can reduce the number of partitions that must be probed on a particular AMP.
specifies values for all primary index columns but does not specify search conditions on the partitioning columns	each partition can be probed individually to locate rows based on their hash value only.
specifies search conditions on the partitioning columns	partition elimination can reduce an all-AMPs full table scan to an all-AMPs scan of only the partitions relevant to the query.
does not specify the values for all primary index columns and there are no constraints on the partitioning columns	an all-AMPs full file scan is required.

Join Access

Indexes are an extremely important component of any join. If, after implementing primary indexes for tables that are frequently joined, you detect significant performance issues when those tables are joined, you might want to reconsider your index column choices, particularly for commonly joined column sets in large tables. You might also want to consider adding additional columns to an existing index, depending on what your EXPLAINs for the query look like.

When designing for join access, you should first consider selecting common join columns for use as primary indexes. Joining tables on their primary indexes permits the Optimizer to specify hash or merge joins, two highly effective techniques, in the join plan to further optimize a query.

2 An active partition is defined as a partition that contains data.

The effectiveness of a merge join relates directly to whether it is made on primary indexes or not. Possible scenarios, in order of optimum performance, are as follows.

Join Predicate	Redistribution Action Required
Tables joined on their primary indexes.	None.
One join column is a primary index; the other is not.	Qualified rows from one table must be redistributed and sorted.
Neither join column is a primary index.	Qualified rows from both tables must be redistributed and sorted.

For multicolumn joins, you should consider using all, or at least a subset, of the join columns as the primary index.

Analogously to designing for row access, the primary guideline for selecting a primary index for join access is to choose a column or, less preferably, a set of columns, that is most frequently used to make the join. In other words, define the primary indexes for the tables to be joined on a column set that is most frequently equated to discrete values in WHERE or ON clause predicates in your application environment.

If you define a primary index on too many columns, then those queries that do not specify all of the columns in the index are not assigned a hash or merge join by the Optimizer.

If you define a partitioned primary index with too large a number of active partitions, join access performance can be degraded.

Distribution Demographics

The more singular the values of a column chosen as the primary index, the more even the distribution of table rows across the AMPs of a system. A unique index is ideal for ensuring optimal distribution. When distribution is optimized, so is parallel processing.

While it is true that the ideal primary index for a table both optimizes retrieval and distribution, the reality is that you are often faced with trading one off against the other. It is not exceedingly rare for a particular primary index to provide maximal access, but poor distribution (or vice versa).

The key to solving such a dilemma is to prototype several possibilities to see which is the best choice. You might have to make do with an optimal, rather than a maximal, solution.

The goal of this guideline is to optimize distribution without detracting from accessibility.

Distribution demographics are described in greater detail in “Analyzing Column Distribution Demographics to Optimize Primary Index Selection” on page 9-24.

With respect to partitioned primary indexes, the granularity of partitioning can be a significant factor in the general effectiveness of the index. The more partitions, the finer the ability to eliminate partitions. At the same time, too fine a granularity can have a negative effect on primary index access as well as joins and aggregations on the primary index. See "Comparison of Partitioned and Nonpartitioned Primary Index Access for Some Typical Operations" on page 9-11 for a detailed comparison of the various access mechanisms for the two primary index types over a range of typical operations and conditions.

Volatility of Indexed Column Values

Primary indexes should be defined on columns that rarely, if ever, change. Changing the value of a primary index almost certainly means that affected rows must be redistributed to different AMPs, which results in excessive I/O traffic on the BYNET and disk subsystems.

The goal of this guideline is to minimize system-wide performance decrements caused by frequently changed primary index values.

Analyzing Column Distribution Demographics to Optimize Primary Index Selection

Introduction

This topic examines guidelines for analyzing column distribution demographic data for use in selecting optimum columns to use for primary indexes.

Distribution Demographics

Among the column demographics retrieved by the COLLECT STATISTICS (Optimizer Form) statement are four figures that are very useful for monitoring how evenly your primary index columns distribute rows across the AMPs.

The four relevant demographics are the following.

- Number of distinct values within a primary index column
- Maximum number of rows per distinct value
- Maximum number of rows null
- Typical number of rows per distinct value

Number of Distinct Values

This number reports how many distinct values a particular primary index column contains.

With respect to NUPI behavior, the closer the ratio of distinct index column values to table cardinality is to 1.0, the better. For a UPI, this value is always 1.0 by definition.

FOR this type of primary index …	The ratio of distinct index column values to table cardinality is …
unique	always 1.0.
nonunique	best for distribution when it is close to 1.0 and worse the further away from 1.0 it becomes.

With respect to space utilization, the number of distinct values should always be *very* much greater than the number of AMPs in the configuration to ensure that each table distributes its rows to all AMPs.

While this might not be possible to achieve for very small tables, it is an absolute necessity for anything larger.

If the number of rows per distinct value for a column is significantly larger than 1, then it is not a good candidate for a primary index.

Maximum Number of Rows Per Distinct Value

This value reports the largest number of rows per primary index column value.

FOR this type of primary index …	The value for this measure is …
unique	always 1.
nonunique	better as it approaches 1 and worse as it gets larger.

If the maximum number of rows per column value is much larger than 1, then the column is probably not a good candidate for a primary index. The severity of the penalty paid for larger values is a function of several variables, including the cardinality of the table, the number of AMPs in the configuration, and so on.

Maximum Number of Rows Null

This value reports the number of rows having a null primary index.

FOR this type of primary index …	The value for this measure is …
unique	better as it approaches 0 and worse as it gets larger.
nonunique	

A very large number reflects a spike in the distribution, indicating the likelihood of serious problems with an uneven consumption of disk space because all rows with a null primary index hash to the same AMP (see "Number of Distinct Values" on page 9-24 for a description of the problem).

If the number of null primary indexes is significantly larger than 0, then the column is not a good candidate for a primary index. See Chapter 13: "Designing for Missing Information," for a description of the many problems nulls cause for database management and suggestions about how to minimize or eliminate nulls from your tables.

Typical Number of Rows Per Distinct Value

This value provides a measure of the overall distribution the selected primary index column set provides.

FOR this type of primary index …	The value for this measure is …
unique	always 1.
nonunique	better as it approaches 1 and worse as it gets larger.

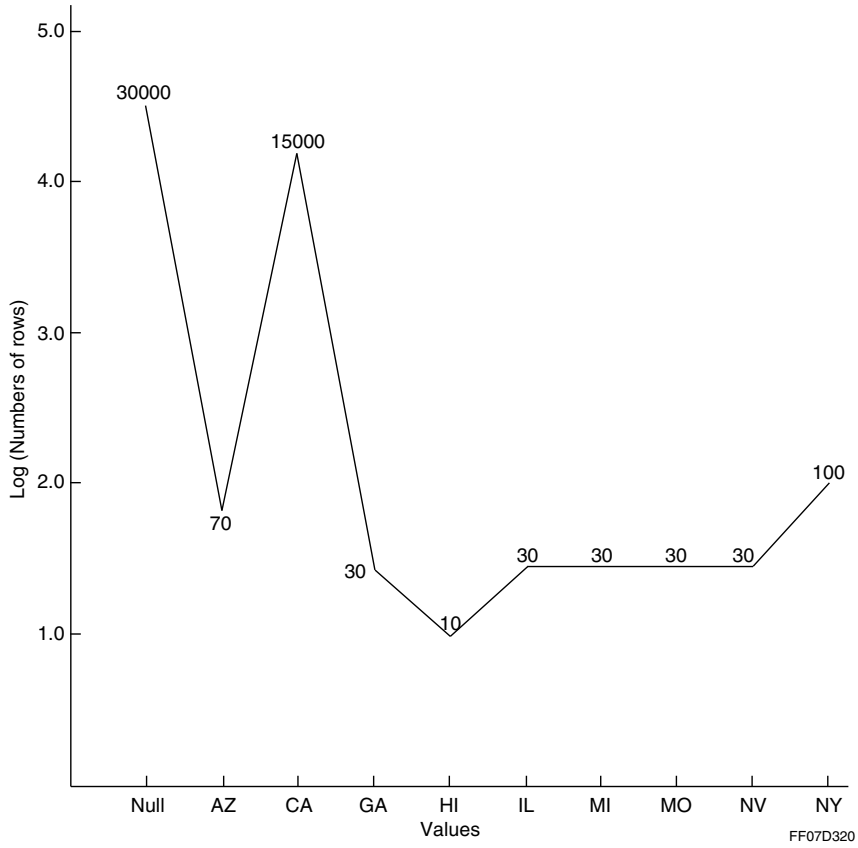
If the typical quantity of rows per column value does not fit into a single data block, then the column set is not a good candidate for a primary index.

You can graph these figures to provide an easy-to-comprehend graph of column value distributions. In the example provided here, a NUPI on an attribute called State was analyzed. Note the exceedingly skewed distribution, suggesting that State is not a good candidate for the primary index on this table.

The following table indicates the raw and logarithmic numbers of rows per state code value.

State Code	Number of Rows	$\log_{10}$(Number of Rows)
Null	30 000	4.477
AZ	70	1.845
CA	15 000	4.176
GA	30	1.477
HI	10	1.000
IL	30	1.477
MI	30	1.477
MO	30	1.477
NV	30	1.477
NY	100	2.000

The row counts are displayed on a logarithmic scale in the following graph. Notice how skewed the distribution is even when displayed on a logarithmic scale.



SQL Scripts For Detecting Skew

If you are analyzing the demographics of an existing table, whether production or prototype, you can use the following set of useful scripts written by the Teradata technical support team to check for data skew. You can adjust the details of these statement to suit the needs of your site.

This is a good practice to undertake when new applications are being loaded on the system. It is also good practice to run these queries regularly if there are many data changes.

The following query identifies tables that are not evenly distributed. Ideally, variance should be less than 5 percent. In this query, variance is set to 1000 percent, which generally indicates that some or many AMPs have no rows from the table in question. You can use the BTEQ command RETLIMIT to limit the number of rows returned.

```
SELECT (MAX(CurrentPerm) - MIN(CurrentPerm)) * 100
/(NULLIF(MIN(currentperm),0))(NAMED variance)(FORMAT 'zzzzz9.99%')
,MAX(CurrentPerm)(TITLE 'Max')(FORMAT 'zzz,zzz,zzz,999')
,MIN(currentperm)(TITLE 'Min')(FORMAT 'zzz,zzz,zzz,999')
,TRIM(DatabaseName)||'.'||TableName (NAMED Tables)
FROM DBC.tablesize
GROUP BY DatabaseName, TableName
HAVING SUM(CurrentPerm) 1000000
AND variance 1000
WHERE DatabaseName NOT IN('CrashDumps','DBC')
ORDER BY Tables;
```

Use the following query to display the detailed distribution of a table that has been identified as having a skewed distribution:

```
SELECT vproc, CurrentPerm
FROM DBC.TableSize
WHERE DatabaseName = '<databasename>'
AND TableName = '<tablename>'
ORDER BY 1;
```

The following query reports the row distribution of a table by AMP:

```
SELECT dt1.a (TITLE 'AMP')
,dt1.b (TITLE 'Rows')
,((dt1.b/dt2.x (FLOAT)) - 1.0)*100 (FORMAT'+++9%',TITLE 'Deviation')
FROM
  (SELECT HASHAMP(HASHBUCKET(HASHROW(<index>)))
  ,COUNT(*)
   FROM <databasename>.<tablename>
   GROUP BY 1
  )dt1 (a,b)
    ,(SELECT (COUNT(*) / (HASHAMP()+1)(FLOAT))
      FROM <databasename>.<tablename>
     )dt2(x)
ORDER BY 2 DESC,1;
```

The following query reports the distribution by AMP of the specified index or column.

```
SELECT HASHAMP(HASHBUCKET(HASHROW(<index or column>)))
,COUNT(*)
FROM <databasename>.<tablename>
GROUP BY 1
ORDER BY 2 DESC;
```

The following query reports the number of row hash collisions for the specified index or column:

```
SELECT HASHROW(index or column), COUNT(*)
FROM <databasename>.<tablename>
GROUP BY 1
ORDER BY 1
HAVING COUNT(*) > 10;
```

The following query reports the number of AMPs and the number of rows a given query accesses:

```
LOCKING TABLE <tablename> FOR ACCESS
SELECT COUNT(DT.ampNum) (TITLE '#AMPS')
,SUM(DT.numRows) (TITLE '#ROWS')
FROM
  (SELECT HASHAMP(HASHBUCKET(HASHROW(<index>)))
  ,COUNT(*)
   FROM <tablename>
   WHERE <selection criteria>
   GROUP BY 1)DT (ampNum, numRows);
```

PPI Case Studies

This topic presents several case study scenarios to evaluate various approaches to partitioning a primary index in such a way that optimal benefit is realized.

- The first scenario (see "Scenario 1" on page 9-31) is set in a retail sales environment. The effects of partitioning a sales table on months is examined from several different perspectives.
- The second scenario (see "Scenario 2" on page 9-35) is a continuation of the first and attempts to boost the benefits of using a PPI table further by partitioning its primary index on days rather than months. The concept is to investigate how the number of primary index partitions affects query workload performance.
- The final scenario (see "Scenario 3" on page 9-37) is set in a telecommunications environment. The effects of partitioning a telephone call tracking table on call date and phone number are compared and contrasted for the same query workloads.

The optimal partitioning scheme for any table depends on the anticipated query mix. Use your extended logical data model as the starting point for making the decision, but you should always test a few different scenarios to ensure the best partitioning scheme for your particular application and system configuration.

Scenario 1

Table Structure and Update Schedule

Assume that a retail enterprise has a large, NPPI sales table containing the details of each transaction for the previous 24 full months plus the current month-to-date. Once a month, the transactions from the oldest month are deleted from the table. Current transactions are loaded into the table nightly using MultiLoad. Most transactions are added on the date they occur, but a small percentage of transactions might be reported a few days after they occur. The number of transactions per month is roughly the same for all months.

Each row contains, among other things, the product code for the item, the transaction date, an identifier for the sales agent, and the quantity sold. The rows are short, and the data blocks are large. The primary index is a composite of product code, transaction date, and the agent identifier.

Query Workload

The query workload is a mix of tactical and strategic requests. There is a modest volume of short-running single-AMP queries. Many ad hoc queries follow a general pattern of comparing current-month-to-date sales to the same days of the previous month, or to the same days of the same month of the previous year for a few product code values. Some queries analyze agent performance, usually over an interval of a calendar quarter or less. Other queries examine sales trends over the previous 24 full months, usually for most or all product code values. The sales table is frequently joined to relatively small tables containing information about each product code and each sales agent.

Problem Statement

The DBA has been told to speed up the ad hoc queries and agent analysis queries. He first considered defining either a UPI or a join index on the transaction date column and had set up tests for those scenarios. Unfortunately, the EXPLAIN reports show that the optimizer finds neither index to be selective enough to improve performance over a full-table scan, and does not use them.

The next option he considers is to split the table into 25 separate tables, each containing transactions for a calendar month, and then to define a view that UNIONs all the tables. This view is intended to be used by the applications that analyze 24 months of sales history. After some analysis, he concludes that this solution could indeed speed up the targeted queries, but that it also added too much complexity for his end users, who would now have to understand the view structure and change the table names in their queries, code more complicated UNION statements, and select appropriate date and product code ranges. The requirement to know the right table name also applies to the

short-running single-AMP queries that specify primary index values. This proposed solution also complicates the nightly load jobs, especially in the first few days of a month when a small number of the transactions would be from the prior month, as well as complicating his long successful archive strategy. The DBA ultimately rejects this alternative as being too complicated and error-prone.

Analysis of PPI Benefits

The DBA decides to investigate using a PPI, which he thinks might be an easy way to optimize his workloads for this situation. He converts the sales table into a PPI table partitioned by transaction month and finds that many of his critical queries run faster with no significant negative tradeoffs.

The following text looks at each element of the workload as it relates to the newly redefined PPI sales table.

- The monthly deletes are faster because instead of a monthly MultiLoad delete job, the DBA can now perform a simple monthly ALTER TABLE statement to do the following things:

  1 Drop the entire partition that contains the oldest data.

  2 Create a few new partitions to hold data for the next few future months.

  Deleting all the rows in a partition is optimized in much the same way that deleting all the rows in an NPPI table is optimized. For example:
  - There is no need to record the individual rows in the transient journal as they are deleted.
  - The rows for the month being deleted are stored contiguously on each AMP instead of being scattered more or less evenly among all the data blocks of the table, so there are fewer blocks to read.
  - Most of the deletes are full-block deletes, so the block does not need to be rewritten.
  - There is no need to touch any of the rows for the other months.

  Dropping the oldest partition set is a nearly instantaneous operation, assuming there are no USIs or join indexes that must be updated.
- The nightly MultiLoad insert job should run somewhat faster than before. Instead of the inserted rows distributing more or less evenly among all the data blocks of the table, they are concentrated in data blocks that correspond to the proper month. This increases the average hits per block count, which is a key measure of MultiLoad efficiency, as well as reducing the number of blocks that must be rewritten.
- The short-running, single-AMP, queries are not affected by the PPI. Because the partitioning column is a component of the primary index, primary index access to the table is not changed.

- The largest gain is seen in the ad hoc queries comparing current month sales to a prior month. Only two of the 25 partitions need to be read, instead of the full-table scan required for an NPPI table. This means that the number of disk reads will be reduced by roughly 92 percent, with a corresponding reduction in elapsed query response time. The 92 percent figure applies to the step that reads the sales table, not to the sum of all the steps used to implement the query.

  Given the stated assumptions, the other steps should take roughly the same amount of time to complete as they did previously.
- The same considerations apply to the agent analysis queries. The number of partitions that must be read is determined by the time period specified in the query is known in most cases to be three or fewer. Even if the analysis is over twelve full months, there is roughly a 50 percent gain in reading twelve of 25 partitions for the step that reads the sales table.
- The decision support queries that analyze 24 months of sales data take roughly the same time and resources as they did for an NPPI table, although there is a marginal gain realized by reading 24 instead of 25 partitions. If the analysis is for 24 months plus the current month (the entire table), then the resource usage is the same as with the NPPI incarnation of the table because full-table scans are not affected by partitioning the primary index.
- The joins take the same amount of time. In the defined workload, the EXPLAIN reports indicate that there are no direct joins against the sales table. Instead, a spool file is always created from the sales table and the spool file is then joined to the smaller tables. In the PPI table case, the spool file might be created more efficiently, but once created it is exactly the same as it is for the NPPI table.

Conclusions

The results show there are few advantages to partitioning this table given the workloads that access it. The PPI table requires somewhat more disk space than its NPPI counterpart. There is a two-byte partition number recorded in each row that consumes additional storage space; however, the percentage increase seen for most row sizes does not exceed about 5 percent, and is often considerably less than that.

The following table summarizes the improvement opportunities for this case study:

Activity	Non-PPI Table	PPI Table	Improvement	Comments
Monthly delete of one month of data	MultiLoad job reads most blocks, updates most blocks	SQL statement deletes partition	Faster performance	Easier maintenance
Nightly inserts	Inserted rows scattered throughout table	Inserted rows concentrated in one partition	Faster performance	No changes to load script needed
Primary index access	One block read	One block read	No change	No SQL changes needed
Comparison of current month to prior month	All blocks read	Two partitions read	Step is 12 times faster (only 2 of 25 partitions read)	No SQL changes needed
Trend analysis over entire table	All blocks read	All blocks read	Little change	• Rows are two bytes longer for PPI • 2% more blocks for 100 byte rows
Direct joins	Not applicable	Not applicable	Not applicable	No direct joins in this example

Scenario 2

Table Structure and Update Schedule, Query Workload, and Problem Statement

This case study is a continuation of the scenario presented in "Scenario 1" on page 9-31. The table structure, update schedule, query workload, and problem statement are unchanged. The scenario evaluates whether the PPI definition for this table can be improved further.

Analysis of PPI Benefits

The PPI definition used for "Scenario 1" on page 9-31 partitioned the primary index by month because many of the queries use months as their basic time unit. Another option is to attempt to optimize access further by changing the granularity of the partitioning scheme by partitioning at a finer level. Suppose the DBA considers partitioning the table by day instead of by month. The table now has about 760 partitions (25 months), with those corresponding to future dates in the current month being empty.

The results of telescoping the partition granularity for this table and query workload look like this:

- The effort to delete the oldest month of data is substantially the same as it was for the same table defined with 25 primary index partitions. The run time for that job is also substantially the same. Making the simplifying assumption that all months have 30 days, the DBA would have to delete 30 smaller partitions with the oldest month of data instead of deleting one larger partition, but the same number of rows are deleted, and the elapsed time to delete them is roughly the same.
- The nightly inserts benefit from the finer partitioning because instead of being concentrated in one or two partitions out of 25, the inserted rows are now directed to three, four, or at most five partitions out of 760; well under one percent of the partitions. Most of the inserts are directed to one partition, the one containing the activity for the day just completed.
- The short-running primary index access queries are not impacted either way by having 760 partitions instead of 25.

- The ad hoc queries that usually analyze two or three months of data are also little changed, now accessing roughly 60 out of 760 partitions rather than 2 out of 25, roughly the same percentage of the defined partitions for the primary index. Queries varying by the time of month, however, might realize some performance gain with the larger number of partitions. For example, a query submitted on the fourth day of the month might be likely to analyze the current day and the previous 33 days of data, while a query submitted later in the month might restrict the analysis to that calendar month. In this case, the 34 day query would involve 68 out of 760 partitions, instead of 4 out of 25, a significantly smaller percentage.
- The analysis queries examining 24 months of data run in about the same length of time because they touch most of the rows in the table in either case.
- The joins are not impacted by the number of partitions because there are no direct joins against this table.

Conclusions

Increasing the number of partitions from 25 to 760 has only a modest effect on performance for this particular workload. The greatest gain is for queries that analyze only a few days of transactions.

Scenario 3

Table Structure and Update Schedule

While transaction date is frequently a good choice for the partitioning column, other choices present themselves for consideration for other categories of workloads and data. Consider a table with detailed information about telephone calls maintained by a telecommunications company. This table stores rows that contain the originating telephone number, a timestamp for the beginning of the call, and the duration of the call, among other things, for each outgoing call. Rows are retained for a variable length of time, but rarely for more than six weeks. Retention is based on the call date and the monthly bill preparation date. The primary index is a composite of the telephone number and the call-start timestamp. This column set implies that the index was chosen to provide good distribution, not to facilitate data access, and also that the likelihood of any direct primary index joins is remote.

Query Workload

Some queries analyze all calls from a particular telephone number, while others analyze all calls for a particular period of time, perhaps for as long as a month, for customers meeting certain criteria.

Analysis of PPI Benefits

There are two candidate schemes for partitioning this table. The first partitioning scheme is to extract the date from the timestamp and partition the rows by date, probably defining one partition for each day. The results of this partitioning scheme for this table and query workload look like this:

- The date partitioning scheme helps with inserting new transaction activity in the same way as the previous scenario (“Scenario 1” on page 9-31).
- Little performance gain is realized for row deletion because the delete operations are not performed strictly by call date. Because of this, the deleted rows are not clustered in a single partition.
- The analysis queries based on the call date benefit from this partitioning scheme, with those queries that specify a range of a few days realizing the greatest gain.

The second possibility for partitioning this table is to use the telephone number. Telephone numbers contain too many digits to assign each number its own partition, but a subset of the digits can be used profitably to optimize telephone number-based row access. If, for example, the table is partitioned on the high order three digits of the telephone number, there are 1 000 partitions, some of which are always be empty because of the way telephone companies assign numbers. The results of this partitioning scheme for this table and query workload look like this:

- This partitioning scheme does not improve the performance of bulk inserts or deletes because they are scattered across all partitions.
- The scheme does not facilitate date-based queries.
- The scheme allows queries that specify a telephone number to run much faster than they otherwise would because only one partition out of 500 or more partitions must be read to access the rows having that number.
- The scheme benefits geographic area analysis, at least in North America, because the first three digits of North American telephone numbers uniquely identify a narrowly defined geographic region.

If 1 000 partitions improve performance, then defining 10 000 partitions using the first four digits of the telephone number would probably improve performance even more. If 10 000 partitions are good, then 50 000 partitions might be better yet. The upper limit on partitions is 65 535 per table, so it is not possible to define 100 000 partitions, but you could use the first five digits of the telephone number and then assign two consecutive numbers to each partition.

If mapping a geographic area to one or more partitions does not solve an application problem, then another potential solution is to maximize the number of partitions by using telephone number modulo 65 535 as the partitioning expression. Assuming the cardinality of the table is roughly 3.276 billion rows, then the average partition contains roughly 50 000 rows with this scheme. If the system has 100 AMPS, then each AMP contains roughly 500 rows per partition, a number that fits into one data block if the row width is fairly narrow. The decrease in response time of a single-partition scan for all activity for a particular telephone number is dramatic compared to the full-table scan that would be required for an NPPI table.

Conclusions

The partitioning scheme you use should be tied directly to the application workload accessing its data. For this scenario, the proposed partitioning schemes affect geographic localization applications differently than they do call date-based applications. To decide which scheme is better, you need to know all the various ways the table is currently accessed and have a solid concept of how it might be accessed in the future.

You need to examine the relative efficiencies of different partitioning schemes for the same table must from that perspective.

Summary of Primary Index Selection Criteria

Introduction

This topic summarizes the guidelines for selecting columns to be the primary index for a table.

Guidelines for Selecting Primary Index Columns

The following table summarizes the guidelines for selecting columns to be used as primary indexes.

Guideline	Comments
Select columns that are most frequently used to access rows.	Restrict selection to columns that are either unique or highly singular.
Select columns that are most frequently used in equality predicate conditions.	Equality conditions permit the system to hash directly to the row having the conditional value. When the primary index is unique, the response is never more than one row. Inequality conditions require additional processing.
Select columns that distribute rows evenly across the AMPs.	Distinct values distribute evenly across all AMPs in the configuration. This maximizes parallel processing. Rows having duplicate NUPI values hash to the same AMP and are stored in the same data block whenever possible. This is good when rows are only moderately nonunique. Rows having NUPI columns that are highly nonunique distribute unevenly, use multiple data blocks, and incur multiple I/Os. Extremely nonunique primary index values can skew space usage so markedly that the system returns a message indicating that the database is full. This occurs when an AMP exceeds the maximum bytes threshold for a user or database calculated by dividing the PERMANENT = n BYTES specification by the number of AMPs in the configuration, causing the system to incorrectly perceive the database to be “full.”
Select columns that are not volatile.	Volatile columns force frequent row redistribution.
Select columns having very many more distinct values than the number of AMPs in the configuration.	If this guideline is not followed, row distribution skews heavily, not only wasting disk space, but also devastating system performance. This rule is *particularly* important for large tables.

Locating a Row Using Its Primary Index

Introduction

This topic illustrates single-AMP retrieval of a row by means of its primary index.

Supporting Query

Assume the following abstract SQL query.

```
SELECT column_1, column_2
FROM table_1
WHERE unique_primary_index = column_value;
```

This query instructs Teradata to retrieve the single row having the unique primary index value *column_value* and then to display the values for column_1 and column_2 of that row.

Procedure for Locating a Row Using Its Unique Primary Index

The following table describes how Teradata locates the row having the unique primary index value *column_value*. Teradata follows the same process if the primary index is nonunique. The only difference is that the query might retrieve more than a single row in that case because multiple rows could have the same value for *column_value*.

Stage	Process
1	The Resolver (see "Definitions for Parsing Engine Software" on page 8-2, for a description of the Resolver) scans the data dictionary to determine the table ID for *table_1*, then adds it to the parse tree for later use by the Generator.
2	The Generator (see "Definitions for Parsing Engine Software" on page 8-2, for a description of the Generator) hashes the primary index value *column_value*, computing its 32-bit row hash value.
3	The Generator builds a 3-part message from the following information. • Table ID for *table_1* • 32-bit row hash for *column_value* • 16-bit hash bucket number • 16-bit remainder • Data value for *column_value*
4	The Generator scans the hash map to determine which AMP owns the hash bucket the row belongs to.

<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>5</td><td>The message is inserted into an AMP step, the Dispatcher (see “Definitions for Parsing Engine Software” on page 8-2, for a description of the Dispatcher) places it on the BYNET, and sends it point-to-point to the AMP identified by the hash map.</td></tr>
<tr><td>6</td><td>The file system on the receiving AMP uses the table ID and row hash value as a key to scan its master index for the cylinder number that contains the data block in which the row is stored.</td></tr>
<tr><td>7</td><td>The file system first determines if the cylinder index is cached.

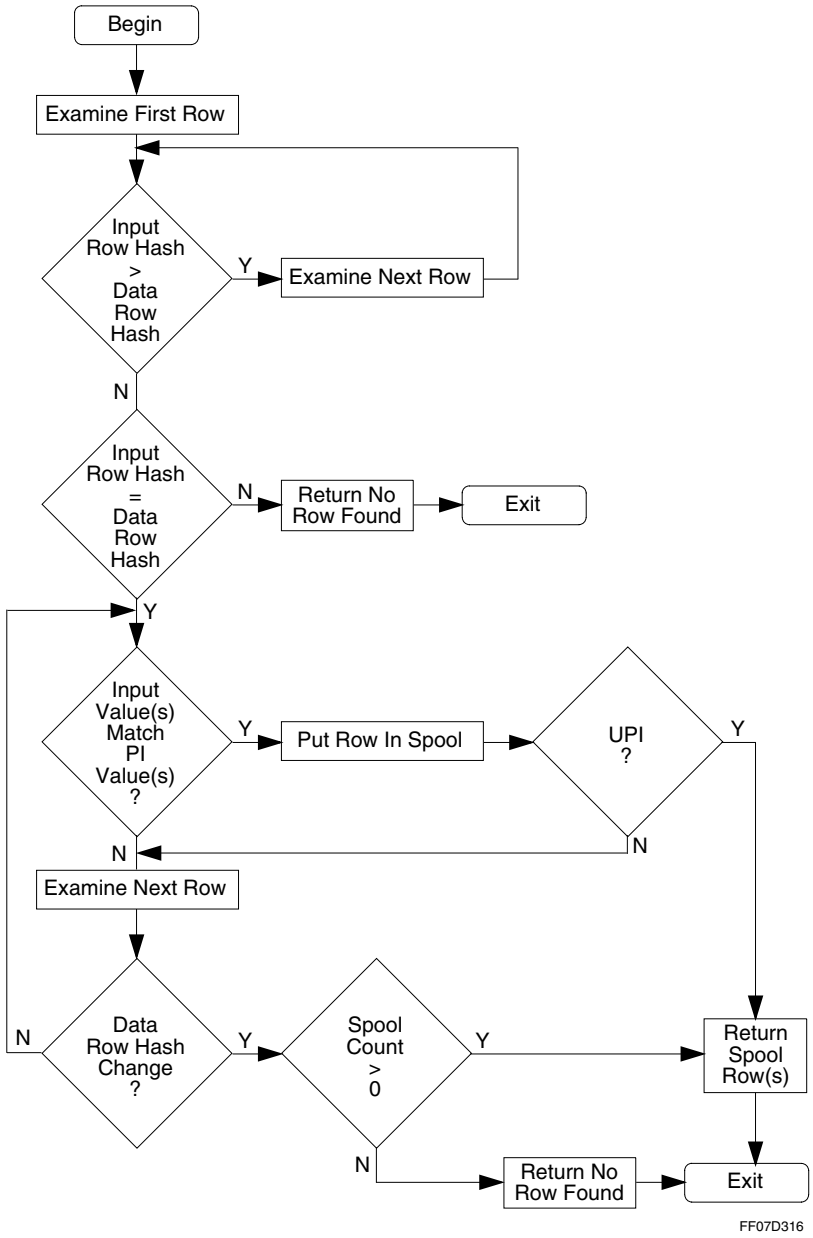
<table><tr><th>WHEN the cylinder index is …</th><th>THEN the file system …</th></tr><tr><td>cached</td><td>scans it for the data block.</td></tr><tr><td>not cached</td><td>retrieves it from disk and scans for the data block.</td></tr></table></td></tr>
<tr><td>8</td><td>The file system uses the table ID, row hash value, and cylinder number as a key to scan the cylinder index for the data block address known to contain the row being retrieved.</td></tr>
<tr><td>9</td><td>The file system determines if the data block is cached.

<table><tr><th>WHEN the data block is …</th><th>THEN the file system …</th></tr><tr><td>cached</td><td>scans it for the row.</td></tr><tr><td>not cached</td><td>retrieves it from disk and scans for the row.</td></tr></table></td></tr>
<tr><td>10</td><td>The file system uses the row hash value and primary index value as a key to scan the data block for the row being retrieved and returns the values for column_1 and column_2 to the requestor.

See the graphic under “Primary Index Reads” on page 9-42 for more details on the process of a primary index read of a data block.</td></tr>
</table>

Primary Index Reads

The following flowchart illustrates the flow of a generic primary index read. The flowchart begins at the point that the data block containing the searched row has been located.



Example: Primary Index Read

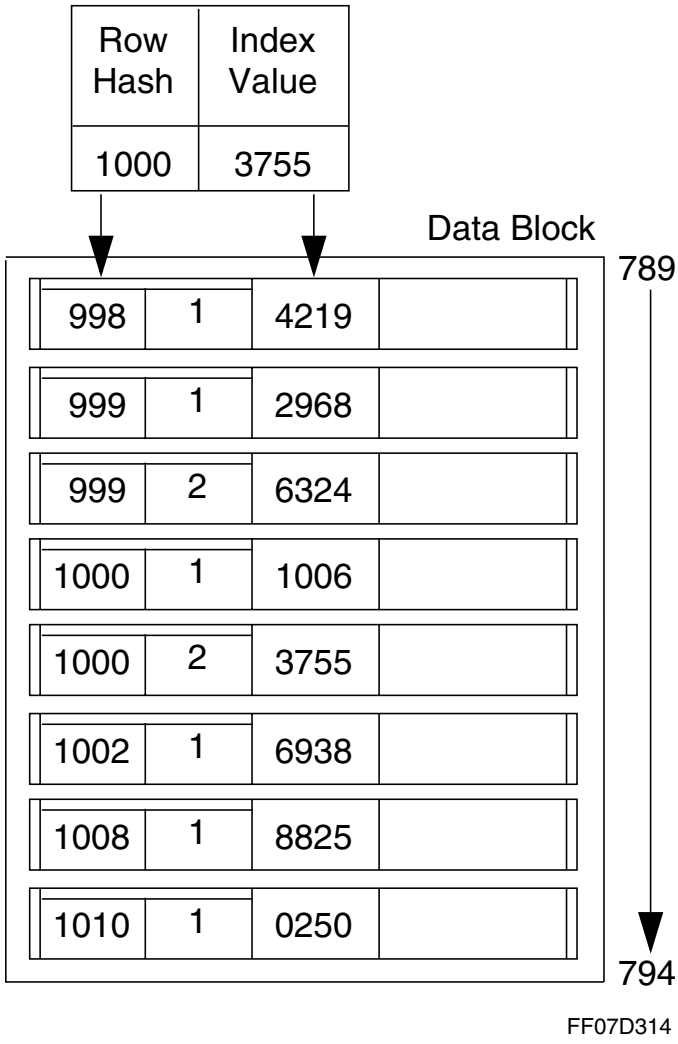
This example illustrates how the Teradata file system uses the primary index to eliminate synonyms from the returned result by performing a row hash search.

The original query is the following.

```
SELECT *
FROM employee
WHERE employee_number = 3755;
```

At this point in the process, the master index lookup showed that the row representing employee number 3755 is stored on cylinder 169. The cylinder index lookup showed that the row is in the six sector block beginning with sector number 789.

The following graphic illustrates the logical representation of that data block.



The file system must locate the row having a row hash value of 1000 and a primary index data value of 3755. The following table describes how the file system undertakes this final task.

Stage	Process
1	The file system begins scanning the data block beginning at sector 789.
2	The first row found with a row hash value of 1000 has an index data value of 1006. This is a hash synonym for the searched row. Because this is not the row specified by the WHERE clause predicate, the scan continues.
3	The next row found with a row hash value of 1000 has an index data value of 3755. Because this is the requested row and it was accessed using a UPI, the data block scan terminates and the row is returned to the requestor. Had the index been a NUPI, the file system would have had to read another row to know that no further rows matched.

Secondary Considerations for Selecting a Primary Index

Introduction

This topic introduces additional considerations for selecting the primary index for a table.

Additional Considerations for Primary Index Selection

When you select primary indexes for your tables, there are several factors to keep in mind that are not orthogonal to the three principle selection criteria described in "Principal Criteria for Selecting a Primary Index" on page 9-19.

These considerations include the following factors.

- Performance
- Duplicate row checks on NUPIs
- Space management

Each of these considerations is described in further detail in additional information maps in this chapter.

Performance Considerations for Primary Indexes

Introduction

This topic describes general performance considerations for both unique and nonunique primary indexes, paying particular attention to how the range of primary index singularity affects performance.

Guideline

Define the primary index for a table on as few columns as possible.

Retrievals on columns that do not match the *entire* primary index do not use the primary index to retrieve the matching rows.

Hashing efficiency increases as the number of primary index columns decreases.

General Considerations

The following factors apply equally to Teradata UPIs and NUPIs.

- Each table has one, and only one, primary index.
- The primary index for a table need not be isomorphic with the primary key for the table. This is obviously true for NUPIs, because primary key values are unique by definition.
- A primary index can contain nulls, while a primary key cannot.

  Primary index nulls are not a good thing. If you anticipate that the primary index for a table might frequently be null, you should consider using a different column.
- All single value primary index accesses are confined to a single AMP. In most cases, a single value primary index access requires only one I/O.

Unique Primary Index Considerations

The following factors describe the general performance considerations associated with UPIs.

- UPIs always guarantee an even distribution of table rows across the AMPs.
- UPIs always guarantee the best retrieval and update performance for single table row access.
- UPIs are never associated more than one base table row.
- UPIs retrievals and updates never require a spool file.
- UPI uniqueness is enforced by Teradata.

Nonunique Primary Index Considerations

The following factors describe the general performance considerations associated with NUPIs.

- At best, NUPIs distribute table rows evenly across the AMPs.

  At worst, NUPI table row distribution can be skewed and, if so, decrement both retrieval and update performance.
- NUPIs at best can effect good retrieval and update performance.
- NUPIs can involve more than one base table row.
- NUPI retrievals can require spool files.
- Duplicate NUPI values hash to the same AMP and, whenever possible, are stored in the same data block.
- If all the duplicate rows for a primary index value fit into a single data block, then only one I/O is required to store or access the entire set.
- If duplicate rows are excluded because the table is defined as SET and has no USI, then the system must make a duplicate row check for every table row inserted or updated. See "Duplicate Row Checks for NUPIs" on page 9-50 for further details.

Normalization, Denormalization, and Primary Index Usage

Introduction

The following example shows how a well-designed normalized table set can be as performant as the equivalent denormalized form while avoiding all the redundancy and update anomaly difficulties of denormalization.

The most frequently performed query against these tables determines the current price for a given part.

Denormalized Table Set

The first set of tables is denormalized by including Current Effective Date and Current Price Amount columns in the Part table. Because the price of the part is also carried in the Part Price History table, this is a violation of normalization due to redundant data.

Part **7 000 000 rows**

Part Number	Quantity	Description	Current Effective Date	Current Price Amount
PK			DD	
1234	100	Widget	1996/01/01	14.25
1235	97	Thingie	1996/01/01	28.99

Part Price History **70 000 000 rows**

Part Number	Former Effective Date	Former Price Amount
PK		
FK		
1234	1992/01/01	13.95
1234	1998/01/01	15.45
1235	1992/01/01	27.89
1235	1998/01/01	30.99

Normalized Table Set

The second set of tables is normalized. By defining a NUPI on Part Number, you can determine the current price of any Part by reading only one additional block on the same AMP that holds the Part table row.

With a composite UPI defined over Part Number and Effective Date, the lookup cost for determining the current price of a Part would be substantially higher.

Part **7 000 000 rows**

Part Number	Quantity	Description
PK		
UPI		
1234	100	Widget
1235	97	Thingie

Part Price History **70 000 000 rows**

Part Number	Effective Date	Price Amount
PK		
FK		
NUPI		
1234	1992/01/01	13.95
1234	1996/01/01	14.25
1234	1998/01/01	15.45
1235	1992/01/01	27.89
1235	1996/01/01	28.99
1235	1998/01/01	30.99

Duplicate Row Checks for NUPIs

Introduction

Teradata tables defined as SET in the CREATE TABLE kind clause do not permit duplicate rows. Any time a row is inserted into a SET table having a NUPI and no unique constraints, the system performs a duplicate row check.

Performance Effects of Duplicate NUPI Row Checks

Duplicate row checking can be a very I/O-intensive operation. The table on page 9-51 illustrates the number of logical reads performed for the duplicate row checks required by a given number of duplicate NUPI row insertions under the following conditions.

- The table is a SET table (no duplicate rows permitted).
- There are no previous hash synonyms for the NUPI value being inserted.

The center column in the table indicates the number of logical reads required before the number of rows specified in the left column can be inserted into the table.

The right column indicates the cumulative number of logical reads required before the number of rows listed in the left column can be inserted. The cumulants are determined from the following equation.

$$(\text{column 3, row n}) = (\text{column 2, row n}) + (\text{column 3, row n-1})$$

More formally, the number of NUPI rows that must be read can be expressed as follows.

$$\text{Number of rows to read} = n \times \frac{(n-1)}{2}$$

where n = number of NUPI rows.

Duplicate NUPI Row Read I/O as a Function of Number of Rows Inserted

This table illustrates the absolute and cumulative number of read I/Os required as a function of the number of duplicate NUPI rows inserted into a table. These figures are not a function of the number of AMPs in the configuration because duplicate NUPI values all hash to the same AMP. Therefore, the magnitude of the effect is independent of the number of AMPs on the system.

As you can see, the read I/O burden is tractable up to about 100 duplicate NUPI rows inserted. Beyond that point, the read I/O requirements are increasingly *in*tractable. The performance of any system is noticeably worsened when any additional duplicate NUPI rows are added.

Number of Rows Inserted	Number of Prior Logical Reads Required to Process This Number of Inserted Rows	Cumulative Number of Logical Reads
1	0	0
2	1	1
3	2	3
4	3	6
5	4	10
6	5	15
7	6	21
8	7	28
9	8	36
10	9	45
20	19	190
30	29	435
40	39	780
50	49	1 225
60	59	1 770
70	69	2 415
80	79	3 160
90	89	4 005
100	99	4 950
200	199	19 900
300	299	44 850
400	399	79 800
500	499	124 750
600	599	179 700
700	699	244 650
800	799	319 600
900	899	404 550
1 000	999	499 500

Guidelines for Keeping NUPI Duplicates Below 100 Rows Per Value

As you can see in the previous table, beyond more than a very few rows, the number of logical reads required to insert duplicate NUPI value rows is prohibitive.

The file system must read all the rows on an AMP that have the same row hash in order to determine their uniqueness values before it can insert a new NUSI duplicate row. Beyond 100 rows per value, the performance decrement created by that activity becomes significant, so you should always limit NUPI duplicates to fewer than 100 rows per value for any table.

The figure of 100 duplicates per NUPI value is based on the overwhelming likelihood that duplicate NUPI value rows will spill over into more than five data blocks. When the number of duplicate NUPI values in a table exceeds 100, performance *always* degrades significantly. The specific operations affected are described in the following table.

Operation	Cause of Performance Degradation
Updates	Increased I/O
• Inserts • FastLoads	Increased comparisons, resulting in greater I/O activity and CPU usage

Note that performance for the following operations is also degraded significantly when the number of duplicate NUSI row values exceeds 100.

- Restore operation of the Archive/Recovery utility
- Table rebuilds using the Table Rebuild utility

These performance problems persist even for the unlikely case where every NUPI value in a table has the same number of duplicates, thus eliminating skew as a factor in the decrement.

Number of Reads for Inserting 10, 100, and 1 000 NUPI Duplicate Rows

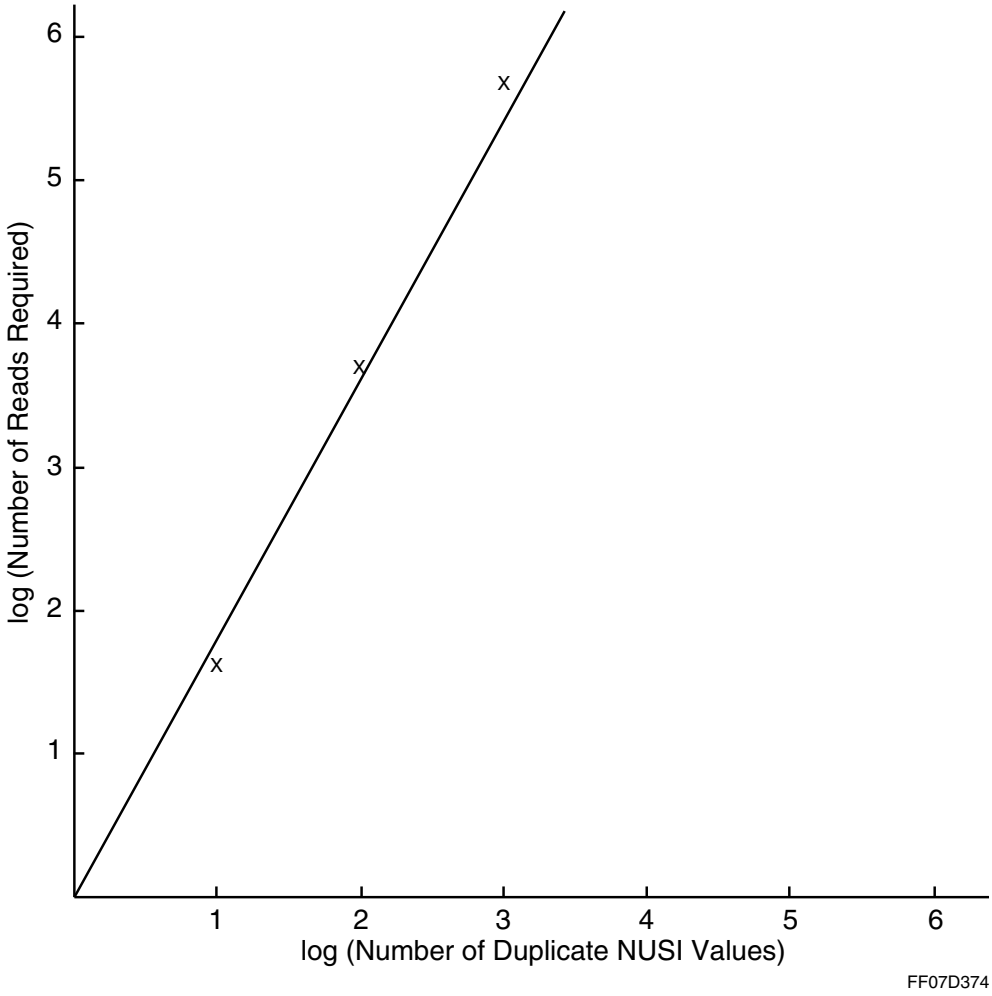
Note that a single order of magnitude increase in the number of duplicate NUPI row inserts into a table, from 10 to 100, results in a two orders of magnitude increase in the cumulative number of reads that must be performed.

When duplicate NUPI inserts increase by another order of magnitude, from 100 to 1 000, there is an additional increase of *four* orders of magnitude over the cumulative number of reads required to insert 10 duplicate NUSI rows.

To insert this many duplicate NUPI rows …		This many cumulative reads must be performed …	
N	$\log_{10}N$	N	$\log_{10}N$
10	1	45	1.6532
100	2	4 950	3.6946
1 000	3	499 500	5.6985

Cumulative Number of Read I/Os as a Function of Duplicate NUPI Inserts

A log-log plot of the data from the above table indicates extremely exponential growth in the cumulative number of read I/Os required to process duplicate NUPIs as a function of the number of duplicate NUPI rows on the AMP.



Minimizing Duplicate NUPI Row Checks

Introduction

There are several ways to minimize or even eliminate duplicate NUPI row checks. This topic describes some of the ways to achieve that goal.

This topic describes the following tactics for reducing the impact of duplicate NUSI row checks.

- Use a non-primary index column set to define a unique constraint.
- Use multiple columns to define NUPIs in order to make the index as singular as possible.
- Keep the number of NUPI duplicate rows for each value below 100.

Eliminating NUPI Duplicate Row Checks

You can eliminate duplicate row checking entirely by defining a USI on the primary key (or some other alternate key that can be constrained uniquely). The details of this technique, including its drawbacks, and described in Chapter 10: "Secondary Indexes," under "Selecting a Secondary Index" on page 10-29.

Optimizing Performance by Using Multicolumn NUPIs

You can minimize duplicate row check performance issues by making your NUPIs as singular as possible. The more singular the NUPI value (that is, the closer to being unique it is), the more likely all rows having that NUPI can be stored within a single data block.

A powerful method for achieving the goal of maximal singularity is to define the NUPI on multiple columns. If you decide to use this approach to enhance the uniqueness of a NUPI, keep in mind that you should also define primary indexes on the *smallest* possible column set (see "Primary Index Value Retrieval Access" on page 9-19). The goal is to optimize the tradeoff between enhancing the generality of the index for row retrieval and reducing the number of duplicate row checks that must be performed.

Drawbacks of this method are described in "Advantages and Disadvantages of Multicolumn NUPIs" on page 9-55.

Consider the following example. Suppose you have a table with three name columns: Last_Name, First_Name, and Middle_Name. You determine that you must use one or more of the name columns as the NUPI for the table.

Step	Action
1	Start with Last_Name. Depending on your demographics, this might be a usable choice. If your demographics indicate that your population has mostly English names, you will probably find numerous Johnsons, Smiths, and Jones among your last name pool. This would provide a fairly skewed distribution of rows across your AMPs.
2	Now add First_Name to Last_Name. Given the same population, you are still likely to find multiple names like Robert Smith or Jennifer Johnson, but the singularity of this NUPI is still greatly enhanced over the single column Last_Name NUPI.
3	Now add Middle_Name to Last_Name and First_Name. You might still find duplicate names like Robert David Smith, but the probability is great that there are fewer Robert David Smiths in your population than there are Robert Smiths. The distribution of rows should be fairly even using this NUPI.

Advantages and Disadvantages of Multicolumn NUPIs

The following table lists the advantages and disadvantages of multicolumn NUPIs.

Advantages	Disadvantages
Singularity is enhanced.	Primary index retrievals are possible only when all NUPI columns are specified in the WHERE clause.
Number of rows per value is lessened.	Partial primary index values cannot be hashed.
Selectivity is enhanced.	Usability is decreased.
Summary Statement	
The more columns defined for a NUPI, the closer it is to being unique.	The more columns defined for a NUPI, the less generalized its use.

Keep the Number of Duplicate NUSI Value Rows Under 100

See "Duplicate Row Checks for NUPIs" on page 9-50 for a detailed explanation of why it is essential to follow this guideline.

Primary Index Uniqueness and Space Utilization

Introduction

The singularity of the primary index for a table is directly related to how evenly table rows are distributed. The more singular the index, the more optimal the use of disk space, with uniqueness producing the most efficient space utilization.

The following mini-case studies illustrate how the degree of singularity of the values in the column selected to be the primary index for a table affects the distribution of rows across the AMPs.

The examples use the following Customer table as their basis. Assignment of the primary index varies from case to case. Only a small sample of table rows is shown. Assume a 6-AMP system.

Customer

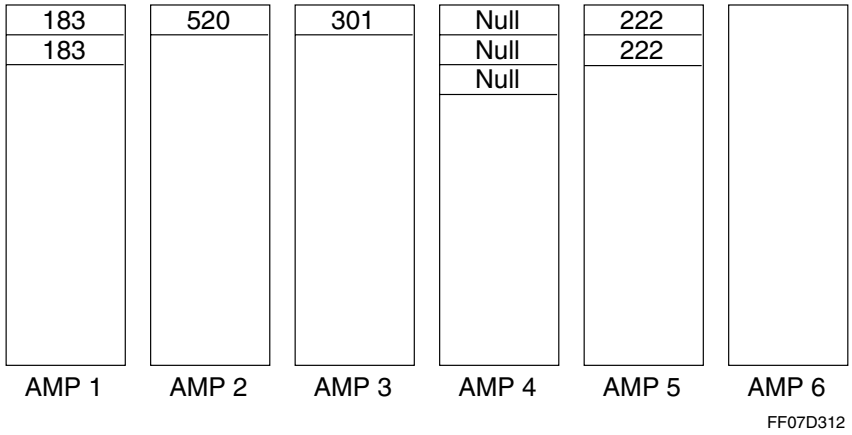
Customer_ Number	Customer_Name	Customer_Status	Parent_ Cust_Number	Sales_ Emp_Number
PK SA			FK	NN
00001	J&R Sales	A	183	1252
00002	Nota Bene Ltd.	I	520	0492
00003	PSI, Inc.	A	301	0655
00004	Vodun and Sons	A	Null	1973
00005	JellyJamUp	A	222	0034
00006	Kimble Korp.	A	Null	0005
00007	Kraus GmbH	I	Null	0708
00008	Piccolo SpA	I	183	0708
00009	The Wiltshire Company	A	222	1439
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…

Case Study 1: Fewer Distinct Primary Index Values Than AMPs

This study defines a NUPI on Parent_Cust_Number, the foreign key. There are only five distinct values for this attribute in the entire Customer table.

Never define a primary index on a column that has fewer distinct values than the number of AMPs in the system unless no other columns are available. Because only five of the 65 536 available hash buckets are used for this table, the very best distribution you can hope for is that each of those five buckets is assigned to a different AMP.

The illustration shows an example of such a best case distribution of rows.

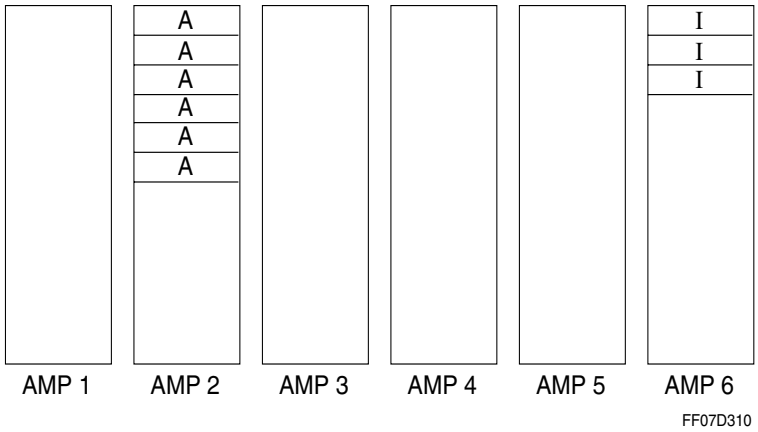


Case Study 2: Highly Nonunique Primary Index Values

This study defines a NUPI on Customer_Status. Because there are only two distinct values for this attribute (Active and Inactive), all rows hash to two AMPs, leaving the other four AMPs in the system unaffected by the Customer table.

Distributions that are skewed like this cause major performance problems because they completely fail to harness the powerful parallel processing capabilities that make Teradata unique among commercially available relational database management systems.

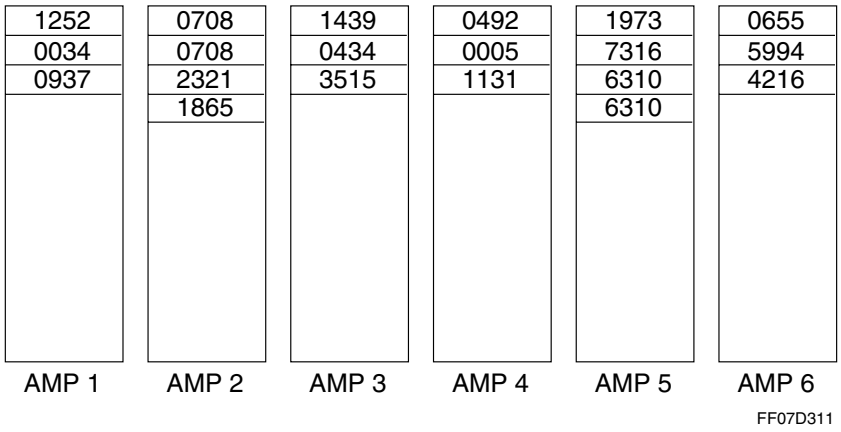
The illustration shows the Customer table rows occupying only two AMPs.



Case Study 3: Highly Singular Primary Index Values

A carefully selected NUPI distributes the rows of a table nearly as evenly as a UPI. If you are fortunate, the NUPI column might never contain duplicates.

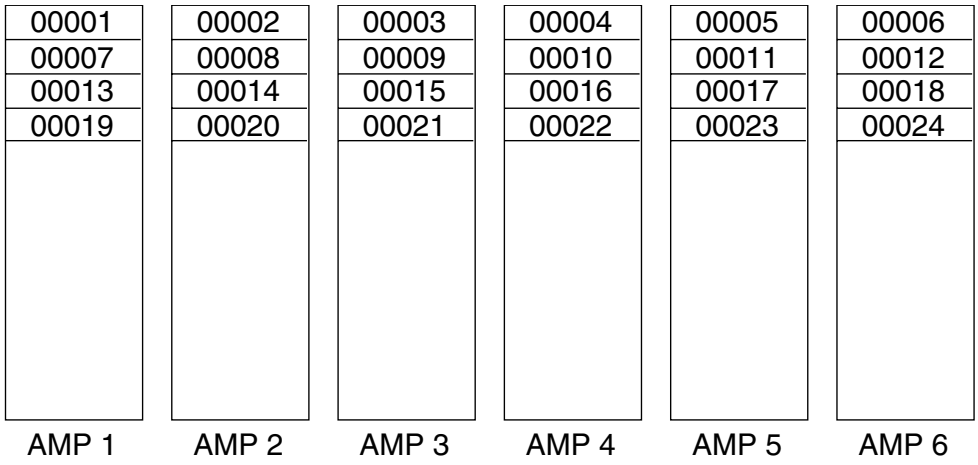
The illustration shows how evenly Customer table rows are hashed when a NUPI is defined on the Sales_Employee_Number attribute.



Case Study 4: Unique Primary Index Values

A UPI always provides even distribution of rows across the AMPs. To ensure efficient space utilization, you should define the primary index for a table on a unique column unless you plan to hash the rows of a minor entity table to the same AMPs as matching rows from a major entity table in order to enhance join processing. Be aware that join indexes might be a better solution, depending on the particular application. See Chapter 11: “Join and Hash Indexes,” for more information.

The illustration shows an ideal distribution of Customer table rows when a UPI is defined on the system-assigned primary key, the Customer_Number attribute.



Chapter 10:

Secondary Indexes

This chapter describes the Teradata secondary indexes, both unique and nonunique.

Processing considerations are given emphasis, as are bit mapping, covering, value-ordered indexes, and the selection of secondary indexes.

Secondary Indexes

Introduction

This topic describes the properties of secondary indexes.

When column demographics suggest their usefulness, the Optimizer selects secondary indexes to provide faster set selection.

A secondary index is never required for Teradata tables, but they can often improve system performance, particularly in decision support environments.

While secondary indexes are exceedingly useful for optimizing repetitive and standardized queries, Teradata is also highly optimized to perform full-table scans in parallel. Because of the strength of full-table scan optimization in Teradata, there is little reason to be heavy-handed about assigning multiple secondary indexes to a table.

Secondary indexes are less frequently included in query plans by the Optimizer than the primary index for the table being accessed.

You can create secondary indexes when you create the table (see "CREATE TABLE" in the chapter "Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*), or you can add them later using the CREATE INDEX statement (see “CREATE INDEX” in the chapter, “Data Definition Language Statement Syntax” in *Teradata RDBMS SQL Reference, Volume 4*). Unlike primary indexes, secondary indexes can be dropped and created at will without altering the table they index.

Data access using a secondary index varies depending on whether the index is unique or nonunique

See “USI Access” on page 10-6 for more information about how Teradata accesses data using a unique secondary index.

See “NUSI Access” on page 10-11 for more information about how Teradata accesses data using a nonunique secondary index.

Purpose

Secondary indexes enhance set selection by specifying access paths less frequently used than the primary index path. Secondary indexes are also used to facilitate aggregate operations.

If a secondary index covers a query (see “NUSIs and Covering” on page 10-25 for a definition of covering), then the Optimizer determines that it would be less costly to accesses *its* rows directly rather than using it to access the base table rows it points to. Sometimes multiple secondary indexes with low individual selectivity can be overlapped and bit mapped to provide enhanced

retrieval when that would decrease the cost of a query (see "NUSI Bit Mapping" on page 10-20).

Distribution of unique secondary index rows to their subtables is controlled using the Teradata hashing algorithm (see "Teradata Hashing Algorithm" on page 8-32).

Nonunique secondary indexes are stored local to the rows they index and are not hash distributed.

Limits

The following restrictions apply to secondary indexes:

- Teradata tables can have up to 32 secondary, hash, and join indexes.
- A secondary index can be defined on anywhere from one to sixteen columns.

Space Considerations

Creating a secondary index causes Teradata to build a subtable to contain its index rows, thus adding another set of rows that requires updating each time a table row is inserted, deleted, or updated.

Secondary index subtables are also duplicated whenever a table is defined with FALLBACK, so the maintenance overhead is effectively doubled.

Unique Secondary Indexes

A secondary index is defined to be either unique or nonunique.

Unique secondary indexes are typically assigned to the primary key of a table whenever its primary index is a NUPI. This is particularly useful for avoiding costly duplicate row checks.

USIs can be assigned to any column having unique values. Note that any column defined with a UNIQUE constraint is implicitly assigned to be a USI.

For more information about USIs, see "Unique Secondary Indexes" on page 10-4.

Nonunique Secondary Indexes

Nonunique secondary indexes are typically assigned to nonunique column sets that frequently appear in WHERE clause selection conditions, join conditions, ORDER BY and GROUP BY clauses, foreign keys, and miscellaneous other conditions such as UNION, DISTINCT, and any attribute that is frequently sorted.

For more information about NUSIs, see "Nonunique Secondary Indexes" on page 10-9.

Unique Secondary Indexes

Introduction

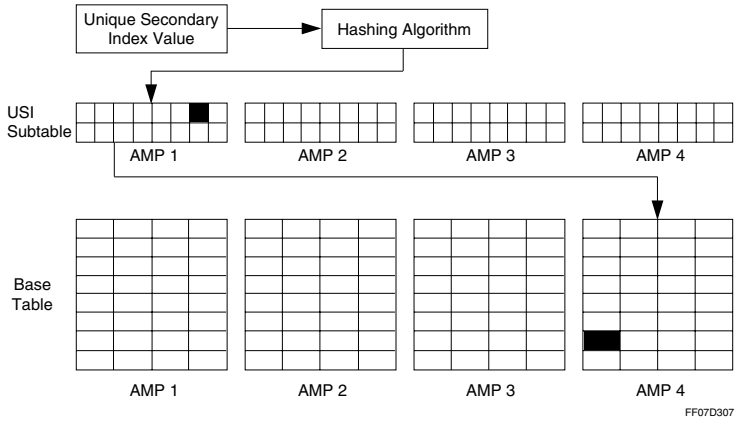
USIs are always preferable to NUSIs for single value access, but might not be as efficient as NUSIs for range access. The principal criterion for choosing between them is the intended application.

USIs are useful both for base table access (because USI access is, at worst, a two-AMP operation) and for enforcing data integrity by applying a uniqueness constraint on a column set. Like a unique primary index, a unique secondary index can be used to guarantee the uniqueness of each value in a column set.

See Chapter 16: "Design Issues for Tactical Queries" for a description of the special design considerations that must be evaluated for using USIs to support tactical queries.

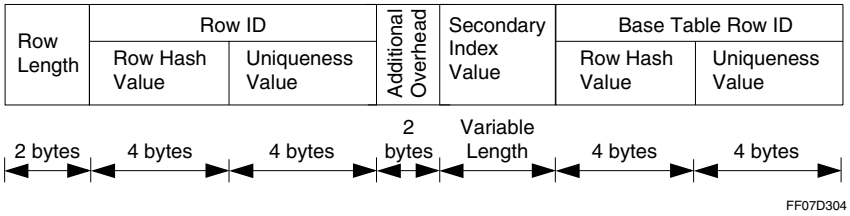
USI Hashing

USIs are hash-partitioned on their index columns, as indicated by the following graphic.



USI Row Layout

The subtable row layout of a USI is described by the following graphic.



Definitions for Row Layout Fields

Stored Data	Length (bytes)	Function
Row length	2	Defines the number of bytes, including overhead, in the row. If the row length is not an even number of bytes, this value defines its precise length, but the file system allocates complete words (an even number of bytes) for each row in the block.
Row ID	8	Defines the USI row uniquely by combining its row hash value with a uniqueness value.
• Row hash	4	Defines the output of the hashing algorithm, which is a unique (or nearly unique) value based on a mathematical transformation of the unique secondary index value.
• Uniqueness value	4	Defines a unique system-generated integer that ensures the uniqueness of the rowID.
Overhead	2	Unused.
Secondary index values	up to 65 524	Defines the column values for the unique secondary index.
Base table row ID	8 for rows in an NPPI table. 10 for rows in a PPI table.	Defines the rowID of the base table row this secondary index row identifies. This row is usually on a different AMP from its unique secondary index rowID.
• Row hash	4	Defines the output of the hashing algorithm on the primary index value.
• Uniqueness value	4	Defines the unique system-generated integer that ensures the uniqueness of the base table row ID.
• Partition number	2	Defines the partition number for a PPI base table row.

Definition for Row Reference Array

Stored Data	Length (bytes)	Function
Row reference array	2	Defines the sequential position of a row within the data block. The row reference array is not part of the row structure: it is a separate structure at the bottom of the data block.

USI Access

USI access is usually a two-AMP operation because a USI typically hashes to a different AMP than the PI for the same row. If the USI subtable row hashes to the same AMP as the base table row it points to, then only one AMP is accessed.

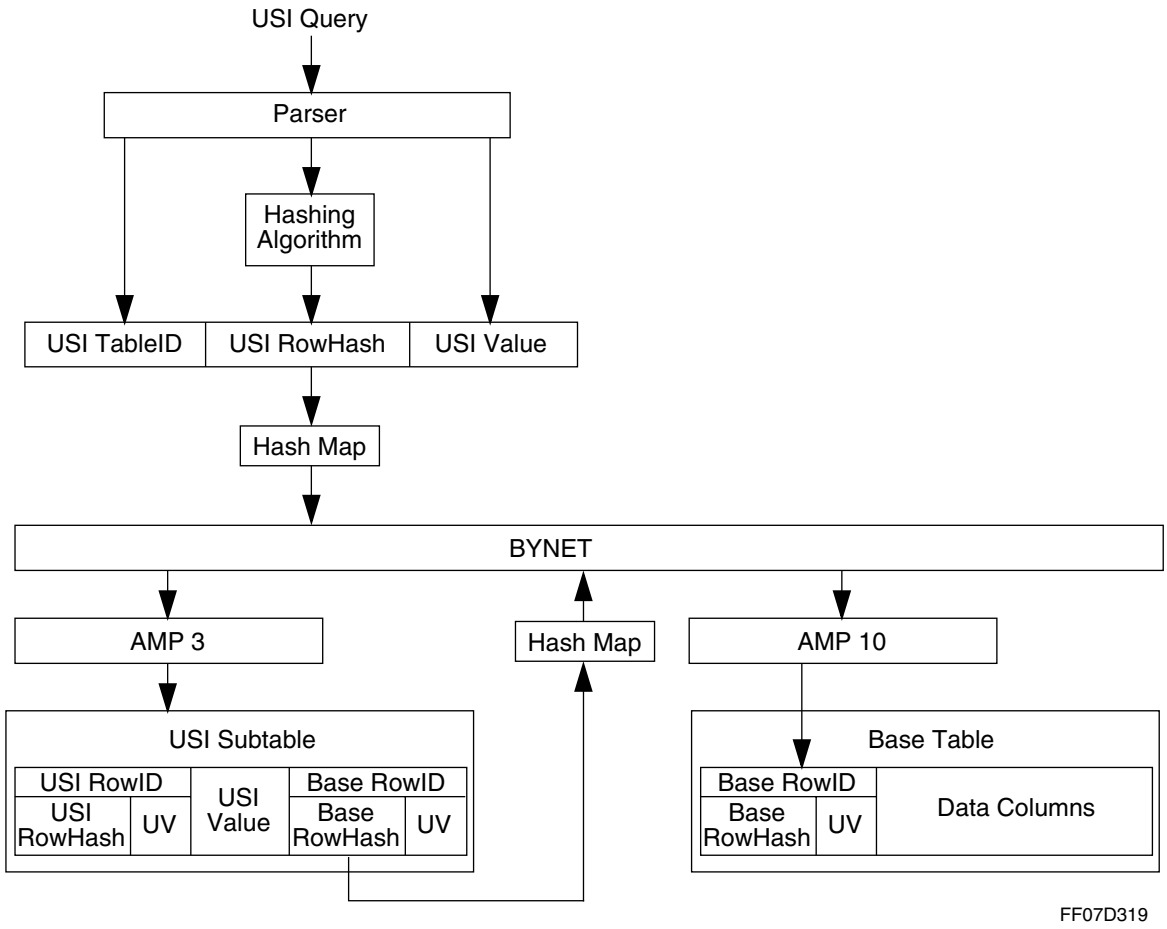
The following stages are involved in a USI base table row access.

Stage	Process
1	The requested USI value is accessed by hashing to its subtable.
2	The pointer to the base table row is read and used to access the stored row directly.

USI access specifies a three-part BYNET message that is identical to the three-part message used for primary index access except that the subtable ID in the message references the USI subtable rather than the base table.

The flow diagram illustrates the following query.

```
SELECT *
FROM table_name
WHERE USI_column = value;
```



The process for locating a row using a USI is as follows.

Stage	Process
1	After checking the syntax and lexicon of the query, the Parser looks up the Table ID for the USI subtable that contains the specified USI value.
2	The hashing algorithm hashes the USI value.
3	The Generator (see "Definitions for Parsing Engine Software" on page 8-2 for a definition of the Generator) creates an AMP steps message containing the USI Table ID, USI row hash value, and USI data value.
4	The Dispatcher (see "Definitions for Parsing Engine Software" on page 8-2 for a definition of the Dispatcher) uses the USI row hash to send the message across the BYNET to AMP 3, which contains the appropriate USI subtable row.
5	The file system on AMP 3 locates the appropriate USI subtable using the USI Table ID.
6	The file system on AMP 3 uses the USI row ID to locate the appropriate index row in the subtable.

Stage	Process
7	The file system reads the base table row ID from the USI row and distributes a message containing the base table ID and the row ID for the requested row across the BYNET to AMP 10, which contains the requested base table row.
8	The file system uses the row ID to locate the base table row.

Example

The following example performs a single row lookup. The column named Cust is a USI for the customer table.

```
SELECT name, phone
FROM customer
WHERE cust = 3;
```

The example Customer table and USI subtable are as follows.

USI Subtable

rowID	Cust	rowID
	USI	
DQ + 1	1	H6 + 1
VP + 1	5	J5 + 1
R9 + 2	3	B4 + 1
R9 + 1	13	N6 + 1
3J + 1	2	E3 + 1
22 + 1	14	E3 + 3
3S + 1	9	D7 + 2

Customer

rowID	Cust	Name	Phone
	PK		
	USI		
B4 + 1	3	Brown	444-3333
A2 + 1	7	Black	333-4444
N6 + 1	13	Rice	888-9999
E3 + 1	2	James	555-4444
E3 + 3	14	Brown	555-4444
L2 + 1	10	Smith	222-9999
J5 + 1	5	Smith	444-6666

The secondary index value (Cust_Num) is hashed to generate rowID R9 (note that +2 represents the uniqueness number). The AMP retrieves row R9+2 from the secondary index subtable. The subtable row contains the rowID of the base table row, which can then be accessed.

Nonunique Secondary Indexes

Introduction

NUSIs are particularly useful for range access and nonequality conditions.

Highly selective NUSIs are useful for reducing the cost of frequently made selections and joins on nonunique columns, but NUSIs with low selectivity can be less efficient than a full-table scan.

NUSIs are a less preferable secondary index choice for other applications for several reasons.

- NUSI access is always an all-AMPs operation.
- Because NUSI subtable access is not hashed, the subtables must be scanned in order to locate the relevant pointers to base table rows.

NUSI Requests Are Not Hashed

Because NUSIs are created and stored locally on the AMPs, NUSI requests are not hashed; instead, are broadcast to all AMPs in the configuration.

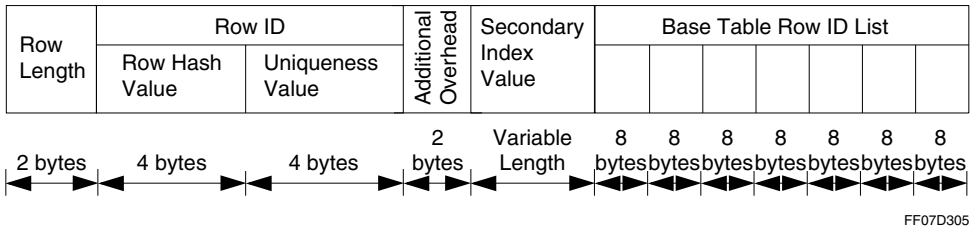
A NUSI Subtable Row Points to One or Many Base Table Rows

A particular NUSI subtable row points to one or many base table rows on that same AMP. The relationship between a NUSI value and any individual AMP in a configuration is either 0:1 or 1:1.

This relationship ...	Reflects the fact that ...
0:1	not all AMPs contain a NUSI subtable index row for a particular NUSI value.
1:1	no more than one instance of a NUSI value is ever stored on an AMP.

NUSI Row Layout

The subtable layout of a NUSI row is described by the following graphic.



The number of base tables that can be referenced is limited by the maximum row size for the system and the length of the secondary index value. See "NUSI Sizing Equation" on page 14-92 for more information about sizing NUSI subtables.

Definitions for Row Layout Fields

Stored Data	Length (bytes)	Function
Row length	2	Defines the number of bytes, including overhead, in the row. If the row length is not an even number of bytes, this value defines its precise length, but the file system allocates complete words (an even number of bytes) for each row in the block.
Row ID	8	Defines the NUSI row uniquely by combining its row hash value with a uniqueness value.
• Row hash	4	Defines the output of the hashing algorithm, which is a unique (or nearly unique) value based on a mathematical transformation of the nonunique secondary index value.
• Uniqueness value	4	Defines a unique system-generated integer that ensures the uniqueness of the Row ID.
Overhead	2	Unused.
Secondary index values	up to (65 524) - (table row ID list bytes)	Defines the column values for the nonunique secondary index.
Table row ID list	8 per base table row	Defines the row IDs for the rows on the same AMP to which this nonunique index points.

Definition for Row Reference Array

Stored Data	Length (bytes)	Function
Row reference array	2	Defines the sequential position of a row within the data block. The row reference array is not part of the row structure: it is a separate structure at the bottom of the data block.

NUSI Access

NUSI access specifies a three-part BYNET message that is identical to the three-part message used for primary index access except that the subtable ID in the message references the NUSI subtable rather than the base table.

NUSI access is always an all AMPs operation. That might make it seem that NUSIs have limited value. If you have to access all AMPs in the configuration to locate the requested rows, why bother with an index?

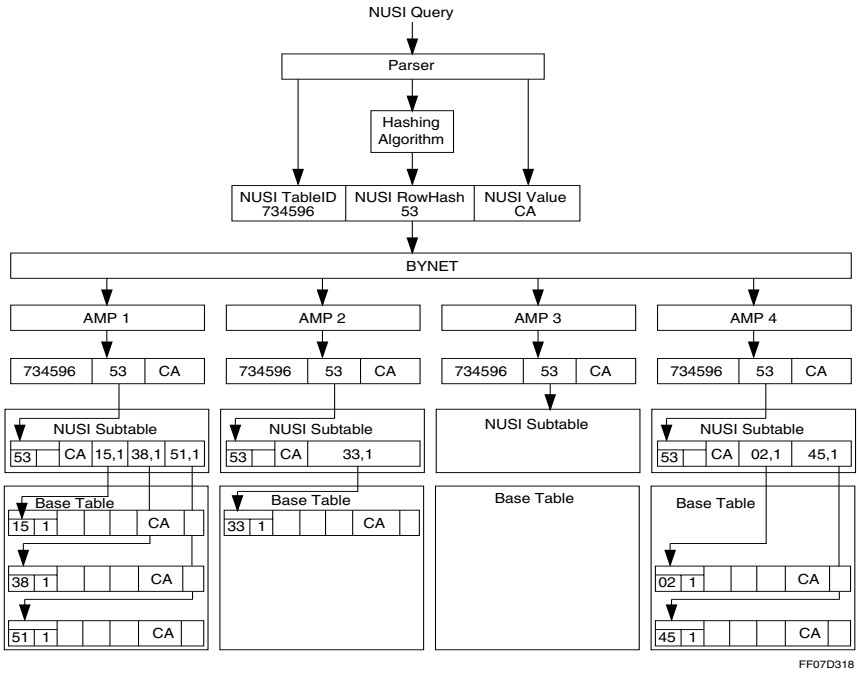
The answer to that question is provided by the following list of ways that NUSIs can improve the performance of your decision support queries beyond the usual vague explanation that they improve set selection.

- NUSI access is often markedly faster than a full-table scan, particularly for extremely large tables.
- A NUSI that covers (see "NUSIs and Covering" on page 10-25 for a definition of covering) the columns requested by a query is often included in Optimizer access plans.
- A single column NUSI that covers a LIKE expression is often included in Optimizer access plans.

At the same time, as the number of rows pointed to per NUSI value increases, the efficiency of a NUSI read decreases proportionately. Depending on demographics and environmental cost variables, the Optimizer will specify a full-table scan instead of a NUSI access when it determines that the scan would be a more efficient access method.

The flow diagram illustrates the following query.

```
SELECT *
FROM table_name
WHERE NUSI_column = 'CA';
```



The process used by this example for locating a row using the NUSI value CA is as follows.

Stage	Process
1	After checking the syntax and lexicon of the query, the Parser looks up the Table ID for the NUSI subtable that contains the NUSI value ‘CA’.
2	The hashing algorithm hashes the NUSI value.
3	The Generator creates an AMP steps message containing the NUSI Table ID (734596), NUSI row hash value (53), and NUSI data value (CA) and then the Dispatcher distributes it across the BYNET to all AMPs.
4	The file system on a receiving AMP locates the appropriate NUSI subtable using the NUSI Table ID.
5	The file system on a receiving AMP uses the NUSI row hash value to locate the appropriate index row in the subtable.
6	If there is a NUSI row, its table row ID list is scanned for base table row IDs.
7	The file system uses the row IDs to locate the base table rows containing the NUSI value ‘CA’.

Selectivity Considerations

Selectivity refers to the percentage of rows in a table containing the same nonunique secondary index value. An index that has high selectivity retrieves few rows. A primary index retrieval, for example, is highly selective because it never returns more than one row. An index that has low selectivity retrieves many rows.

Low Selectivity Index

When an index is said to have low selectivity, that means that many rows have the same NUSI value and there are relatively few distinct values.

A column with those characteristics is usually a poor choice for a NUSI because the cost of using it may be as high or higher than a full-table scan.

For example, assume that Employee table contains 10 000 rows of about 100 bytes each, and there are only 10 different departments. If an average employee data block is 2 560 bytes and can store about 25 rows, then the entire table requires about 400 data blocks.

If DeptNo is defined as a nonunique secondary index on the Employee table, and the DeptNo values are evenly distributed, then the following query accesses about 1 000 row selections.

```
SELECT *
FROM Employee
WHERE DeptNo = 300;
```

Each AMP reads its own rows of the DeptNo secondary index subtable. If any rows contain the index value 300, the AMP uses the appended rowIDs to select the data rows from its portion of the Employee table.

Regardless of the number of AMPs involved, this retrieval requires 1 000 row selections from the Employee table. To satisfy this number of select operations, it is likely that all 400 Employee data blocks would have to be read.

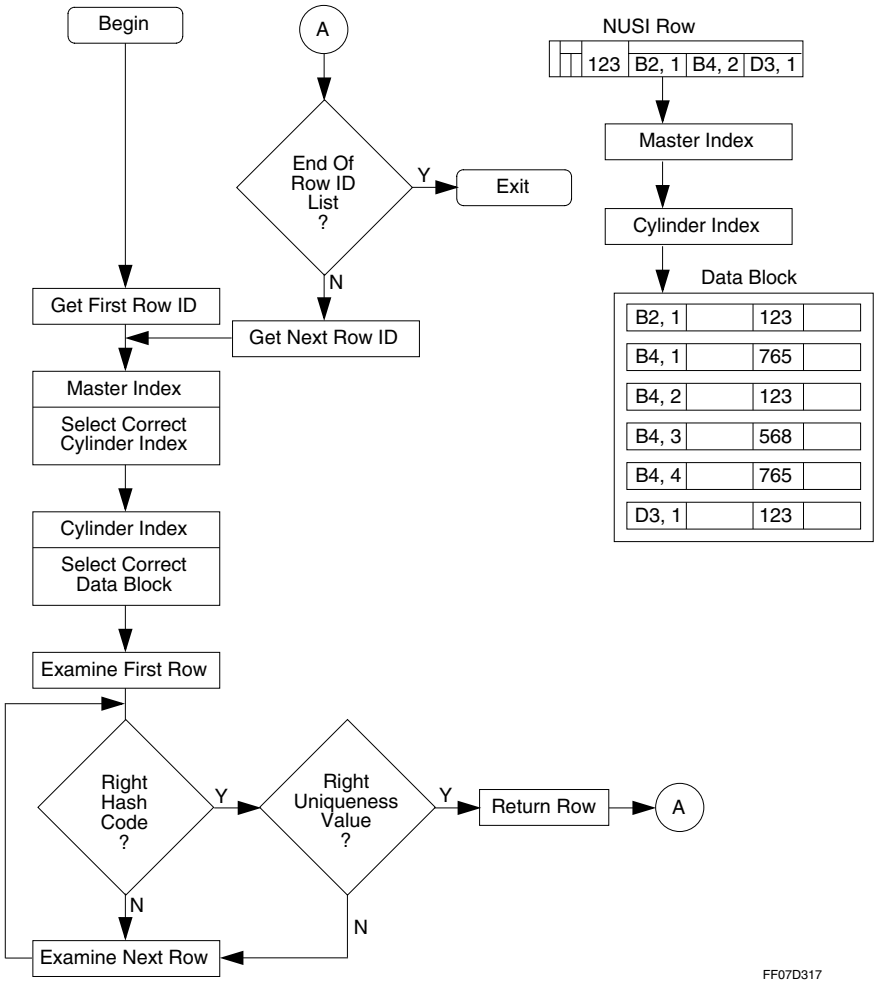
If that were the case, then the number of I/O operations undertaken by the retrieval could easily exceed the number required for a full-table scan. In such instances, a table scan would actually be a much more efficient solution than a NUSI-based retrieval.

High Selectivity Index

If DeptNo is a high selectivity index, where few Employee rows share the same DeptNo value, then using DeptNo for retrieval provides better performance than a full-table scan. Because of these selectivity conditions, the Optimizer specifies NUSI access for request processing when it is less costly than a full-table scan.

Generic NUSI Read Process Flow

The following flowchart illustrates the logic of a NUSI read. The graphic also illustrates a somewhat more concrete NUSI read example where the rowID value B2,1 (where B2 is the row hash value and 1 is the uniqueness value for the row) read is followed in turn through the master index, the cylinder index, and the data block to access the base table row set matching that value.



The following process table describes the process flow for locating a row using a NUSI.

Stage	Process
1	Read the first rowID from the subtable row. In the graphic example, the value read is B2, 1.
2	Scan the Master Index for the required Cylinder Index.
3	Scan the Cylinder Index for the required data block.
4	Scan the rows in the data block until the first row having the requested rowID is located.
5	Continue to scan the rows for additional requested rows until a new rowID is found. Because the rows are stored logically in rowID order, the search process terminates as soon as a different rowID is detected.
6	Return the retrieved row set to the requestor.

Dual NUSI Access

Introduction

Database designers frequently define multiple NUSIs on a table. Whether the Optimizer chooses to include one, all, or none of them in its query plan depends entirely on their individual and composite selectivity.

Dual NUSI Access Compared With Composite NUSI Access

Dual secondary indexes are a good choice when their combination is highly selective: the number of rows returned is a small fraction of the total number of rows in the table.

For example, consider the following query.

```
SELECT last_name, first_name, salary_amount
FROM employee
WHERE department_number = 500
AND job_code = 2147;
```

The conditions have NUSIs defined on them.

```
CREATE INDEX (department_number) ON EMPLOYEE;
CREATE INDEX (job_code) ON EMPLOYEE;
```

IF ...	THEN the Optimizer uses ...
both indexes have low selectivity	bit mapping (see "NUSI Bit Mapping" on page 10-20).
one index has higher selectivity than the other	the index with higher selectivity to select the rows and then uses the index with lower selectivity to qualify the accessed rows.

IF ...	THEN the Optimizer ...
both columns are indexed	most often specifies a full-table scan of the index subtables, calling for a non-duplicate spool file of the rowIDs, and then uses the result to access the base table.
one column is indexed	always specifies a full-table scan.

Because OR conditions specify separate set selections, such a query might execute faster if done using a UNION operation instead.

ANDed expressions in a WHERE clause represent a set selection.

A composite set selection is one for which multiple logically related conditions apply.

An ORed composite set selection is one for which *any* predicate condition in the WHERE clause can evaluate to TRUE in order to retrieve rows that match the condition specified by the predicate.

An ANDed composite set selection is one for which *all* predicate conditions in the WHERE clause must evaluate to TRUE. If any condition evaluates to FALSE, then no row is retrieved for that composite set of conditions.

A composite condition set can reference any number of indexes, including none. The Optimizer selects an index for the query plan only when values for all of its components are supplied in the condition.

When a condition references an indexed column set, then the Optimizer selects that index for the query plan and the Database Manager uses it to access the table. Any remaining, or residual, conditions are applied to the intermediate result set to accomplish additional selectivity.

The following table explains how the Optimizer uses secondary indexes in the same scenario.

The Optimizer uses …	**WHEN** …
a single secondary index	its selectivity is high (returning fewer than 10 percent of the table rows).
multiple secondary indexes	their combined selectivity is high.

Example NUSIs

Consider the following two NUSIs created on an Employee table.

```
CREATE INDEX (department_number) ON employee;
CREATE INDEX (job_code) ON employee;
```

The examples "AND Equality Condition" on page 10-17 and "OR Equality Condition" on page 10-18 examine how these dual NUSIs on the Employee table might be used by the Optimizer when they are used as ANDed and ORed conditions in a WHERE clause.

AND Equality Condition

Consider the following simple query with an ANDed predicate set.

```
SELECT last_name, first_name, salary_amount
FROM employee
WHERE department_number = 500
AND job_code = 2147;
```

Whether the Optimizer includes these NUSIs in its query plan varies depending on their relative selectivity.

IF ...	THEN the Optimizer ...
only one of the NUSIs is strongly selective	uses that NUSI alone in its query plan.
both NUSIs are weakly selective individually, but strongly selective when combined	creates a bit-mapped intersection of their common rowIDs and uses them in its query plan (see "NUSI Bit Mapping" on page 10-20).
both NUSIs are weakly selective both when examined individually and when combined	selects neither for its query plan and instead specifies a full-table scan.

OR Equality Condition

Consider the following simple query with an ORed predicate set.

```
SELECT last_name, first_name, salary_amount
FROM employee
WHERE department_number = 500
OR job_code = 2147;
```

If both columns are indexed, as they are in this particular case, then a likely Optimizer plan would include the following steps.

Stage	Process
1	Perform a full subtable scan on both NUSI subtables.
2	Retrieve the rowIDs of any qualifying base table rows and place them into separate spool files.
3	Eliminate rowIDs that are duplicates between the spools and then merge them into a single spool file.
4	Use the resulting spool to access the qualified base table rows.

This processing could be extended to more than two NUSIs, of course, but the likelihood of that being a profitable exercise for the Optimizer to undertake vanishes rapidly as the number of NUSIs increases. A full-table scan would nearly always take less time to perform than the NUSI subtable scans spool file production, duplicate elimination, and merging required by this method.

Queries Using BETWEEN, LESS THAN, or GREATER THAN Operators

Assume that a query involves BETWEEN, LESS THAN, or GREATER THAN operations on a single NUSI as in the following examples.

```
CREATE INDEX (hire_date) ON employee;

SELECT last_name, first_name, hire_date
FROM employee
WHERE hire_date BETWEEN 880101 AND 881231;

SELECT last_name, first_name, hire_date
FROM employee
WHERE hire_date < 880101;

SELECT last_name, first_name, hire_date
FROM employee
WHERE hire_date > 881231;
```

The following table outlines the approach taken to respond to these requests.

Stage	Process
1	The Optimizer determines whether it is more efficient to do a full-table scan of the base table rows or to scan the secondary index subtable to get the rowIDs of the qualifying base table rows. For purposes of this example, assume that the NUSI access method is the more efficient technique to pursue.
2	The access plan specifies to place those rowIDs into a spool file.
3	The access plan specifies to use the resulting rowIDs to access the base table rows.

NUSI Bit Mapping

Introduction

Bit mapping is a technique used by the Optimizer to effectively link several weakly selective indexes in a way that creates a result that drastically reduces the number of base rows that must be accessed to retrieve the desired data. The process determines common rowIDs among multiple NUSI values by means of the logical intersection operation. The method is explained in more detail in "Computing NUSI Bit Maps" on page 10-22.

Performance Advantages of NUSI Bit Mapping

Bit mapping is significantly faster than the three-part process of copying, sorting, and comparing rowID lists. Additionally, the technique dramatically reduces the number of base table I/Os required to retrieve the requested rows.

Restrictions for When Bit Mapping Is Used

Teradata only performs NUSI bit mapping when weakly selective indexed conditions are ANDed. When WHERE conditions are connected by a logical OR, the Optimizer typically performs a full-table scan and applies the ORed conditions to every row without considering an index.

IF all conditions are ANDed together and …	AND their composite selectivity is …	THEN the Optimizer…
one index is strongly selective		selects it alone and applies the remaining constraints to the selected rows as residual constraints.
all of the indexes are weakly selective	also weakly selective	performs a full-table scan and applies the conditions to every row.
all of the indexes are weakly selective	strongly selective	can instruct each AMP to construct bit maps to determine which rowIDs their local NUSI rows have in common and then access just those rows, applying the conditions to them exclusively.

For example, consider the following SELECT statement with three WHERE conditions, each of which has a weakly selective NUSI defined on its column.

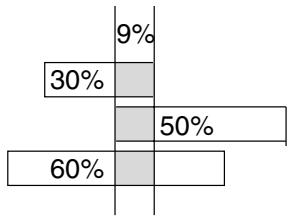
```
SELECT *
FROM employee_table
WHERE salary_amount > 20000
AND   sex_code = 'M'
AND   full_time = 'FT';
```

Suppose that the index on salary_amount has 30 percent selectivity, the index on sex_code 50 percent selectivity, and the index on full_time 70 percent selectivity. The individual selectivity of these indexes is very weak.

If the overlap among their selectivities is such that the effect of combining their relative weaknesses results in a significant strength, then the Optimizer instructs the AMPs to use bit mapping to do just that, and the end result is vastly improved performance of the retrieval in comparison with a full-table scan.

In this example, the overlap selectivity is 9 percent, which is both significantly better than any of the individual selectivities of the NUSIs in the query and typically better than a full-table scan.

```
SELECT *
  FROM  employee_table
    WHERE   salary_amount  > 20000
      AND   sex_code  = 'M'
      AND   full_time    = 'FT' ;
```



FF07D346

Using NUSIs For Complex Conditional Expressions

Complex conditional expressions can be based on nonunique secondary indexes. Bit mapping is often used to solve such expressions when they are applied to a very large table.

Take the following request as an example:

```
SELECT COUNT(*)
FROM LargeTable
WHERE NUSI_1 = 'condition_1'
AND NUSI_2 = 'condition_2'
AND NUSI_3 = 'condition_3'
AND NUSI_4 = 'condition_4' ;
```

To resolve this request, a bit map is built for each referenced index. In each bit map, every qualifying data row bit is turned ON. The bit maps are ANDed to form one large bit map, which is held in a spool (temporary) file. The ON bits are used to access the result data rows directly.

Computing NUSI Bit Maps

We use the following previously described query to example how NUSI bit maps are computed.

```
SELECT *
FROM employee_table
WHERE salary_amount > 20000
AND sex_code = 'M'
AND full_time = 'FT';
```

Recall the truth table for the AND operator.

0 + 0 = 0

0 + 1 = 0

1 + 0 = 0

1 + 1 = 1

The following table illustrates the method for computing NUSI bit maps. The actions described occur concurrently on all AMPs.

Stage	Process
1	The literals 20 000, M, and FT are used to access the corresponding rows from the index subtables.
2	For *n*-1 indexes, each AMP creates separate bit maps of all 0 values. In this case, *n*-1 is 2 because there are three NUPIs defined, one for each condition in the WHERE clause.
3	Using a proprietary algorithm, the AMPs equate each base table rowID in each qualifying NUPI row with a single bit in the map, and each map bit is turned ON.
4	The resulting bit maps are ANDed together to form the final bit map.
5	Each base table rowID in the remaining NUSI is equated to a single bit in the final bit map, and the state of that map is checked.

IF the map bit is …	THEN …
OFF (0)	no action is taken and the next map bit is checked.
ON (1)	the rowID is used to access the table row and all remaining, or residual, conditions are applied to that row.

How To Determine If Bit Mapping Is Being Used

To determine whether bit mapping is being used for your NUSIs, use the EXPLAIN statement and examine the reports it produces.

Example

The following example illustrates the bit mapping process.

The table queried has the following set of values.

Base Table

RowID	column_1	column_2	column_3
12,07	A	B	C
22,03	A	C	C
48,12	A	B	D
63,15	B	B	C
…	…	…	…

The query ANDs three equality conditions on three NUSI columns in the WHERE clause.

```
SELECT *
FROM Base_Table
WHERE column_1 = 'A'
AND column_2 = 'B'
AND column_3 = 'C';
```

The Optimizer evaluates the query and determines that bit mapping over the three columns is the least costly access plan.

The truth table looks like this.

NUSI Column Number	Value Required to Satisfy Condition	Base Table RowID List				
		12,07	22,03	48,12	63,15	...
1	A	1	1	1	0	…
2	B	1	0	1	1	…
3	C	1	0	0	0	…
ANDed Sum		1	0	0	0	…

Of the four rows examined in the example, only the row identified by rowID 12,07 qualifies and is returned to the requesting application.

Further Information

Also see the topics “EXPLAIN Modifier” and EXPLAIN and Indexed Access” in the chapter “SQL Data Manipulation Language Statement Syntax” in *Teradata RDBMS SQL Reference, Volume 6*.

NUSIs and Covering

Introduction

The Optimizer aggressively pursues NUSIs when they cover a query. The expression “covering” means that all of the columns requested in a query are also available from an existing index subtable, making it unnecessary to access the base table rows to complete the query.

The Optimizer also selects a secondary index to process an aggregation on indexed values if the index covers the necessary values.

If an aggregate join index covers the query, the Optimizer will select it for the access plan before considering a secondary index. See Chapter 11: “Join and Hash Indexes” for more information about aggregate join indexes.

Column Candidacy for Covered Access

The Optimizer selects covering for its query plans even when classical indexing techniques do not apply. This is because scanning an index is almost always faster than scanning the base table it references, and the enhanced efficiency sometimes results in significantly faster retrieval.

Example 1: Simple Query Considered for Index Covering

The following example demonstrates a query considered for index covering:

```
CREATE INDEX IdxOrd (o_orderkey, o_date, o_totalprice)
ON ORDERS;

SELECT o_date, AVG(o_totalprice)
FROM ORDERS
WHERE o_orderkey >1000
GROUP BY o_date;
```

Example 2: Aggregate Query Considered for Index Covering

An index is defined on DeptNo. The Optimizer selects that index and instructs the AMPs to aggregate on its values rather than undertaking the less performant access to the underlying base tables.

```
SELECT DeptNo, COUNT(*)
FROM Employee
GROUP BY DeptNo;
```

Example 3: Aggregate Query Considered for Index Covering

This example shows how index values can be manipulated arithmetically as well as being counted.

```
SELECT SUM(Salary_Amount)
FROM Employee
GROUP BY Salary_Amount;
```

Value-Ordered NUSIs and Range Conditions

Introduction

Value-ordered NUSIs are very efficient for range conditions. Because the NUSI rows are sorted by data value, it is possible to search only a portion of the index subtable for a given range of key values.

Limitations on Value-Ordered NUSIs

Value-ordered NUSIs have the following limitations.

- The sort key is limited to a single numeric column.
- The sort key column cannot exceed four bytes.

Typical Uses of Value-Ordered and Hash-Ordered NUSIs

The typical use of a hash-ordered NUSI is with an equality condition on the secondary index column set. The specified secondary key value is hashed, and then each NUSI subtable is probed for rows having the same row hash. For every matching NUSI entry, the corresponding rowIDs are used to access the base rows on the same AMP. Because the NUSI rows are stored in row hash order, searching the NUSI subtable for a particular row hash value is very efficient.

Value-ordered NUSIs, on the other hand, are useful for processing range conditions and conditions with an inequality on the secondary index column set.

The following query is an example of the sort of search conditions for which value-ordered NUSIs were designed.

```
SELECT *
FROM Orders
WHERE o_orderdate
BETWEEN '1998-10-01'
AND '1998-10-07';
```

Although hash-ordered NUSIs can be selected by the Optimizer to access rows for range conditions, a far more common response is to specify a full-table scan of the NUSI subtable to find the matching secondary key values. Therefore, depending on the size of the NUSI subtable, this might not be very efficient.

By sorting the NUSI rows by data value, it is possible to search only a portion of the index subtable for a given range of key values. The major advantage of a value-ordered NUSI is in the performance of range queries.

Example of a Value-Ordered NUSI

The following example illustrates a value-ordered NUSI (defined by an ORDER BY clause that specifies the VALUES keyword option) and a query that would probably be solved more efficiently if the specified value-ordered NUSI were selected by the Optimizer to access the requested rows.

```
CREATE INDEX Idx_Date (o_orderdate)
ORDER BY VALUES (o_orderdate)
ON Orders ;

SELECT *
FROM Orders
WHERE o_orderdate
BETWEEN '1997-10-01'
AND '1997-10-07';
```

Selecting a Secondary Index

Introduction

When assigning columns to be a secondary index for a table, there are numerous factors to consider, the most important of all being the selectivity of the index.

While USI retrievals are always very efficient, the efficiency of NUSI retrievals varies greatly depending on their selectivity.

You can use the Teradata Index Wizard client utility to assist your selection of secondary indexes for a table. The utility recommends a set of indexes based on specific query workloads you supply. Teradata Index Wizard can also validate its recommendations to ensure they actually provide the benefits expected. See *Teradata Index Wizard User Guide* for further information about how to use the utility and the chapter "Database Foundations for the Teradata Index Wizard" in *Teradata RDBMS SQL Reference, Volume 2* for information about how the utility performs its component tasks.

Optimal Data Access

Selectivity is a relative term that refers to the number of rows returned by an index. Most retrievals aim to return only a select few rows: very specific answers in response to a very specific request.

An index that returns a small number of rows is said to be highly selective. This is a positive attribute.

Indexes that return a large number of rows are said to have low selectivity. This is generally a negative attribute; so negative that, as often as not, the Optimizer selects a full-table scan over a low selectivity NUSI because the full-table scan can be less costly.

All UPIs and USIs are highly selective by definition, as are most well-chosen NUPIs. High selectivity is favored not only because of its precision, but also because of its low cost, involving a very small number of disk I/Os, which is always a performance-enhancing attribute.

Criteria for Selecting a Secondary Index

The following rules of thumb and performance considerations apply to selecting a unique or nonunique column set as a secondary index for a table.

- Consider assigning a USI to the primary key of any table built with a NUPI. This both enforces uniqueness, eliminating the burden of making row uniqueness checks, and enhances retrieval for applications where the primary key is frequently used as a selection or join criterion.

  A primary key USI might *not* be a good decision for a table that is frequently updated by OLTP applications.
- Consider building secondary indexes on column sets frequently involved in the following clauses, predicates, and other logical operations.
  - Selection criteria
  - Join criteria
  - ORDER BY clauses
  - GROUP BY clauses
  - Foreign keys (because of join and subquery processing)
  - UNION, DISTINCT, and other sort operations

  When these operations act on well-indexed column sets, the number of scans and sorts that must be performed on the data by the database manager can be greatly reduced.
- Consider creating covering indexes wherever possible. The Optimizer frequently selects covering indexes to substitute for a base table access whenever the overall cost of the query plan is reduced. Such index-only access promotes faster retrievals.

  Many applications are well served by join indexes, which can be used profitably in many covering situations where multiple columns are frequently joined. See Chapter 11: “Join and Hash Indexes” for further information about join indexes.
- Consider creating secondary indexes on columns frequently operated on by built-in functions such as aggregates.
- Avoid assigning secondary indexes to frequently updated column sets.
- Avoid assigning secondary indexes to columns with lumpy distributions because there is a slight chance the Optimizer might mistake their usefulness.
- Avoid creating excessive secondary indexes on a table, particularly for a table used heavily, or even moderately, for OLTP processing. The less frequently the table is updated, the more desirable a multiple index solution.

- Plan to dynamically drop and recreate secondary indexes to accommodate specific processing and performance requirements such as bulk data loading utilities, database archives, and so on.

  Create appropriate macros to perform these drop and create index operations if you need to undertake such specific processing tasks regularly.
- Consider naming secondary indexes whenever possible using a standard naming convention.
- Ensure that your indexes are being used as planned by issuing EXPLAIN statements to audit index selection for those queries they were designed to facilitate.

  Indexes that are never selected by the Optimizer are a burden to the system for the following reasons:
  - They consume disk resources that could profitably be used to store data that *is* used.
  - They degrade update processing performance unnecessarily.

Secondary Index Access Summarized by Example

Introduction

This example indicates how a series of queries against a table can use various secondary indexes to access the rows in that table.

Configuration

The system used for this demonstration has four AMPs.

Table Definition

The table used in this demonstration is a simple three-column Customer table, defined as follows:

Column Name	Attribute Described	Type of Index Defined on the Column
Cust	Customer Number	USI
Name	Customer Last Name	NUSI
Phone	Customer Phone Number	NUPI

The following is a snapshot instance of this table:

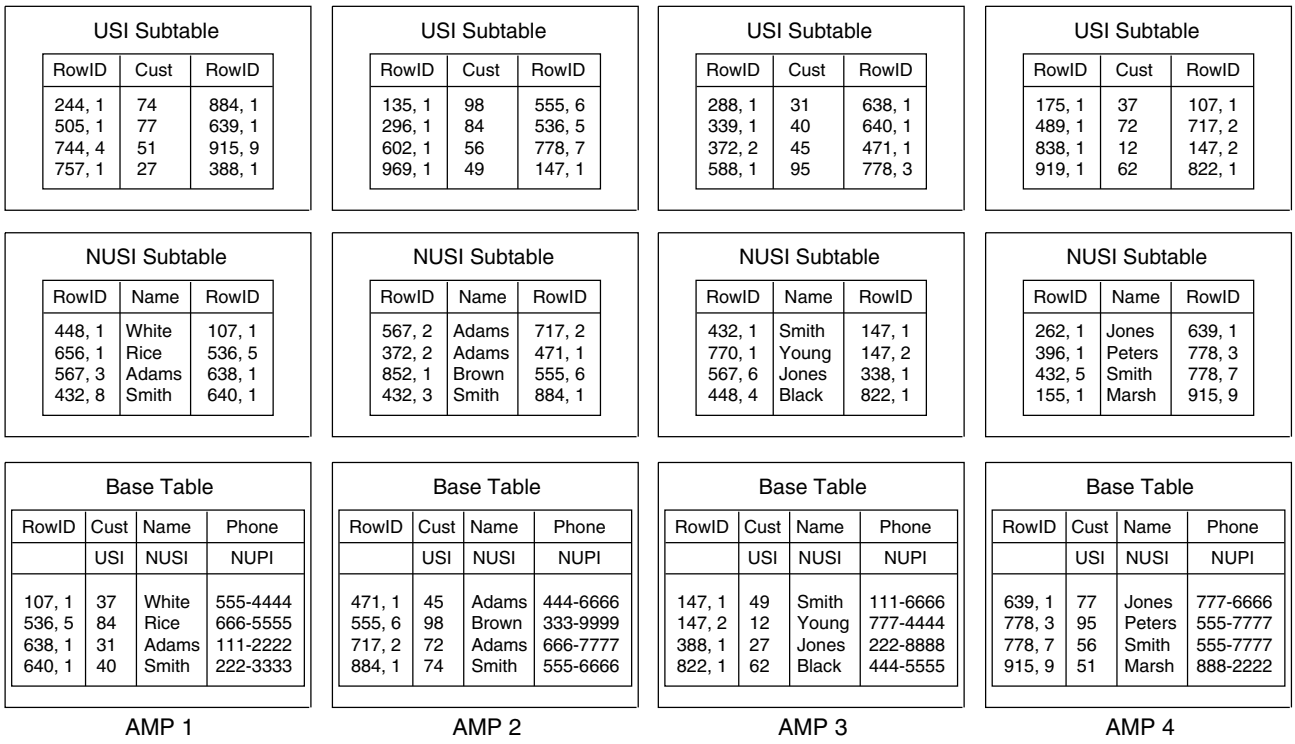
Customer

Cust	Name	Phone
USI	**NUSI**	**NUPI**
37	White	555-4444
98	Brown	333-9999
74	Smith	555-6666
95	Peters	555-7777
27	Jones	222-8888
56	Smith	555-7777
45	Adams	444-6666
31	Adams	111-2222
40	Smith	222-3333

Customer

Cust	Name	Phone
USI	**NUSI**	**NUPI**
72	Adams	666-7777
84	Rice	666-5555
49	Smith	111-6666
12	Young	777-7777
62	Black	444-5555
77	Jones	777-6666
51	Marsh	888-2222

Base table and secondary index subtable rows are partitioned as illustrated by the following graphic:



FF07D315

Sample Queries

The following queries can all be answered without having to do a full-table scan. Note that *uniqueness value* is abbreviated UV throughout.

Query 1

The first query uses the NUPI column Phone as the WHERE clause attribute, with the requested value being 555-7777.

```
SELECT *
FROM Customer
WHERE Phone = '555-7777';
```

Stage	Process
1	The hashing algorithm generates the rowID for this primary index access and finds that rows having the NUPI value 555-7777 hash to AMP 4.
2	The Dispatcher sends an AMP retrieval step directly to AMP 4, where the file system retrieves two matching rows. The first matching row has rowID=778 and UV=3, while the second matching row has rowID=778 and UV=7. The rowIDs are identical because this primary index is nonunique.
3	The file system reads the requested rows and returns them to the requestor. Cust=95, Name=Peters, Phone=555-7777 Cust=56, Name=Smith, Phone=555-7777

Query 2

The second query uses the USI column Cust as the WHERE clause attribute, with the requested value being 95.

```
SELECT *
FROM Customer
WHERE Cust = 95;
```

Stage	Process
1	The hashing algorithm generates the rowID for this USI access. The hash map indicates that an index row having the value 95 hashes to the Customer USI subtable on AMP 3 with a rowID=588.
2	The Dispatcher sends an AMP retrieval step directly to AMP 3, where the file system reads the subtable row having the rowID=588 to determine which AMP owns the base table row.
3	The file system retrieves the base table rowID 778 and uniqueness value 3 from the USI subtable row and determines that the requested base table row is stored on AMP 4.

Stage	Process
4	The retrieval directive is passed to AMP 4 where the row for customer number 95, having rowID value 778,3 is located.
5	The file system reads the requested row and returns it to the requestor. Cust=95, Name=Peters, Phone=555-7777

Query 3

The third query uses the NUSI column Name as the WHERE clause attribute, with the requested value being Smith.

```
SELECT *
FROM Customer
WHERE Name = 'Smith';
```

Stage	Process
1	The Dispatcher broadcasts an AMP retrieval step containing the NUSI value Smith to all AMPs.
2	The file system scans the NUSI subtable for Customer on each AMP, retrieving all the base table rowIDs and uniqueness values associated with the Name Smith. The steps are processed in parallel and one row is located on each AMP. • AMP1 (NUSI rowID=432, UV=8; Base table rowID=640, UV=1) • AMP2 (NUSI rowID=432, UV=3; Base table rowID=884, UV=1) • AMP3 (NUSI rowID=432, UV=1; Base table rowID=147, UV=1) • AMP4 (NUSI rowID=432, UV=5; Base table rowID=778, UV=7)
3	The file system directly retrieves the base table rows on each AMP in parallel and returns them to the requestor. Cust=40, Name=Smith, Phone=222-3333 Cust=74, Name=Smith, Phone=555-6666 Cust=49, Name=Smith, Phone=111-6666 Cust=56, Name=Smith, Phone=555-7777

Secondary Index Build and Access Operations Summarized

Teradata processes USIs and NUSIs very differently, as summarized in the table on the following page, which is based on the following simple SQL statements.

- Supporting USI build statement.

```
CREATE UNIQUE INDEX (customer_number)
ON customer_table;
```

- Supporting NUSI build statement.

```
CREATE INDEX NUSI_name (customer_name)
ON customer_table;
```

- Supporting USI access statement.

```
SELECT *
FROM customer_table
WHERE customer_number=12;
```

- Supporting NUSI access statement.

```
SELECT *
FROM customer_table
WHERE customer_name='SMITH';
```

Stage	Process			
	Build		Access	
	USI	NUSI	USI	NUSI
1	Each AMP accesses its subset of the base table rows.		The supplied index value hashes to the corresponding secondary index row and the base table rowID is located.	A message containing the secondary index value is broadcast to every AMP.
2	Each AMP copies the secondary index value and appends the rowID for the base table row.	Each AMP builds a spool file containing each secondary index value found followed by the rowID for the row it came from.	The retrieved base table rowID is used to access the specific data row.	For a hash-ordered NUSI, each AMP creates a local row hash and uses it to access its portion of the index subtable to see if a corresponding row exists. Value-ordered NUSI index subtable values are scanned only for the range of values specified by the query.

Stage	Process			
	Build		Access	
	USI	NUSI	USI	NUSI
3	Each AMP creates a Row Hash on the secondary index value and puts all three values onto the BYNET.	For hash-ordered NUSIs, each AMP sorts the rowIDs for each secondary index value into ascending order. For value-ordered NUSIs, the rows are sorted in NUSI value order.	The process is complete. This is typically a two-AMP operation.	If an index row is found, the AMP uses the base table rowID or value order list to access the corresponding base table rows.
4	The appropriate AMP receives the data and creates a row in the index subtable.	For hash-ordered NUSIs, each AMP creates a row hash value for each secondary index value on a local basis and creates a row in its portion of the index subtable. For value-ordered NUSIs, storage is based on NUSI value rather than the row hash value for the secondary index.		The process is complete. This is always an all-AMPs operation.
5	The process is complete.	The process is complete.		

Secondary Index Usage Summary

General Summary

All Teradata secondary indexes have the following properties:

- Can enhance the speed of data retrieval.

  Because of this, secondary indexes are most useful in decision support applications.
- Do not affect base table data distribution.
- Maximum of 32 secondary and join indexes defined per table.

  The limit of 32 indexes applies to any combination of secondary and join indexes defined on a table, ranging from 0 secondary indexes and 32 join indexes to 32 secondary indexes and 0 join indexes.
- Can be composed of as many as 64 concatenated columns.
- Can be created or dropped dynamically as data usage changes or if they are found not to be useful for optimizing data retrieval performance.
- Require additional disk space to store subtables.
- Require additional I/Os on INSERTs, DELETEs, and possibly on UPDATEs.

  Because of this, secondary indexes are not nearly as useful in OLTP applications as they are in DSS applications.
- Should not be defined on columns whose values change frequently.
- Should not define individual secondary indexes if typical queries using them access more than ten percent of the rows in the table.
- Should not include columns that do not enhance selectivity.
- Should not use composite secondary indexes when multiple single column indexes and bit mapping might be used instead.
- Composite secondary index is useful if it reduces the number of rows that must be accessed.
- The Optimizer does not use composite secondary indexes unless a WHERE clause condition specifies explicit values for each column in the index.
- Most efficient for selecting a small number of rows.
- Can be unique or nonunique.
- NUSIs can be hash-ordered or value-ordered.
- Ordering for NUSIs defined with an ORDER BY clause is restricted to a single numeric column of 4 of fewer bytes.

USI Summary

Unique secondary indexes have the following properties:

- Guarantee that each complete index value is unique.
- Any access is a two-AMP operation.

NUSI Summary

Nonunique secondary indexes have the following properties:

- Useful for locating rows having a specific (typically concatenated) value in the index.
- Can be hash-ordered or value-ordered.

  Value-ordered NUSIs are particularly useful for enhancing the performance of range queries.
- Any access is an all-AMPs operation.

Further Information

See the chapter "Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4* under the topics "CREATE TABLE" and "CREATE INDEX" for more information about secondary indexes.

Secondary indexes are also described in *Teradata RDBMS SQL Reference, Volume 1*.

Also consult *Teradata RDBMS Performance Optimization* for more detailed information on using secondary indexes to enhance the performance of your databases.

Chapter 11:

Join and Hash Indexes

This chapter describes the Teradata join and hash indexes. These indexes permit you to undertake a wide variety of physical denormalizations of the database without affecting the normalization of the logical and physical models.

The various types of hash and join indexes are described, including the following special join indexes.

- "Single-Table Join Indexes" on page 11-23
- "Aggregate Join Indexes" on page 11-26

Other topics include information on how to estimate the overhead of various join indexes, Optimizer criteria for selecting a join index, special storage options offered by join indexes, and numerous examples.

See Chapter 16: "Design Issues for Tactical Queries" for a description of the special design considerations that must be evaluated for using hash and join indexes to support tactical queries.

Join Indexes

Introduction

Join indexes are file structures designed to permit queries (join queries in the case of multitable join indexes) to be resolved by accessing the index instead of having to access and join their underlying base tables.

Multitable join indexes are useful for queries where the index structure contains all the columns referenced by one or more joins, thereby allowing the index to cover that part of the query, making it possible to retrieve the requested data from the index rather than accessing its underlying base tables. For obvious reasons, an index with this property is often referred to as a covering index. Even if a join index does not completely cover a query, the Optimizer can often use it to join to its underlying base tables in a way that provides better query optimization than scanning the base tables for all the columns specified in the request (see "Partial Query Coverage" on page 11-5).

From the point of view of a database designer, the multitable join index permits great adaptability because it provides the flexibility of normalization while at the same time offering the opportunity to create alternative, denormalized virtual data models, providing what might be called materialized views of the database.

Depending on how the index is defined, single-table join indexes can also be useful for queries where the index structure contains only *some* of the columns referenced in the statement. This situation is referred to as a partial cover of the query.

Join indexes are also useful for queries that aggregate columns from tables with large cardinalities. These indexes play the role of prejoin and summary tables without denormalizing the logical design of the database and without incurring the update anomalies and performance problems presented by denormalized tables. While it is true that denormalization often enhances the performance of a particular query or family of queries, it can and often does make other queries perform more poorly.

Unlike traditional indexes, join indexes do not store pointers to their associated base table rows. Instead, they are generally used as a fast path *final* access point that eliminates the need to access and join the base tables they represent. They substitute *for* rather than point *to* base table rows. The only exception to this is the case where an index partially covers a query. If the index is defined using either the ROWID keyword or the UPI of its base table as one of its columns, then it can be used to join with the base table to cover the query.[1]

In this case, you might create a join index to contain only the join column and the ROWID or the UPI column set of the table. The process is as follows:

1. The join index is joined with the other table and any selection conditions that can be evaluated during the join to eliminate disqualified rows are applied.
2. The result of the join is stored in a temporary table and redistributed based on the ROWID or UPI of the base table to perform a join with the base table.

This join can happen at different points in the query plan, depending on estimated costs. For example, the Optimizer might determine that it is cheaper to perform a join with another table involved in the query before joining to the base table.

Summary of Join Index Functions

A join index always has at least one of the following functions.

- Joins multiple tables (optionally with aggregation) in a prejoin table.
- Replicates all, or a vertical subset, of a single base table and partitions its rows using a different primary index than the base table, such as a foreign key column to facilitate joins of very large tables by hashing them to the same AMP.
- Aggregates one or more columns of a single table as a summary table.

The AMP software updates join indexes automatically, so the only administrative task a DBA must perform is to keep the statistics on multitable join index columns and their indexes current. For single-table join indexes, the best policy is to use base table statistics rather than to collect statistics directly on the columns of the index.

---

1 You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement. See "CREATE JOIN INDEX" in *Teradata RDBMS SQL Reference, Volume 4*.

Similarities of Join Indexes to Base Tables

In many respects, hash and join indexes are identical to base tables.

For example, you can do the following things with a join index:

- Create nonunique secondary indexes on its columns.
- Perform any of the following statements against it.
  - COLLECT STATISTICS (Optimizer Form)
  - DROP JOIN INDEX
  - DROP STATISTICS (Optimizer Form)
  - HELP JOIN INDEX
  - SHOW JOIN INDEX

Unlike base tables, you *cannot* do the following things with join indexes:

- Query or update join index rows.

  For example, if ordCustIdx is a join index, then the following query is not legal:

  ```
  SELECT o_status, o_date, o_comment
  FROM ordCustIdx;
  ```

- Store and maintain arbitrary query results.

  Note that you *can* maintain aggregates in a join index if you define it to do so.
- Create unique indexes on its columns.

Join Index Applications

Join indexes are useful for queries where the index table contains all the columns referenced by one or more joins, thereby allowing the Optimizer to cover all or part of the query by planning to access the index rather than its underlying base tables. An index that supplies all the columns requested by a query is said to cover that query and, for obvious reasons, is often referred to as a covering index. The Optimizer can also use join indexes that only cover a query partially if the index is defined properly (see "Partial Query Coverage" on page 11-5). Note that query covering is not restricted to join indexes: other indexes can also cover queries either in whole or in part.

Join indexes are also useful for queries that aggregate columns from tables with large cardinalities. For these applications, join indexes play the role of prejoin and summary tables without denormalizing the logical design of the database and without incurring the update anomalies and ad hoc query performance issues frequently presented by denormalized tables.

You can create join indexes that limit the number of rows in the index to only those that are accessed when a frequently run query references a small, well-known subset of the rows of a large base table. You create this type of join index by specifying a constant expression as the RHS of the WHERE clause, which narrowly filters the rows included in the join index. This is known as a sparse join index.

Partial Query Coverage

Partial query coverage allows join indexes whose columns do not match an entire query to be used to cover a subset of it. For example, one or two tables specified by the query might be covered by the join index, but the entire request is not. In some situations, there might be a number of commonly performed queries that join several tables where each of the queries joins two tables, say t1 and t2, on the same columns. For this situation, you can create a join index to join t1 and t2, and the Optimizer can use that join index for any queries that need to perform that join.

Partial query coverage also allows join indexes that contain only a subset of the columns of a base table referenced in the query to cover the query if that join index can be joined to the base table to retrieve additional referenced columns (this form of partial coverage is also used to implement hash indexes: see "Hash Indexes" on page 11-31).

For example, suppose there is a large table that needs to be joined frequently with another table on a column that is not the distributing column of the table. You can define a join index that redistributes the base table by the join column. However, because of the large number of rows and columns that need to be projected into the join index, the extra disk storage required does not allow the creation of such a join index.

Collecting Statistics on a Join Index

Issues concerning collecting statistics on the indexes of a join index are documented in "CREATE JOIN INDEX" in the chapter "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*.

Besides using the COLLECT STATISTICS statement (see "COLLECT STATISTICS (Optimizer Form)" in *Teradata RDBMS SQL Reference, Volume 4*) to collect statistics on join index indexes, you can also use the Teradata Statistics Wizard (see *Teradata Statistics Wizard User Guide*).

Fallback With Join Indexes

You can define fallback for join indexes. The criteria for deciding whether to define a join index with fallback are similar to those used for deciding whether to define fallback on base tables.

If you do not define fallback for a join index and an AMP is down, then the following additional criteria become critical.

- The join index cannot be used by the Optimizer to solve any queries that it covers.
- The base tables on which the join index is defined cannot be updated.

Further Information

Consult the following manuals for more detailed information on creating and using join indexes to enhance the performance of your database applications.

- *Teradata RDBMS Performance Optimization*
- *Teradata RDBMS SQL Reference, Volume 4*

Types of Join Index

Introduction

You can define several types of join index, each having its unique role in enhancing the performance of your database queries without denormalizing the base tables that support your ongoing ad hoc data warehouse activities. Because join index tables cannot be accessed by users and are not part of the logical design for a database, they can be used to create persistent prejoin and summary tables without removing or otherwise lessening the ability of your databases to support a wide range of ad hoc queries.

Sparse Join Indexes

Any join index, whether simple or aggregate, multitable or single-table, can be sparse. A sparse join index uses a constant expression in the WHERE clause of its definition to narrowly filter its row population. For example, the following DDL creates an aggregate join index containing only the sales records from 2000:

```
CREATE JOIN INDEX j1 AS
SELECT storeid, deptid, SUM(sales_dollars)
FROM sales
WHERE EXTRACT(year, sales_date) = 2000
GROUP BY storeid, deptid;
```

When a user query is entered, the Optimizer determines if accessing j1 gives the correct answer and is more efficient than accessing the base tables. This sparse join index would be selected by the Optimizer only for queries that restricted themselves to data from the year 2000. For example, a query might require data from June of 2000. Because the join index j1 contains all of the data for year 2000, it might be used to satisfy the following query:

```
SELECT storeid, deptid, SUM(sales_dollars)
FROM sales
WHERE sales_date >= 6/1/2000 AND sales_date < 7/1/2000
GROUP BY storeid, deptid;
```

As another example, the following DDL creates a join index containing only those customers living in four western states of the US:

```
CREATE JOIN INDEX westcust AS
SELECT cust.id, cust.address, donations.amount
FROM cust, donations
WHERE cust.custid = donations.custid
AND donations.ddate > '1999/01/01'
AND cust.state IN ('CA', 'OR', 'WA','NV');
```

The Optimizer can use this join index for the following query written to find all donors from California with donations of 1 000 USD or more made since January 1, 1999.

```
SELECT custid, cust.address
FROM cust, donations
WHERE donations.amount > 1000
AND cust.state = 'CA'
AND donations.ddate > '1999/01/01';
```

Because the customers and donations considered by the query are part of the subset included in the join index, the Optimizer uses it to answer the query. This can save a great deal of time especially in situations where the base tables are very large, but queries typically look only at subsets of the tables.

Simple Join Indexes

The primary function of a simple join index is to provide the Optimizer with a performant, cost-effective means for satisfying any query that specifies a frequently performed join operation. The simple join index permits you to define a permanent prejoin table without violating the normalization of the database schema.

For example, suppose that a common task is to look up customer orders by customer number and date. You might create a join index like the following, linking the customer table, the order table and the order detail table:

```
CREATE JOIN INDEX cust_ord2
AS SELECT cust.customerid,cust.loc,ord.ordid,item,qty,odate
FROM cust, ord, orditm
WHERE cust.customerid = ord.customerid
AND ord.ordid = orditm.ordid
```

While you might never issue a query that completely joined these three tables, the key benefit of this join index is its versatility. For example, a query that only looks at the customers for a single state, like the following, can still use the cust_ord2 join index rather than accessing its underlying base tables.

```
SELECT cust.customerid, ord.ordid, item, qty
FROM cust, ord, orditm
WHERE cust.customerid = ord.customerid
AND ord.ordid = orditm.ordid
AND cust.loc = 'WI'
```

It is also possible for a join index to be used to partially cover a query to improve query performance. For example, if you wanted to count the number of orders made by customers in the European region during October, you might use the following query:

```
SELECT cust.customerid,COUNT(ord.ordid)
FROM cust, ord, orditm, location
WHERE ord.ordid = orditm.ordid
AND cust.customerid = ord.customerid
AND cust.loc = location.loc
AND location.region = 'EUROPE'
AND EXTRACT(MONTH, ord.orddate) = 10
GROUP BY cust.customerid;
```

In this example, the query includes the LOCATION table which is not included in the join index. Teradata can still use the join index to partially cover the query by joining the contents of the join index with the LOCATION table.

Join indexes can, and should, be defined using outer joins. These outer join indexes can be used to cover both inner and outer join queries. To understand how this is possible, consider that an outer join can be thought of as producing a result consisting of two sets of rows. The first set corresponds to the set of matched rows obtained when a row from the outer table matches one or more rows from the inner table (this set corresponds to the set of rows defined by the inner join with the same join condition). The second corresponds to the set of unmatched rows: those rows from the outer table that do not match any rows from the inner table.

Except for the presence of the unmatched row set, an outer join is the same as an inner join, and produces the same result. Therefore if an inner join query can be completely satisfied by the matched set of rows from an outer join index, the Optimizer uses it.

For example consider the following join index defined on the three tables t1, t2, and t3. Tables t1 and t2 are joined with an inner join, and the result is joined with table t3 using an outer join. The outer table is the result of joining tables t1 and t2.

```
CREATE JOIN INDEX ji1 AS
SELECT a1, a2, a3
FROM (t1 JOIN t2 ON a1 = a2)
LEFT OUTER JOIN t3 ON a1 = a3;
```

Column a3 is the unique primary index for table t3. Column a3 might be specified in the table definition explicitly as a primary index, or simply as unique. This means that all of the rows from the join of t1 with t2 are in the join index exactly once, either in the matched set, or in the unmatched set. Therefore, the following query can be satisfied by the join index:

```
SELECT a1, a2
FROM t1, t2
WHERE a1 = a2;
```

The Teradata coverage algorithm determines that there is partial coverage, and the Optimizer uses the join index to join with the base tables to project the non-covered columns if the cost is lower than performing the query using the base tables alone.

Single-Table Join Indexes

You can also define a simple join index on a single table. This permits you to hash some or all of the columns of a large replicated base table on a foreign key that hashes rows to the same AMP as another large table. In some situations, this is more performant than building a multitable join index on the same columns. In effect, you are redistributing an entire base table or a frequently accessed subset of base table columns using a join index when you do this. The main advantage comes from less under-the-covers update maintenance on the single table form of the index.

For example, suppose you have a primary index defined on a major entity column that joins with many foreign key subentity columns. The cost of the maintenance required to update a multitable join index defined on this table is many times greater than the cost of maintaining the underlying base table.

For a single-table join index, on the other hand, the maintenance cost is roughly double the cost of maintaining the base table. Some of these savings are later lost to the cost of having to join single-table index rows, but the overall savings remains substantially greater in many cases.

When you design a schema, there is often a small number of tables that is queried in such a way that for some frequently run workloads, the table is joined on one column, but for another important query, the table is joined on another column. The usual design solution is to distribute the rows of this table on the column that is most frequently used in a join. If there is more than one column, then a join index might be a good design choice. A join index can be used to redistribute the table on the secondary join attribute so that joins can be done without a redistribution step.

Join indexes can also be used to evaluate parameterized queries. For the Optimizer to use a join index in this situation, the query must also contain a non-parameterized condition in its WHERE clause that the join index covers.

For example, suppose you create the following base table and single-table join index:

```
CREATE TABLE tp1 (
pid     INTEGER,
name    VARCHAR(32),
address VARCHAR(32),
zipcode INTEGER
);

CREATE JOIN INDEX tp1_ji AS
SELECT pid, name, zipcode
FROM tp1
WHERE zipcode >50000
AND zipcode < 55000;
```

The following parameterized query can use this join index because the Optimizer knows that the matching rows are contained in the index because the WHERE clause predicate in the query is a conjunction between the covered term zipcode and the parameterized term :N.

```
USING (N VARCHAR(32))
SELECT pid, name
FROM tp1
WHERE zipcode IN (54455, 53066)
AND name = :N;
```

The explanation for this query looks like the following report:

```
Explanation
------------------------------------------------------------------------
  1) First, we lock a distinct CURT."pseudo table" for read on a
     RowHash to prevent global deadlock for CURT.TP1_JI.
  2) Next, we lock CURT.TP1_JI for read.
  3) We do an all-AMPs RETRIEVE step from CURT.TP1_JI by way of an
     all-rows scan with a condition of ("(CURT.TP1_JI.NAME = :N) AND
     ((CURT.TP1_JI.ZIPCODE = 54455) OR (CURT.TP1_JI.ZIPCODE = 53066 ))")
     into Spool 1 (group_amps), which is built locally on the AMPs.
     The size of Spool 1 is estimated with no confidence to be 1 row.
     The estimated time for this step is 0.15 seconds.
  4) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> The contents of Spool 1 are sent back to the user as the result of
     statement 1.  The total estimated time is 0.15 seconds.
```

Only prototyping can determine which is the better design for a given set of tables, applications, and hardware configuration.

Aggregate Join Indexes

Define an aggregate join index as a join index that specifies SUM or COUNT aggregate operations. No other aggregate functions are permitted in the definition of a join index. The primary function of an aggregate join index is to provide the Optimizer with a performant, cost-effective means for satisfying any query that specifies a frequently made aggregation operation on one or more columns.

In other words, aggregate join indexes permit you to define a summary table without violating the normalization of the database schema. This allows a join index to pre-compute an aggregate value that would otherwise potentially require a table scan and sort operation.

Aggregate join indexes can be especially helpful for queries that roll up values for dimensions other that the primary key dimension, which would otherwise require redistribution.

An aggregate join index can be used to cover aggregate queries that only consider a subset of groups contained in the join index or have more join tables than the join index. In order to allow the aggregate join index to be used in this way, its definition must satisfy the following conditions:

- The grouping clause must include all columns that are specified in the grouping clause of the query.
- All columns in the query WHERE clause that join to tables not in the aggregate join index must be part of the join index definition.

Join Index Type Summary

The following table lists and briefly describes the various join index types and their common uses.

Join Index Type	Description
Sparse	Any join index that limits its rows to those satisfying a constant condition in the WHERE clause of its definition. A sparse join index permits the database designer to limit the rows in the index to a tightly filtered subset of the rows in the component base table set. A common use of this feature might be to restrict the population of a join index to only those rows that are frequently accessed by a commonly performed query or set of queries.
Simple	A join index table defined without aggregation.
• Single Table	A columnar subset of a very large base table defined with a NUPI defined on a join key that causes its rows to be hashed to the same AMP as another very large table with a primary index defined on that same join key and to which it is frequently joined. Useful for resolving joins on large tables without having to redistribute the joined rows across the AMPs. Subentities are typically small tables, while major entity tables are typically quite large. See "Single-Table Join Indexes" on page 11-23 for additional information.
• Multitable	A columnar subset of two or more major tables that are frequently joined defined as a prejoin of those tables. The defined prejoin, or join index, permits the Optimizer to select it to cover frequently made join queries rather than specifying that its underlying base tables be searched and joined on the fly.

<table>
<tr><th>Join Index Type</th><th>Description</th></tr>
<tr><td>Aggregate

• Single Table</td><td>A join index table defined with aggregation on one or more of its columns.

A columnar subset of a base table defined with additional columns that are aggregate summaries of base table columns.

<table>
<tr><th>IF the summary table is to be used …</th><th>THEN its NUPI …</th></tr>
<tr><td>simply to maintain aggregates for an overlying base table without denormalizing the database</td><td>need not be defined on a join key column set.</td></tr>
<tr><td>to maintain aggregates for an overlying base table and to be joined frequently with another very large table</td><td>should be defined on a join key column set that hashes its rows to the same AMPs as the base table primary index.</td></tr>
</table>
</td></tr>
<tr><td>• Multitable</td><td>A columnar subset, including aggregates defined for one or more columns, of two or more major tables that are frequently joined defined as a prejoin of those tables.

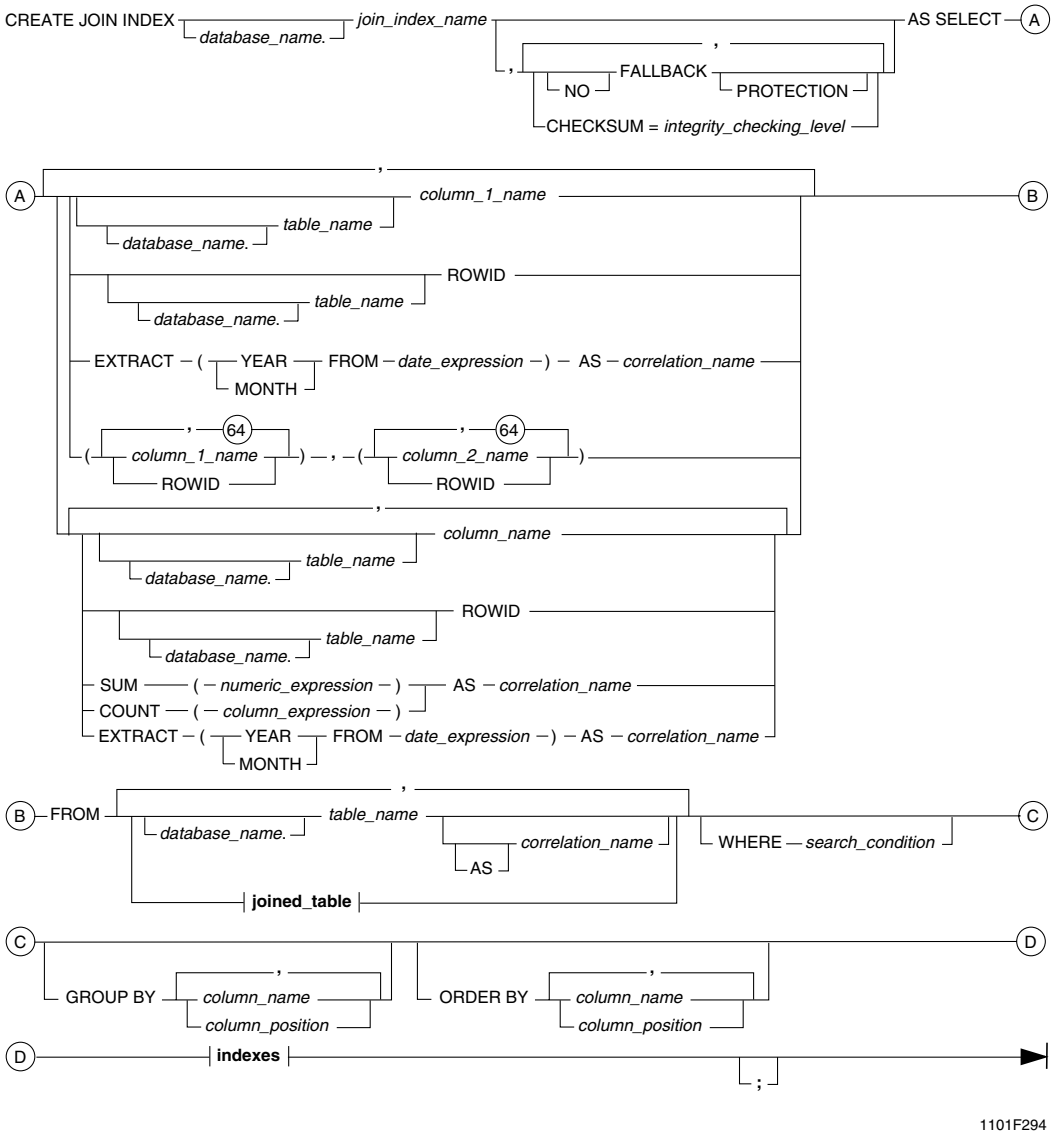
The defined prejoin, or join index, permits the Optimizer to select it to cover frequently made join queries that also compute aggregates rather than specifying that its underlying base tables be searched, aggregated, and joined on the fly.</td></tr>
</table>

Join Index Syntax Terminology

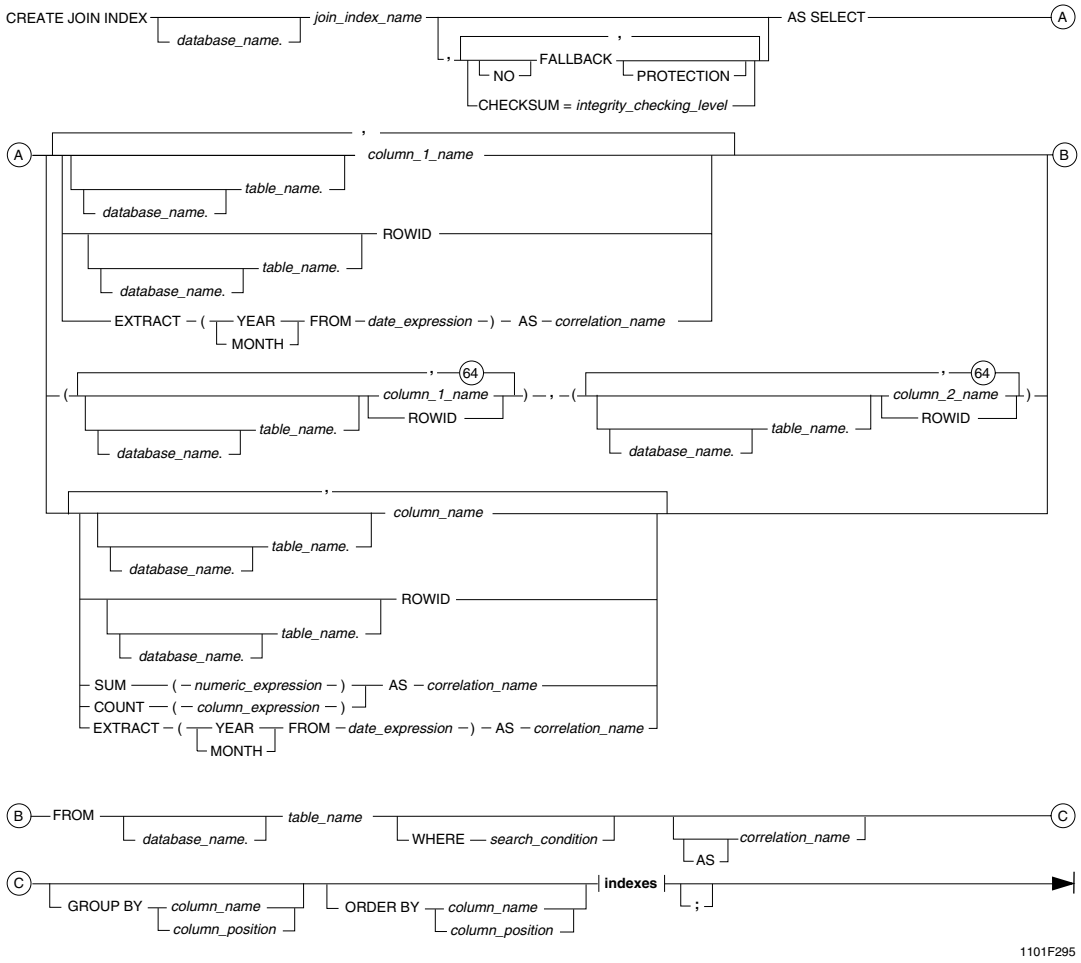
Introduction

Because some of the terminology used in the rest of this chapter relies heavily on the syntax of the CREATE JOIN INDEX statement, the syntax diagram for multitable and single-table join indexes and some definitions of syntax variables are repeated here.

Multitable Syntax

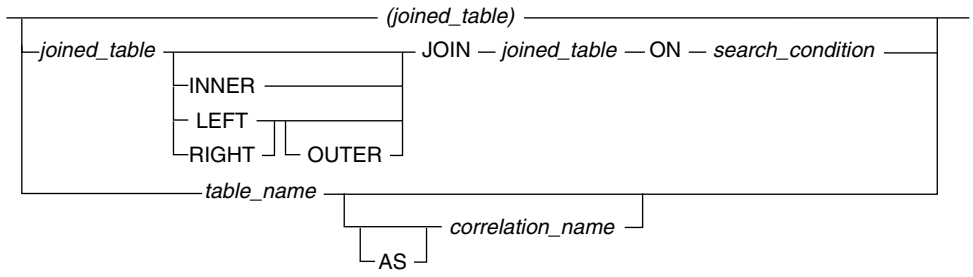


Single-Table Syntax



Joined_Table Excerpt

joined_table



where:

Syntax Element ...	Specifies ...
column_1_name	the name of a column to be contained in the join index. If a *column_2_name* is also defined, then each distinct value of *column_1_name* is stored only once.
column_2_name	a list of multiple columns to be contained in the join index. Represents the repeated (compressed) portion of the join index row.
joined_table	the name of a table (or already joined tables) from which the columns of this join index are to be derived. Not used for single-table join index syntax.
column_name	the name of the column within the join index definition used to group index rows on the AMPs.
column_position	the position of the column within the join index definition used to group index rows on the AMPs.

Note that at least one *column_name* value in a single-table join index definition must be the keyword ROWID or the UPI for the base table to enable the Optimizer to use the index when it partially covers an SQL query.[2]

2 You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement. See "CREATE JOIN INDEX" in *Teradata RDBMS SQL Reference, Volume 4*.

Join Index Definition Restrictions

Introduction

This topic lists several restrictions on join index definitions. For complete details about join index syntax, see "CREATE JOIN INDEX" in the chapter "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*.

Restrictions on Number of Join Indexes Defined Per Base Table

The maximum number of secondary, hash, and join indexes that can be defined for a table, in any combination, is 32.

For example, suppose you have four tables, each with multiple secondary, hash, and join indexes defined on them.

- Table_1 has 32 secondary indexes and no hash or join indexes.
- Table_2 has 16 secondary indexes, no hash indexes, and 16 join indexes.
- Table_3 has 10 secondary indexes, 10 hash indexes, and 12 join indexes.
- Table_4 has no secondary or hash indexes, but has 32 join indexes.

Each of these combinations is valid, but the first and last operate at the boundaries of the defined limits.

Restrictions on Number of Columns Per Referenced Base Table

The following restrictions apply to the number of columns per table that can be specified in a join index definition.

- The maximum number of columns that can be specified in a join index per referenced base table is 64.
- The maximum number of columns that can be specified in a multitable join index definition is 2 048.

Restrictions on Outer Join Definitions

The following restrictions apply to outer joins when used to define a join index.

- FULL OUTER JOIN is not valid.
- When you specify a LEFT or RIGHT outer join, the following rules apply:
  - No subsequent WHERE clause is permitted.
  - The outer table joining column for each condition must be contained in either *column_1_name* or *column_2_name*.
  - The inner table of each join condition must have at least one non-nullable column in either *column_1_name* or *column_2_name*.

    See "Join Index Syntax Terminology" on page 11-14 for definitions of *column_1_name* and *column_2_name*.

Restrictions on Primary and Secondary Index Definitions

You cannot explicitly define join index primary and secondary indexes as UNIQUE. In other words, you cannot use the keyword UNIQUE to define an index on a join index. This syntactical limitation pertains to index *definitions* for join indexes only: it does not restrict you from defining indexes on unique *column sets*, which is not only permitted, but encouraged.

You cannot specify a partitioned primary index for a join index.

Restrictions on Partial Covering by Join Indexes

The Optimizer can use a join index that partially covers a query in the following cases.

- One of the columns in the index definition is the keyword ROWID
- The column set defining the UPI of the underlying base table is also carried in the definition.
- The column set defining the NUPI of the underlying base table *plus* either of the following is also carried in the definition.
  - One of the columns in the definition of that index is the keyword ROWID.[3]
  - The column set defining a USI on the underlying base table.

  The ROWID option is the preferable choice.
- Partial covering is not supported for the inner table of an outer join.

When the operation is cost-effective, the Optimizer can then specify that the partially covering join index be joined to its underlying base table to pick up the column data not defined for the index itself.

For example, suppose you define the tables t1, t2, and t3 and the join indexes j1 and j2 as follows:

```
CREATE TABLE t1
(a1 INTEGER,
 b1 INTEGER,
 c1 INTEGER);

CREATE TABLE t2
(a2 INTEGER,
 b2 INTEGER,
 c2 INTEGER);
```

3 You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement. See "CREATE JOIN INDEX" in *Teradata RDBMS SQL Reference, Volume 4*.

```
CREATE TABLE t3
(a3 INTEGER,
b3 INTEGER,
c3 INTEGER);

CREATE JOIN INDEX j1 AS
 SELECT b1, b2, t1.ROWID t1rowid
 FROM t1,t2
 WHERE a1=a2;

CREATE JOIN INDEX j2 AS
 SELECT b1, b2, t1.ROWID t1rowid, t2.rowid t2rowid
 FROM t1,t2
 WHERE a1=b2;
```

Join index j1 partially covers the following queries:

```
SELECT a1, b1, c1, b2
FROM t1,t2
WHERE t1.a1=t2.b2
AND t1.b1=10;

SELECT a1, b1, c1, b2, b3, c3
FROM t1,t2,t3
WHERE t1.a1=t2.b2
AND t1.b1=t3.a3
AND t3.b3 > 0;
```

The same join index does *not* partially cover the following queries:

```
SELECT *
FROM t1,t2
WHERE t1.a1=t2.b2
AND t1.b1=10;

SELECT *
FROM t1,t2,t3
WHERE t1.a1=t2.b2
AND t1.b1=t3.a3
AND t3.b3 > 0;
```

Join index j2, on the other hand, partially covers all of these queries.

Even though you do not explicitly specify the join back to the base table when you write your query, it counts against the 64 tables per query block restriction on joins.

Restriction on Number of Join Indexes Selected Per Table

The Optimizer can use several join indexes for a single query, selecting one or more multitable join indexes as well as additional single-table join indexes for its join plan. The join indexes selected depend on the structure of the query, and the Optimizer might not choose all applicable join indexes for the plan. Always examine your EXPLAIN reports to determine which join indexes *are* used for the join plans generated for your queries. If a join index you think should have been used by a query was not included in the join plan, try restructuring the query and then EXPLAIN it once again.

The join planning process selects a multitable join index to replace any individual table in a query when the substitution further optimizes the query plan. For each such table replaced in the join plan by a multitable join index, as many as two additional single-table join indexes can also be added if their inclusion reduces the size of the relation to be processed, provides a better distribution, or offers additional covering.

The limit on the number of join indexes substituted per individual table in a query is enforced to limit the number of possible combinations and permutations of table joins in the Optimizer search space during its join planning phase. The rule helps to ensure that the optimization is worth the effort: in other words, that the time spent generating the query plan does not exceed the accrued performance enhancement.

Restrictions on Aggregate Functions

The following restrictions apply to aggregate functions used to define aggregate join indexes.

- Only the COUNT and SUM functions, in any combination, are valid.
- COUNT DISTINCT and SUM DISTINCT are not valid.
- Always type the COUNT and SUM fields in a join index definition as FLOAT to avoid overflow.

IF you ...	THEN Teradata ...
do not define an explicit data type for a COUNT or SUM field	assigns the FLOAT type to it.
define a COUNT or SUM field as anything other than FLOAT	returns an error and does not create the aggregate join index.

Restrictions on WHERE Clause Predicates

The following restrictions apply to WHERE clause join conditions specified in a join index definition.

- Data types for any columns used in a join condition must be drawn from the same domain because neither explicit nor implicit data type conversions are permitted.
- Multiple join conditions must be connected using the AND logical operator.

  The OR operator is *not* a valid way to connect multiple join conditions in a join index definition.
- You cannot specify independent inequality WHERE clause join conditions in a join index definition.

  The following rules apply to the use of inequality join conditions in a join index definition WHERE clause.
  - Inequality join conditions are supported only if they are ANDed to at least one equality join condition.
  - Inequality join conditions can be specified only for columns having the same data type in order to enforce domain integrity.
  - The only valid comparison operators for an inequality join condition are the following:
    - <
    - <=
    - >
    - >=

  The following join index definition is valid because the WHERE clause inequality join condition on o_totalprice and c_acctbal it specifies is ANDed with the previous equality join condition on o_custkey and c_custkey.

```
CREATE JOIN INDEX ord_cust_idx AS
SELECT c_name, o_orderkey, o_orderdate
FROM orders, customer
WHERE o_custkey = c_custkey
AND o_totalprice > c_acctbal;
```

  The following join index definition is *not* valid because the WHERE clause inequality join condition has no logical AND relationship with an equality join condition.

```
CREATE JOIN INDEX ord_cust_idx AS
SELECT c_name, o_orderkey, o_orderdate
FROM orders, customer
WHERE o_totalprice > c_acctbal;
```

Restrictions on ORDER BY Clauses

The following restrictions apply to ORDER BY clauses specified in a join index definition.

- Sort order is restricted to ASC. DESC is not valid.
- Aggregate columns and expressions are *not* permitted as a *column_name* or *column_position*.

  See "Join Index Syntax Terminology" on page 11-14 for definitions of *column_name* and *column_position*.
- Supported data types for *column_name* are the following:
  - DATE
  - INTEGER
  - SMALLINT
  - BYTEINT
  - DECIMAL

    The DECIMAL type is valid only for columns of four bytes or less.

Single-Table Join Indexes

Introduction

A single-table join index is a database object created using the CREATE JOIN INDEX statement, but specifying only one table in its FROM clause.

Because it does not relate to self-joins, the expression “single-table join index” is a contradiction in terms. The function performed by a single-table join index was not conceived until after the expression “join index” had already been selected and was in common use within Teradata. This application came about because an observant performance analyst had the insight that it was possible to use the join index mechanism with a single table to accomplish a different purpose from the original intent: to partition all or a subset of a very large base table as a “join index” on a different primary index than that used by the original base table in order to hash its rows to the same AMPs as another very large base table that with which it was frequently joined.

By repartitioning the base table, the logical design remains intact while at the same time its physical manifestation is, in a sense, radically denormalized, resulting in a marked performance enhancement for commonly made large table joins. By not denormalizing the base tables, the generality of the original logical design is retained with the result that the neither the ease nor the performance of making *any* ad hoc query against the database is compromised.

Functions of Single-Table Join Indexes

When you have an application for which join queries against a base table would benefit from replicating some or all of its columns in a different table hashed on the join key (usually the primary index of the table to which it is to be joined) rather than the primary index of the original base table, then you should consider creating one or more single-table join indexes on that table.

For example, you might want to create a single-table join index to avoid redistributing a large base table or to avoid the sometimes prohibitive storage requirements of a multitable join index. A single-table join index might be useful for commonly made joins having low predicate selectivity but high join selectivity, for example.

This strategy substitutes the join index for the underlying base table and defines a primary index that ensures that rows containing only the columns to be joined are hashed to the same AMPs, eliminating the need to redistribute rows when the database manager joins the tables.

Even though each single-table join index you create partly or entirely replicates its base table, you cannot query or update them directly just as you cannot directly query or update any other join index.

Related Strategies

Other functionally similar strategies for solving this problem can also be used. In general, only prototyping can determine which among the possible choices is best for a particular application environment and hardware configuration.

- You can create a hash index.

  For many applications, a hash index is a more versatile choice than a single-table join index.

  See "Hash Indexes" on page 11-31 for more information.
- The design technique of assigning a NUPI to a subentity table that hashes related rows to the same AMPs as a related major entity is superficially similar to a single-table join index. The differences are as follows.
  - Cardinalities

    The cardinalities of tables for which a single-table join index is defined are typically very similar to the base tables they are designed to be joined with, while those for major entity-subentity joins are typically very different, with the major entity typically having many more rows than the subentity.

    The entity PI-subentity NUPI strategy is typically used when the subentity is a relatively small table in terms of its degree as well as its cardinality.

    The single-table join index strategy is typically used when only a small subset of the columns from the base table from which the single-table join index is derived are frequently joined with the base table in question.
  - Specialization

    When you create a single-table join index, the parent base table from which it is derived has a different primary index and its rows hash to different AMPs. The single-table join index is a denormalized, specialized database object defined for a specific purpose, while the parent base table is a normalized, more general database object. Both tables in an entity-subentity relationship remain normalized and generalized database objects.
- You can create a standard multitable join index that prejoins the entity attributes most likely to be joined in a query.

  The potential downside of this strategy is that tables with high cardinality (tables of the size for which the single-table join index solution is ideal) present a higher maintenance profile because the rows in the join index must be updated each time *either* of the base tables is updated. The single-table join index solution effectively reduces this factor by half.

  The upside of a standard multitable join index strategy is that, at least for the queries for which they are designed, the AMP software does not have to perform any join processing because the required rows are already prejoined. The single-table join index avoids a costly redistribution of table rows, but join processing is still required to respond to the query.

- You can create a denormalized prejoin base table.

  Denormalization reduces the generality of the database for ad hoc queries and data mining operations as well as introducing various problematic update anomalies. Nevertheless, a relatively mild degree of denormalization is standard in physically implemented databases, and for some applications might be the only performant solution.

See "Example 5: Single-Table Join Index" on page 11-61 for an example of using a single-table join index.

Partially Covering a Query With a Join Index

If one column reference of the definition for a join index is substituted with the keyword ROWID, or if its definition includes the columns that define either the UPI or USI of its underlying base table, then the Optimizer can use that index to partially cover a query.

If statistics indicate that it would be cost-effective, the Optimizer can specify that the partially covering single-table join index be joined to its underlying base table using either the ROWID or the UPI or USI to pick up the column data not defined for the index itself.

Even though you do not explicitly specify this join when you write your query, it counts against the 64 table restriction on joins.

Aggregate Join Indexes

Introduction

An aggregate join index is a database object created using the CREATE JOIN INDEX statement, but specifying one or more columns that are derived from an aggregate expression.

Only the SUM and COUNT aggregate operators are permitted in the definition of an aggregate join index; however, most of the other simple aggregate statistics can be derived from these using column expressions.

Functions of Aggregate Join Indexes

Besides the benefits offered by the simple join index, the aggregate form also permits you to create persistent summary tables without denormalizing your database. Not only can these summaries be stored in persistent tables, they can also exempt the AMP software from the labor of computing column aggregate expressions on the fly.

Related Strategies

Other functionally similar strategies for solving this problem can also be used. In general, only prototyping can determine which among the possible choices is best for a particular application environment and hardware configuration.

- You can create a global temporary table definition with aggregate expressions defined on some or all its columns. This is not so much an alternate strategy as it is an entirely *different* strategy designed for an entirely different application.

  Temporary tables are private to the session that materializes them and their data does not persist beyond the end of that session. Unless you specify the ON COMMIT PRESERVE option when the definition for the temporary table is created, its contents do not persist even across individual database *transactions*. Unless provisions are made to write their contents to a persistent base table when a session ends, their data is not saved. Without numerous safeguards built into the process, this is not a method that provides any assurances about the integrity of the data it produces because while the contents of any global temporary table are private to the session in which it is created, the definition of the table is global within a database and any number of different sessions and users could materialize a different version of the table, populate it, and write the results to the same base table as any other session.
- You can create a volatile table with aggregate expressions defined on some or all its columns. The drawbacks of global temporary tables for this application apply equally to volatile tables.

- You can create a denormalized base table with aggregate expressions defined on some or all its columns.

  Denormalization always reduces the generality of the database for any ad hoc queries or data mining operations you might want to undertake as well as introducing various problematic update anomalies. While a relatively mild degree of denormalization is standard in physically implemented databases, the sort of denormalization called for by this solution is probably beyond what most DBAs would find acceptable, the enthusiasm of dimensional modeling theorists notwithstanding.

  Nonetheless, for some applications this approach might be the only performant solution.

Optimizer Criteria for Deciding When to Use Aggregate Join Indexes in a Query Plan

The Optimizer analyzes all queries for the use of an aggregate join index. In determining if an aggregate join index can be used to process a query, the Optimizer applies the following rules.

- An aggregate join index cannot be used to substitute for the base tables of a nonaggregate query.
- Like a simple join index, an aggregate join index qualifies for inclusion in the plan if it specifies either the same joins or a subset of the joins of the query.
- If an aggregate join index specifies a subset of joins of the query, then all the joined fields of the remaining join conditions on the tables of the aggregate join index must be specified in the GROUP BY clause *and* in the select list of the aggregate join index definition.
- Even if an aggregate join index specifies the same joins as the query, it does not qualify if its GROUP BY columns are neither the same nor a superset of the GROUP BY columns for the query.
- An aggregate join index must contain all the columns needed by the query for the tables that it replaces. When trying to match expressions from a query to the select list for a join index, the Optimizer applies the following rules.
  - Addition and multiplication are commutative.
  - COUNT(*) and COUNT(*non-null_expression*) are used interchangeably.
  - A numeric expression from the join index that is converted to FLOAT can be used to match any expression of the same type defined with or without the conversion from the query.
  - An AVERAGE aggregate expression in a query is converted to a SUM/COUNT expression when an aggregate join index substitutes for it.
  - A SUM(0) expression in a query is converted to a constant having value 0.
  - A SUM(*constant*) expression in a query matches with *constant* * COUNT(*) in a aggregate join index definition.
  - The SUM or COUNT function from SUM(CASE *expression*) or COUNT(CASE *expression*) can be pushed to all the resulting expressions of the CASE expression in order to find a match. For example, the following expression can be converted to the expression following it in order to find a match.

```
SUM(CASE WHEN x1=1
         THEN 1
         ELSE 0)

CASE WHEN x1=1
THEN SUM(1)
ELSE SUM(0)
```

- Predicate conditions are converted according to the following conversion table.

This predicate condition …	Is converted to this equivalent expression …
expression >= 'YYYY0101'	'EXTRACT(YEAR FROM *expression*) >= YYYY'
expression > 'YYYY1231'	'EXTRACT(YEAR FROM *expression*) > YYYY'
expression < 'YYYY0101'	'EXTRACT(YEAR FROM *expression*) < YYYY'
expression <= 'YYYY1231'	'EXTRACT(YEAR FROM *expression*) <= YYYY'
expression >= 'YYYYMM01'	'EXTRACT(YEAR FROM *expression*) > YYYY OR (EXTRACT(YEAR FROM *expression*) = YYYY AND EXTRACT(MONTH FROM *expression*) >= MM)'
expression > 'YYYYMMLD' where LD is the last day of the month specified by MM	'EXTRACT(YEAR FROM *expression*) > YYYY OR (EXTRACT(YEAR FROM *expression*) = YYYY AND EXTRACT(MONTH FROM *expression*) > MM)'

This predicate condition ...	Is converted to this equivalent expression ...
expression <= 'YYYYMMLD' where LD is the last day of the month specified by MM	'EXTRACT(YEAR FROM *expression*) < YYYY OR (EXTRACT(YEAR FROM *expression*) =YYYY AND EXTRACT(MONTH FROM *expression*) <= MM)'
expression < 'YYYYMM01'	'EXTRACT(YEAR FROM *expression*) < YYYY OR (EXTRACT(YEAR FROM *expression*) = YYYY AND EXTRACT(MONTH FROM *expression*) < MM)'

Hash Indexes

Introduction

Hash indexes are file structures that share properties with both single-table join indexes (see "Comparison of Hash and Single-Table Join Indexes" on page 11-31) and secondary indexes (see "Comparison of Hash and Single-Table Join Indexes" on page 11-31).

From an architectural perspective, the incorporation of auxiliary structures as a transparently embedded element of the hash index column set is what most distinctly distinguishes hash indexes from single-table join indexes. These auxiliary structures are components of the base table, and are added to the hash index definition by default if they are not explicitly declared by the CREATE HASH INDEX column set definition.

If the columns you specify for the hash index column set duplicate the default auxiliary structure columns, then those columns are not added redundantly. The auxiliary structures provide indexed access to base table rows.

If you do not specify a partition key explicitly with the BY clause of the CREATE HASH INDEX statement, then Teradata adds this auxiliary pointer data to the hash index rows automatically and then uses it to partition them.

Comparison of Hash and Single-Table Join Indexes

The reasons for using hash indexes are similar to those for using single-table join indexes. Not only can hash indexes optionally be specified to be partitioned in such a way that their rows are AMP-local with their associated base table rows, they can also provide a transparent direct access path to those base table rows to complete a query only partially covered by the index. This facility makes hash indexes somewhat similar to secondary indexes in function. Hash indexes are also useful for covering queries so that the base table need not be accessed at all.

The most apparent external difference between hash and single-table join indexes is in the syntax of the SQL used to create them. The syntax for CREATE HASH INDEX is similar to that for CREATE INDEX. As a result, it is simpler to create a hash index than to create a single-table join index.

The following list summarizes the similarities of hash and single-table join indexes.

- Primary function of both is to improve query performance.
- Both are maintained automatically by the system when the relevant columns of their base table are updated by a DELETE, INSERT, or UPDATE statement.

- Both can be the object of any of the following SQL statements.
  - COLLECT STATISTICS (Optimizer Form)
  - DROP STATISTICS (Optimizer Form)
  - HELP INDEX
  - SHOW HASH INDEX
- Both receive their space allocation from permanent space and are stored in distinct tables.
- Both can be hash- and value-ordered.
- Both can be compressed.
- Both can be FALLBACK protected.
- Neither can be queried or directly updated.
- Neither can have a partitioned primary index.
- Neither can store an arbitrary query result.
- Both share the same restrictions for use with the MultiLoad, FastLoad, and Archive/Recovery utilities.

The following table summarizes the differences between hash and join indexes.

Hash Index	Join Index
Indexes one table only.	Can index multiple tables.
A logical row corresponds to one and only one row in its referenced base table.	A logical row can correspond to multiple rows in the referenced base tables.
Column list cannot contain aggregate or ordered analytical functions.	Column list can contain aggregate functions.
Cannot have a secondary index.	Can have a secondary index.
Supports transparently added, system-defined columns that point to the underlying base table rows.	Does not add underlying base table row pointers implicitly. Pointers to underlying base table rows can be created explicitly by defining one element of the column list using the keyword ROWID.[a]

a. You can only specify ROWID in the outermost SELECT of the CREATE JOIN INDEX statement. See "CREATE JOIN INDEX" in *Teradata RDBMS SQL Reference, Volume 4*.

Summary of Hash Index Functions

A hash index always has at least one of the following functions.

- Replicates all, or a vertical subset, of a single base table and partitions its rows with a user-specified partition key column set, such as a foreign key column to facilitate joins of very large tables by hashing them to the same AMP.
- Provides an access path to base table rows to complete partial covers.

The AMP software updates hash indexes automatically, so the only task a DBA must perform is to keep the statistics on hash index or base table columns current (see "Collecting Statistics on a Hash Index" on page 11-36).

Similarities of Hash Indexes to Base Tables

In many respects, a hash index is identical to a base table.

For example, you can do the following things with a hash index:

- Perform any of the following statements against it.
  - COLLECT STATISTICS (Optimizer Form)
  - DROP HASH INDEX
  - DROP STATISTICS (Optimizer Form)
  - HELP HASH INDEX
  - SHOW HASH INDEX

Unlike base tables, you *cannot* do the following things with hash indexes:

- Query or update hash index rows.

  For example, if ordCustHdx is a hash index, then the following query is not legal:

  ```
  SELECT o_status, o_date, o_comment
  FROM ordCustHdx;
  ```
- Store and maintain arbitrary query results.
- Create secondary indexes on its columns.

Similarities of Hash Indexes to Secondary Indexes

Hash indexes are file structures that can be used either to resolve queries by accessing the index instead of its underlying base table or to enhance access performance when they do not cover a query by providing a secondary access path to requested base table rows. They can either substitute *for* or point *to* base table rows.

Because of these properties, hash indexes are useful for queries where the index structure contains all the columns referenced by a query, thereby *covering* the query, and making it possible to retrieve the requested data from the index rather than accessing its underlying base tables. For obvious reasons, an index with this property is often referred to as a covering index.

Hash indexes put in double duty by also providing pointers to base table rows to facilitate their access in situations where the index does *not* cover the query. This application of hash indexes is similar to how the Optimizer uses secondary indexes when it creates its access plans.

Because the partitioning of hash index rows is under user control through the specification of an explicit partition key in the CREATE HASH INDEX statement, rows can be distributed in various ways, depending on the requirements of an application. Distribution of secondary index rows, on the other hand, is not under user control. Because of this, they are less adaptable to the specific requirements of an individual application than hash indexes.

Both secondary and hash indexes provide access paths to base table rows and can be selected by the Optimizer to cover SQL queries. Both also share a similar DDL syntax that is very different from the syntax used to create single-table join indexes.

The key difference between hash and secondary indexes is that the hash index partition key is user-selectable, which often makes it more useful for processing queries.

Hash Index Applications

Hash indexes are useful for queries where the index table contains the columns referenced by a query, thereby allowing the Optimizer to cover it by planning to access the index rather than its underlying base table. An index that supplies all the columns requested by a query is said to cover that query and, for obvious reasons, is referred to as a covering index. Note that query covering is not restricted to hash indexes: other indexes can also cover queries.

Hash indexes can also be defined on a table in place of secondary indexes. This usage makes more sense when you partition the hash index in such a way that it facilitates a wider range of query processing than a secondary index might. Because hash indexes carry more of an update burden than secondary indexes, you should not define them on a table when a secondary index would serve the same intended function.

Compression of Hash Index Rows

When the columns of the order key are not unique, the rows of the hash index are compressed. Rows having the same values for the order key are compressed into a single physical row having fixed and repeating parts. If the columns do not fit into a single physical row, they can spill over to additional physical rows as is necessary.

Hash index compression, unlike join index compression, is not specified in the CREATE statement. Instead, the Parser determines which columns are to be compressed and which are not.

The fixed part of the row is made up of the columns that define the order key. The repeating part is composed of the remaining columns. See "Hash and Join Index Storage" on page 11-74 for a more detailed description of this sort of row compression.

Fallback With Hash Indexes

You can define fallback for hash indexes. The criteria for deciding whether to define a hash index with fallback are similar to those used for deciding whether to define fallback on base tables.

If you do not define fallback for a hash index and an AMP is down, then the following additional factors become critical.

- The hash index cannot be used by the Optimizer to solve any queries that it covers.
- The base tables on which the hash index is defined cannot be updated.

Further Information

Consult the following manuals for more detailed information on creating and using hash indexes to enhance the performance of your database applications.

- *Teradata RDBMS Performance Optimization*
- *Teradata RDBMS SQL Reference, Volume 4, Data Definition Statements*

Collecting Statistics on a Hash Index

Introduction

When there are no statistics for a hash index, the Optimizer uses the statistics of the corresponding base table rows, just as it does for a single-table join index.

This topic describes guidelines for collecting statistics on hash index columns.

General Guidelines

You should collect statistics on appropriate columns of a hash index frequently just as you would for any base table or join index. For most applications, you should collect the statistics on base table columns rather than on hash index columns. See "Collecting Statistics on Base Table Columns Instead of Hash Index Columns" on page 11-37 for further information.

When you create a hash index with a BY clause, you can collect and drop statistics on those columns using an INDEX clause to specify the column set. For example, suppose a hash index has the following definition.

```
CREATE HASH INDEX OrdHIdx8 (o_custkey, o_orderdate) ON Orders
BY (o_custkey, o_orderdate)
ORDER BY (o_orderdate);
```

Then you can collect statistics on the partitioning columns as shown in the following example.

```
COLLECT STATISTICS ON OrdHIdx8 INDEX (o_custkey, o_orderdate);
```

Note that you can *only* use columns specified in a BY clause definition if you use the keyword INDEX in your COLLECT STATISTICS statement.

A poorer choice would be to collect statistics using the column clause syntax. To do this, you must perform two separate statements.

```
COLLECT STATISTICS ON OrdHIdx8 COLUMN (o_custkey);
COLLECT STATISTICS ON OrdHIdx8 COLUMN (o_orderdate);
```

It is always better to collect the statistics for a multicolumn index on the index itself rather than individually on its component columns because its selectivity is much better when you collect statistics on the index columns as a set.

For example, a query with a predicate like `WHERE x=1 AND y=2` is better optimized if statistics are collected on INDEX (x,y) than if they are collected individually on column x and column y.

The same restrictions hold for the DROP STATISTICS statement.

You cannot use the Teradata Statistics Wizard to collect statistics on its primary or secondary indexes.

Collecting Statistics on Base Table Columns Instead of Hash Index Columns

The Optimizer substitutes base table statistics for hash index statistics when no demographics have been collected for its hash indexes. Because of the way hash index columns are built, it is generally best not to collect statistics directly on the index columns and instead to collect them on the corresponding columns of the base table. This optimizes both system performance and disk storage by eliminating the need to collect the same data redundantly.

There are two reasons why you might need to collect statistics on hash index columns.

- If you decide not to collect the statistics for the relevant base table columns for some reason, then you should collect them directly on the corresponding hash index columns.
- If the partitioning column set consists of more than one column, and there is no primary or secondary index on the base table that includes those columns, then you *must* collect the statistics for that column set on the hash index itself.

Guidelines for Collecting Statistics On Hash Index Columns

The guidelines for selecting hash index columns on which to collect statistics are similar to those for base tables and join indexes. The primary factor to consider in all cases is whether the statistics provide better access plans. If they do not, consider dropping them. If the statistics you collect produce *worse* access plans, then you should always report the incident to NCR support personnel.

When you are considering collecting statistics for a hash index, it might help to think of the index as a special kind of base table that stores a derived result. For example, any access plan that uses a hash index must access it with a direct probe, a full table scan, or a range scan. With this in mind, consider the following things when deciding which columns to collect statistics for.

-

IF an execution plan might involve ...	THEN collect statistics on the ...
search condition keys	column set that constitutes the search condition predicate.
joining the hash index with another table	join columns to provide the Optimizer with the information it needs to best estimate the cardinalities of the join.

-

IF a hash index is defined …	THEN you should collect statistics on the …
with a BY clause or ORDER BY clause (or both)	partition and order keys specified by those clauses.
without a BY clause	primary index column set of the base table on which the index is defined.
without an ORDER BY clause and the order key column set from the base table is not included in the *column_name_1* list	order key of the base table on which the index is defined. This action provides the Optimizer with several essential baseline statistics, including the cardinality of the hash index.

- If a hash index column appears frequently in WHERE clause predicates, you should consider collecting statistics on it as well, particularly if that column is the sort key for a value-ordered hash index.

Hash Index Definition Restrictions

Introduction

This topic lists several restrictions on hash index definitions. For complete details about hash index syntax, see "CREATE HASH INDEX" in the chapter "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*.

Restrictions on Number of Hash Indexes Defined Per Base Table

The maximum number of secondary, hash, and join indexes that can be defined for a table is 32, in any combination.

For example, suppose you have four tables, each with multiple secondary, hash, and join indexes defined on them.

- Table_1 has 32 secondary indexes and no hash or join indexes.
- Table_2 has 16 secondary indexes, no hash indexes, and 16 join indexes.
- Table_3 has 10 secondary indexes, 10 hash indexes, and 12 join indexes.
- Table_4 has no secondary or hash indexes, but has 32 join indexes.

Each of these combinations is valid, but the first and last operate at the boundaries of the defined limits.

Restrictions on Number of Columns Per Referenced Base Table

The maximum number of columns that can be specified in a hash index is 64.

Restriction on Number of Hash Indexes Selected Per Base Table

The Optimizer can use several hash indexes for a single query, selecting one or more multitable join indexes as well as additional hash indexes for its join plan. The hash indexes selected depend on the structure of the query, and the Optimizer might not choose all applicable hash indexes for the plan. Always examine your EXPLAIN reports to determine which hash indexes *are* used for the query plans generated for your queries. If a hash index you think should have been used by a query was not included in the query plan, try restructuring the query and then EXPLAIN it once again.

The join planning process selects a multitable join index to replace any individual table in a query when the substitution further optimizes the query plan. For each such table replaced in the join plan by a multitable join index, as many as two additional hash indexes can also be added if their inclusion reduces the size of the relation to be processed, provides a better distribution, or offers additional covering.

The limit on the number of hash indexes substituted per individual table in a query is enforced to limit the number of possible combinations and permutations of table joins in the Optimizer search space during its join planning phase. The rule helps to ensure that the optimization is worth the effort: in other words, that the time spent generating the query plan does not exceed the accrued performance enhancement.

Hash and Join Index Interactions With Other Teradata Systems and Features

Introduction

This topic describes how hash and join indexes interact with the following Teradata systems and features.

Triggers

You cannot define triggers and hash or join indexes on the same table.

Whichever feature is defined first for a table blocks the other from being created and returns an error message to the requestor.

Permanent Journal Recovery

You can use ROLLBACK or ROLLFORWARD statements to recover base tables with hash or join indexes defined on them; however, the indexes are not rebuilt during the recovery process. Instead, any such hash or join index is marked as non-valid, and you must drop and recreate it before the Optimizer can use it again in a query plan.

When a hash or join index has been marked not valid, the SHOW HASH INDEX and SHOW JOIN INDEX statements display a special status message to inform you that the index has been so marked.

Load Utilities

You cannot use FastLoad or MultiLoad to load data into base tables that have hash or join indexes because those indexes are not maintained during the execution of these utilities. If you attempt to load data into base tables with hash or join indexes using either of these utilities, an error message returns and the load does not continue.

To load data into a hash- or join-indexed base table using FastLoad or MultiLoad, you must first drop all hash and join indexes defined on that table.

Load utilities like TPump, which performs standard SQL row inserts and updates, *are* supported for hash- and join-indexed tables.

TPump

You *can* use the TPump utility, which perform standard SQL row inserts and updates, to unload data into base tables that have a hash or join index defined on them because those indexes are properly maintained during its execution.

Archive and Restore

Archive and Restore cannot be performed on hash or join indexes, though archiving is supported for a base table or database that has an associated hash or join index defined.

During a restore of such a base table or database, the system does not automatically rebuild the index. Instead, it is marked as non-valid. You must drop the index and then recreate it before the Optimizer can again select it for a query plan.

Summary

The following table summarizes the Teradata features that do not support hash or join indexes.

Feature	Reason Not Supported	Recommended Workaround
Triggers	Triggers are handled by the Resolver, while join indexes are handled by the Optimizer.	Do not define triggers and hash or join indexes on the same base tables.
Permanent journal recovery	Recovery process does not rebuild join indexes.	Rebuild hash and join indexes after the permanent journal recovery completes.
MultiLoad	Utilities do not maintain join indexes.	1 Drop hash and join indexes before loading or restoring the base tables. 2 Recreate hash and join indexes after loading or restoring the base tables.
FastLoad		
Archive/Restore	Restore process marks join indexes as non-valid and does not rebuild them.	

Improving Join Index Performance

Introduction

Depending on the complexity of the joins or the cardinalities of base tables that must be aggregated, join indexes optimize the performance of certain types of workloads.

Selecting the Primary Index for Join Indexes

As with any other Teradata table, you must define a primary index for any join index you create. While the primary index for any join index must be defined as a NUPI, the column on which it is defined need not be nonunique; in fact, the usual preferences for defining a unique rather than a nonunique column set as the primary index for a base table also apply to join indexes.

Because join indexes are not part of the logical model for a database, and because they are de facto denormalized database objects, there might not be an overriding reason to define one column set over another as the primary index apart from how much more evenly one set might distribute join index table rows than another.

It is usually important to use the join key as the primary index for single-table join indexes, but otherwise there is no compelling theoretical reason to select one unique (or, if not unique, highly singular) column set over another as the primary index.

Join index primary indexes are defined analogously to base table primary indexes: with the CREATE JOIN INDEX statement.

Selecting Secondary Indexes for Join Indexes

Join indexes are often more performant if they have one or more secondary indexes defined on them. The Optimizer adds a join index (even a partially covering join index) to its query plan whenever it can, and just because you define it to have a particular most likely use and access path, there is no reason to believe that the same join index might not also be useful for other, unplanned, queries. The Optimizer will join a base table that is unrelated to a join index *with* that join index if the query plan can be made more cost effective by doing so.

As for a primary index, a secondary index on a join index cannot be defined as UNIQUE, even though the column set on which it is defined *is* unique. This rule is enforced because of the way indexes on join index tables are handled internally.

See “CREATE JOIN INDEX” in the chapter “SQL Data Definition Language Statement Syntax” in *Teradata RDBMS SQL Reference, Volume 4*, for further information about creating join indexes and using secondary indexes with them.

Also see *Teradata Statistics Wizard User Guide* and “Collecting Statistics on a Join Index” on page 11-5 for information about how to use the Teradata Statistics Wizard to determine which join index columns and indexes statistics should be collected on.

Statistics and Other Demographic Data for Join Indexes

Collect statistics on the indexes of a join index to provide the Optimizer with the information it needs to generate an optimal plan.

The COLLECT STATISTICS (Optimizer Form) statement collects demographics, computes statistics from them, and writes the resulting data into individual entries for each individual base table and join index table on the system.

As far as the Optimizer is concerned, a multitable join index and the base tables it supports are entirely separate entities. You must collect statistics on multitable join index columns separately from the statistics you collect on their underlying base table columns because the column statistics for multitable join indexes and their underlying base tables are *not* interchangeable.

On the other hand, it is generally preferable to collect statistics on the underlying base table of a single-table join index and not directly on the join index columns. See “Guidelines for Collecting Statistics on Single-Table Join Indexes” on page 11-45 and “Collecting Statistics on Base Table Columns Instead of Single-Table Join Index Columns” on page 11-46 for further details on collecting statistics for single-table join indexes.

Guidelines for Collecting Statistics on Multitable Join Indexes

The guidelines for collecting statistics on the relevant columns are the same as those for any regular join query that is frequently executed or whose performance is critical. The only difference with join indexes is that the join result is persistently stored and maintained by the AMP software without user intervention.

Note the following guidelines for collecting statistics for join indexes.

- To improve the performance of creating a join index and maintaining it during updates, collect statistics on its base tables immediately prior to creating the join index.
- Collect statistics on all the indexes defined on your join indexes to provide the Optimizer with the information it needs to generate an optimal plan.
- Collect statistics on additional join index columns that frequently appear in WHERE clause search conditions, especially when the column is the sort key for a value-ordered join index because the Optimizer can then use that

information to more accurately compare the cost of using a NUSI-based access path in conjunction with range or equality conditions specified on the sort key column.

- In general, there is no benefit in collecting statistics on a join index for joining columns specified in the join index definition itself. Statistics related to these columns should be collected on the underlying base tables rather than on the join index.

  The only time you gain an advantage by collecting statistics on a join column of the join index definition is when that column is used as a join column to other base tables in queries where the join index is expected to be used in the Optimizer query plan.

Guidelines for Collecting Statistics on Single-Table Join Indexes

For most applications, you should collect the statistics on base table columns rather than on single-table join index columns. See "Collecting Statistics on Base Table Columns Instead of Single-Table Join Index Columns" on page 11-46 for further information.

You can collect and drop statistics on the primary index columns of a single-table join index using an INDEX clause to specify the column set. For example, suppose a join index has the following definition.

```
CREATE JOIN INDEX OrdJIdx8 AS
SELECT o_custkey, o_orderdate
FROM Orders
PRIMARY INDEX (o_custkey, o_orderdate)
ORDER BY (o_orderdate);
```

Then you can collect statistics on the index columns as a single object as shown in the following example.

```
COLLECT STATISTICS ON OrdJIdx8 INDEX (o_custkey, o_orderdate);
```

Note that you can *only* use columns specified in a PRIMARY INDEX clause definition if you use the keyword INDEX in your COLLECT STATISTICS statement.

A poorer choice would be to collect statistics using the column clause syntax. To do this, you must perform two separate statements.

```
COLLECT STATISTICS ON OrdJIdx8 COLUMN (o_custkey);
COLLECT STATISTICS ON OrdJIdx8 COLUMN (o_orderdate);
```

It is always better to collect the statistics for a multicolumn index on the index itself rather than individually on its component columns because its selectivity is much better when you collect statistics on the index columns as a set.

For example, a query with a predicate like `WHERE x=1 AND y=2` is better optimized if statistics are collected on INDEX (x,y) than if they are collected individually on column x and column y.

The same restrictions hold for the DROP STATISTICS statement.

Collecting Statistics on Base Table Columns Instead of Single-Table Join Index Columns

The Optimizer substitutes base table statistics for single-table join index statistics when no demographics have been collected for its single-table indexes. Because of the way single-table join index columns are built, it is generally best *not* to collect statistics directly on the index columns and instead to collect them on the corresponding columns of the base table. This optimizes both system performance and disk storage by eliminating the need to collect the same data redundantly.

There are two reasons why you might need to collect statistics on single-table join index columns instead of their underlying base table columns.

- If you decide not to collect the statistics for the relevant base table columns for some reason, then you should collect them directly on the corresponding single-table join index columns.
- If the primary index column set consists of more than one column, and there is no primary or secondary index on the base table that includes those columns, then you *must* use the COLLECT STATISTICS ON … INDEX syntax to collect the statistics for that column set on the single-table join index itself.

Guidelines for Collecting Statistics On Single-Table Join Index Columns

The guidelines for selecting single-table join index columns on which to collect statistics are similar to those for base tables. The primary factor to consider in all cases is whether the statistics provide better access plans. If they do not, consider dropping them. If the statistics you collect produce *worse* access plans, then you should always report the incident to NCR support personnel.

When you are considering collecting statistics for a single-table join index, it might help to think of the index as a special kind of base table that stores a derived result. For example, any access plan that uses a single-table join index must access it with a direct probe, a full table scan, or a range scan. With this in mind, consider the following factors when deciding which columns to collect statistics for.

- Always consider collecting statistics on the primary index. This is particularly critical for accurate cardinality estimates.
-

IF an execution plan might involve …	THEN collect statistics on the …
search condition keys	column set that constitutes the search condition predicate.
joining the single-table index with another table	join columns to provide the Optimizer with the information it needs to best estimate the cardinalities of the join.

-

IF a single-table join index is defined ...	THEN you should collect statistics on the ...
with an ORDER BY clause	order key specified by that clause.
without an ORDER BY clause and the order key column set from the base table is not included in the *column_name_1* list	order key of the base table on which the index is defined. This action provides the Optimizer with several essential baseline statistics.

- If a single-table join index column appears frequently in WHERE clause predicates, you should consider collecting statistics on it as well, particularly if that column is the sort key for a value-ordered single-table join index.

Value-Ordered Storage of Join Index Rows

Introduction

The partitioning of the subtable rows for a join index across the AMPs is controlled by its NUPI. By default, join index subtable rows are sorted locally on each AMP by the row hash value of the NUPI column set. You can also specify that rows be stored in value-order.

Value Ordering and Range Queries

You can specify value ordering by means of the optional ORDER BY clause in the join index definition. Sorting a join index by values, as opposed to row hash values, is especially useful for range queries involving the sort key. Value ordering is limited to a single numeric column with a size of 4 bytes or less.

For example, the following join index rows are hash-partitioned using c_name and are value-ordered on each AMP using c_custkey as the sort key.

```
CREATE JOIN INDEX OrdCustIdx AS
SELECT (o_custkey
       ,c_name)
       ,
       (o_status
       ,o_date
       ,o_comment)
FROM Orders LEFT JOIN Customer
ON o_custkey = c_custkey
ORDER BY o_custkey
PRIMARY INDEX (c_name);
```

Example 1: Defining a Simple Join Index on a Binary Join Result

Table Definitions

Suppose you define the following Customer and Orders tables.

```
CREATE TABLE Customer
  (
   c_custkey    INTEGER NOT NULL,
   c_name       CHARACTER(26) NOT NULL,
   c_address    VARCHAR(41),
   c_nationkey  INTEGER,
   c_phone      CHARACTER(16),
   c_acctbal    DECIMAL(13,2),
   c_mktsegment CHARACTER(21),
   c_comment    VARCHAR(127)
  )
PRIMARY INDEX(c_custkey);

CREATE TABLE Orders
  (
   o_orderkey      INTEGER NOT NULL,
   o_date          DATE FORMAT 'yyyy-mm-dd',
   o_status        CHARACTER(1),
   o_custkey       INTEGER,
   o_totalprice    DECIMAL(13,2),
   o_orderpriority CHARACTER(21),
   o_clerk         CHARACTER(16),
   o_shippriority  INTEGER,
   o_comment       VARCHAR(79)
  )
UNIQUE PRIMARY INDEX(o_orderkey);
```

Example Query Statement

Consider the following SELECT statement against these tables.

```
SELECT o_custkey
, c_name
, o_status
, o_date
, o_comment
FROM orders, customer
WHERE o_custkey=c_custkey;
```

The next few topics examine the query plan for this SELECT statement: first without and then with a join index.

Query Plan: No Join Index Defined

Without a defined join index, the execution plan for this query would typically redistribute the Orders table into a spool file, sort the spool on O_CUSTKEY, and then perform a merge join between the spool and the CUSTOMER table.

Query Plan: Join Index Defined

Now consider the execution plan for this same query when the following join index has been defined:

```
CREATE JOIN INDEX OrdCustIdx AS
SELECT (o_custkey, c_name), (o_status, o_date, o_comment)
FROM orders, customer
WHERE o_custkey=c_custkey;
```

With this join index defined, the execution plan for the query specifies a simple scan of the join index without accessing any of the underlying base tables and without having to join them on the predicate WHERE o_custkey = c_custkey.

In the join index defined for this example, (o_custkey, c_name) is the specified fixed part of the index and (o_status, o_date, o_comment) is the repeated portion. Therefore, assume the following specimen base table entries (where ? indicates a null):

Customer

custkey	name	address
100	Robert	San Diego
101	Ann	Palo Alto
102	Don	El Segundo

Orders

orderkey	date	status	custkey	comment
5000	1997-10-01	S	102	rush order
5001	1997-10-01	S	100	big order
5002	1997-10-03	D	102	delayed
5003	1997-10-05	U	?	unknown customer
5004	1997-10-05	S	100	credit
5005	1997-10-08	P	101	discount

Materialized Join Index

The materialized logical join index rows are the following.

OrdCustIdx

Fixed Part		Repeated Part		
custkey	**name**	**status**	**date**	**comment**
100	Robert	S	1997-10-01	big order
		S	1997-10-05	credit
101	Ann	P	1997-10-08	discount
102	Don	S	1997-10-01	rush order
		D	1997-10-03	delayed

Note that the information for the null customer is not included in this join index because it was defined using an inner join.

Join indexes are not limited to binary joins: like any other join, they can be defined on joins involving as many as 64 tables.

Example 2: Using Outer Joins to Define Join Indexes

When Possible, Use Outer Joins in the Join Index Definition

There are several benefits in defining non-aggregate join indexes with outer joins.

- Unmatched rows are preserved.

  These rows allow the join index to satisfy queries with fewer join conditions than those used to generate the index.
- Outer table row scans can provide the same performance benefits as a single-table join index.

  For example, the Optimizer can choose to scan the outer table rows of a join index to satisfy a query that only references the outer table provided that a join index scan would be more performant than scanning the base table or redistributing rows.

Redefined Join Index

The following example changes the previous join index example (see "Example 1: Defining a Simple Join Index on a Binary Join Result" on page 11-49) to use an Outer Join in the join index definition.

```
CREATE JOIN INDEX OrdCustIdx AS
SELECT (o_custkey
       ,c_name)
       ,
       (o_status
       ,o_date
       ,o_comment)
FROM orders
LEFT JOIN Customer ON o_custkey=c_custkey;
```

Materialized Join Index

The resulting join index rows would be the following (where ? indicates a null).

OrdCustIdx

Fixed Part		Repeated Part		
custkey	**name**	**status**	**date**	**comment**
100	Robert	S	1997-10-01	big order
		S	1997-10-05	credit
101	Ann	P	1997-10-08	discount
102	Don	S	1997-10-01	rush order
		D	1997-10-03	delayed
?	?	U	1997-10-05	unknown customer

With the join index defined in this way, the following query could be resolved using just the join index, without having to scan the base tables.

```
SELECT o_status, o_date, o_comment
FROM orders;
```

In this particular case, it is more efficient to access the join index than it is to access the Orders base table. This is true whenever the cost of scanning the join index is less than the cost of scanning the Orders table. The SQL query optimizer evaluates both access methods, choosing the more efficient, less costly of the two for its query plan.

Example 3: Different Join Index Created Using an Outer Join

Introduction

This example examines the EXPLAIN reports for a different outer join-based join index in some detail.

Table Definitions

This example set uses the following table definitions.

```
CREATE TABLE customer
     (c_custkey    INTEGER NOT NULL,
      c_name       CHARACTER(26),
      c_address    VARCHAR(41),
      c_nationkey  INTEGER,
      c_phone      CHARACTER(16),
      c_acctbal    DECIMAL(13,2),
      c_mktsegment CHARACTER(21),
      c_comment    VARCHAR(127))
UNIQUE PRIMARY INDEX(c_custkey);

CREATE TABLE orders
     (o_orderkey      INTEGER NOT NULL,
      o_custkey       INTEGER,
      o_orderstatus   CHARACTER(1),
      o_totalprice    DECIMAL(13,2) NOT NULL,
      o_orderdate     DATE FORMAT 'yyyy-mm-dd' NOT NULL,
      o_orderpriority CHARACTER(21),
      o_clerk         CHARACTER(16),
      o_shippriority  INTEGER,
      o_commment      VARCHAR(79))
UNIQUE PRIMARY INDEX (o_orderkey);
```

```
CREATE TABLE lineitem
     (l_orderkey      INTEGER NOT NULL,
      l_partkey       INTEGER NOT NULL,
      l_suppkey       INTEGER,
      l_linenumber    INTEGER,
      l_quantity      INTEGER NOT NULL,
      l_extendedprice DECIMAL(13,2) NOT NULL,
      l_discount      DECIMAL(13,2),
      l_tax           DECIMAL(13,2),
      l_returnflag    CHARACTER(1),
      l_linestatus    CHARACTER(1),
      l_shipdate      DATE FORMAT 'yyyy-mm-dd',
      l_commitdate    DATE FORMAT 'yyyy-mm-dd',
      l_receiptdate   DATE FORMAT 'yyyy-mm-dd',
      l_shipinstruct  VARCHAR(25),
      l_shipmode      VARCHAR(10),
      l_comment       VARCHAR(44))
PRIMARY INDEX (l_orderkey);
```

Join Index Definition

The following statement defines a join index on these tables. Subsequent examples demonstrate the effect of this join index on how the Optimizer processes various queries.

```
CREATE JOIN INDEX order_join_line
AS SELECT (l_orderkey, o_orderdate, o_custkey,
 o_totalprice),(l_partkey, l_quantity,
 l_extendedprice, l_shipdate)
FROM lineitem
LEFT JOIN orders ON l_orderkey = o_orderkey
ORDER BY o_orderdate
PRIMARY INDEX (l_orderkey);

 *** Index has been created.
 *** Total elapsed time was 15 seconds.
```

EXPLAIN for Query With Simple Predicate

The following EXPLAIN shows how the newly created join index, order_join_line, might be used by the Optimizer.

```
EXPLAIN
SELECT o_orderdate, o_custkey, l_partkey, l_quantity,
       l_extendedprice
FROM lineitem , orders
WHERE l_orderkey = o_orderkey;
```

```
Explanation
------------------------------------------------------------------------
  1) First, we lock a distinct LOUISB."pseudo table" for read on a
     Row Hash to prevent global deadlock for LOUISB.order_join_line.
  2) Next, we lock LOUISB.order_join_line for read.
  3) We do an all-AMPs RETRIEVE step from join index table
     LOUISB.order_join_line by way of an all-rows scan with a condition
     of ("NOT (LOUISB.order_join_line.o_orderdate IS NULL)") into Spool 1,
     which is built locally on the AMPs.  The input table will not be
     cached in memory, but it is eligible for synchronized scanning.
     The result spool file will not be cached in memory.  The size of
     Spool 1 is estimated to be 1,000,000 rows.  The estimated time for
     this step is 4 minutes and 27 seconds.
  4) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

EXPLAIN for Query With More Complicated Predicate

The following EXPLAIN shows how the join index might be used in a query when an additional search condition is added on the join indexed rows.

```
EXPLAIN
SELECT o_orderdate, o_custkey, l_partkey,
       l_quantity, l_extendedprice
FROM lineitem , orders
WHERE l_orderkey = o_orderkey
AND o_orderdate > '1997-11-01';

Explanation
------------------------------------------------------------------------
  1) First, we lock a distinct LOUISB."pseudo table" for read on a
     Row Hash to prevent global deadlock for LOUISB.order_join_line.
  2) Next, we lock LOUISB.order_join_line for read.
  3) We do an all-AMPs RETRIEVE step from join index table
     LOUISB.order_join_line with a range constraint of (
     "LOUISB.order_join_line.Field_1026 > 971101") with a residual
     condition of ("(NOT (LOUISB.order_join_line.o_orderdate IS NULL ))
     AND (LOUISB.order_join_line.Field_1026 > 971101)") into Spool 1,
     which is built locally on the AMPs.  The input table will not be
     cached in memory, but it is eligible for synchronized scanning.
     The size of Spool 1 is estimated to be 1000 rows.  The estimated time
     for this step is 0.32 seconds.
  4) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

EXPLAIN for Query With Aggregation

The following EXPLAIN shows how the join index might be used in a query when aggregation is performed on the join indexed rows.

```
EXPLAIN
SELECT l_partkey, AVG(l_quantity), AVG(l_extendedprice)
FROM lineitem , orders
WHERE l_orderkey = o_orderkey
AND o_orderdate > '1997-11-01'
GROUP BY l_partkey;
```

```
Explanation
-----------------------------------------------------------------------
  1) First, we lock a distinct LOUISB."pseudo table" for read on a
     RowHash to prevent global deadlock for LOUISB.order_join_line.
  2) Next, we lock LOUISB.order_join_line for read.
  3) We do a SUM step to aggregate from join index table
     LOUISB.order_join_line with a range constraint of (
     "LOUISB.order_join_line.Field_1026 > 971101") with a residual
     condition of ("(LOUISB.order_join_line.Field_1026 > 971101) AND (NOT
     (LOUISB.order_join_line.o_orderdate IS NULL ))"), and the grouping
     identifier in field 1.  Aggregate Intermediate Results are
     computed globally, then placed in Spool 3.  The input table will
     not be cached in memory, but it is eligible for synchronized
     scanning.
  4) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of
     an all-rows scan into Spool 1, which is built locally on the AMPs.
     The size of Spool 1 is estimated to be 10 rows.  The estimated time
     for this step is 0.32 seconds.
  5) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

EXPLAIN for Query With Base Table-Join Index Table Join

The following EXPLAIN shows how the join index might be used in a query when join indexed rows are used to join with another base table.

```
EXPLAIN
SELECT o_orderdate, c_name, c_phone, l_partkey, l_quantity,
 l_extendedprice
FROM lineitem, orders, customer
WHERE l_orderkey = o_orderkey
AND o_custkey = c_custkey;

Explanation
-----------------------------------------------------------------------
  1) First, we lock a distinct LOUISB."pseudo table" for read on a
     Row Hash to prevent global deadlock for LOUISB.order_join_line.
  2) Next, we lock a distinct LOUISB."pseudo table" for read on a
     Row Hash to prevent global deadlock for LOUISB.customer.
  3) We lock LOUISB.order_join_line for read, and we lock LOUISB.customer
     for read.
  4) We do an all-AMPs RETRIEVE step from join index table
     LOUISB.order_join_line by way of an all-rows scan with a condition
     of ("NOT (LOUISB.order_join_line.o_orderdate IS NULL)") into Spool 2,
     which is redistributed by hash code to all AMPs.  Then we do a
     SORT to order Spool 2 by row hash.  The size of Spool 2 is
     estimated to be 1,000,000 rows.  The estimated time for this step
     is 1 minute and 53 seconds.
  5) We do an all-AMPs JOIN step from LOUISB.customer by way of a
     Row Hash match scan with no residual conditions, which is joined to
     Spool 2 (Last Use).  LOUISB.customer and Spool 2 are joined using
     a merge join, with a join condition of ("Spool_2.o_custkey =
     LOUISB.customer.c_custkey").  The result goes into Spool 1, which
     is built locally on the AMPs.  The size of Spool 1 is estimated to
     be 1,000,000 rows.  The estimated time for this step is 32.14
     seconds.
  6) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

EXPLAIN for Query Against Single Table

The following EXPLAIN shows how the join index might be used in a query of a single table.

```
EXPLAIN
SELECT l_orderkey, l_partkey, l_quantity, l_extendedprice
FROM lineitem
WHERE l_partkey = 1001;

Explanation
------------------------------------------------------------------------
  1) First, we lock a distinct LOUISB."pseudo table" for read on a
     Row Hash to prevent global deadlock for LOUISB.order_join_line.
  2) Next, we lock LOUISB.order_join_line for read.
  3) We do an all-AMPs RETRIEVE step from join index table
     LOUISB.order_join_line by way of an all-rows scan with a condition
     of ("LOUISB.order_join_line.l_partkey = 1001") into Spool 1, which
     is built locally on the AMPs.  The input table will not be cached
     in memory, but it is eligible for synchronized scanning.  The
     result spool file will not be cached in memory.  The size of Spool
     1 is estimated to be 100 rows.  The estimated time for this step
     is 59.60 seconds.
  4) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

Example 4: Defining and Using a Simple Join Index With an *n*-way Join Result

Introduction

The following example shows the creation of a join index defined with a multiway join result and then shows how the Optimizer uses the join index to process a query on the base tables for which it is defined.

Join Index Definition

The following statement defines a join index with a three-table join using both natural and outer joins.

```
CREATE JOIN INDEX cust_order_join_line AS
SELECT (l_orderkey, o_orderdate, c_nationkey, o_totalprice),
 (l_partkey, l_quantity, l_extendedprice, l_shipdate)
FROM (lineitem LEFT JOIN orders
ON l_orderkey = o_orderkey)
INNER JOIN customer ON o_custkey = c_custkey
PRIMARY INDEX (l_orderkey);

 *** Index has been created.
 *** Total elapsed time was 20 seconds.
```

EXPLAIN for Query With Complicated Predicates

The following EXPLAIN shows how the Optimizer might use the join index for a query that accesses all three of the base tables defined in the index.

```
EXPLAIN
SELECT l_orderkey, o_orderdate, o_totalprice, l_partkey,
 l_quantity, l_extendedprice, l_shipdate
FROM lineitem, orders, customer
WHERE l_orderkey = o_orderkey
AND o_custkey = c_custkey
AND c_nationkey = 10;

 *** Help information returned. 16 rows.
 *** Total elapsed time was 1 second.
```

```
Explanation
---------------------------------------------------------------------------
  1) First, we lock a distinct LOUISB."pseudo table" for read on a
     Row Hash to prevent global deadlock for LOUISB.cust_order_join_line.
  2) Next, we lock LOUISB.cust_order_join_line for read.
  3) We do an all-AMPs RETRIEVE step from join index table
     LOUISB.cust_order_join_line by way of an all-rows scan with a
     condition of ("LOUISB.cust_order_join_line.c_nationkey = 10") into
     Spool 1, which is built locally on the AMPs.  The input table will
     not be cached in memory, but it is eligible for synchronized
     scanning.  The result spool file will not be cached in memory.
     The size of Spool 1 is estimated to be 200 rows.  The estimated time
     for this step is 3 minutes and 57 seconds.
  4) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

Example 5: Single-Table Join Index

Introduction

This example shows how a single-table join index can be used as a substitute for a standard join index to minimize update maintenance while at the same time making join processing more performant than it would otherwise be.

Table Definitions

Suppose you have the following tables that you query frequently using join expressions, and both are very large.

Table Name	Primary Index	Primary Index Type
LineItem	OrderKey	NUPI
Part	PartKey	UPI

The table definitions are as follows.

```
CREATE TABLE LineItem
  (l_OrderKey      INTEGER NOT NULL,
   l_PartKey       INTEGER NOT NULL,
   l_SupplierKey   INTEGER,
   l_LineNumber    INTEGER,
   l_Quantity      INTEGER NOT NULL,
   l_ExtendedPrice DECIMAL(13,2) NOT NULL,
   l_Discount      DECIMAL(13,2),
   l_Tax           DECIMAL(13,2),
   l_ReturnFlag    CHARACTER(1),
   l_LineStatus    CHARACTER(1),
   l_ShipDate      DATE FORMAT 'yyyy-mm-dd',
   l_CommitDate    DATE FORMAT 'yyyy-mm-dd',
   l_ReceiptDate   DATE FORMAT 'yyyy-mm-dd',
   l_ShipInstruct  VARCHAR(25),
   l_ShipMode      VARCHAR(10),
   l_Comment       VARCHAR(44)
   )
PRIMARY INDEX (l_OrderKey);

CREATE TABLE Part
   (p_PartKey         INTEGER NOT NULL,
    p_PartDescription CHARACTER(26),
    p_SupplierNumber  INTEGER
    )
UNIQUE PRIMARY INDEX (p_PartKey);
```

Example Query Statement

A frequently performed query on these tables might be the following.

```
SELECT l_PartKey, p_PartDescription, l_Quantity, l_SupplierKey
FROM LineItem, Part
WHERE l_PartKey=p_PartKey;
```

Decision: Ordinary Join Index or Single-Table Join Index?

You could create an ordinary join index on the LineItem and Part tables, but there is a high cost to keeping this join index updated because each update requires a costly minijoin operation, so the ordinary join index would have to be updated each time either a line item or a new part was inserted, deleted, or updated in the respective primary base tables.

A better solution might be to create a single-table join index on the columns of LineItem that need to be joined frequently with the Part table and then make the primary index for the join index l_PartKey. Single-table join indexes are not cost-free, but the cost of performing the single row updates for a single-table index is far less expensive than the minijoins required by a multitable join index.

Single-Table Join Index Definition

The definition for the join index might look something like this.

```
CREATE JOIN INDEX PartKeyLineItem AS
SELECT l_PartKey, l_Quantity, l_SupplierKey
FROM LineItem
PRIMARY INDEX (l_PartKey);
```

The intent of defining this join index is to permit the Optimizer to select it in place of the base table LineItem in cases like the equality condition l_PartKey = p_PartKey, eliminating the need to redistribute the LineItem table (because its proxy, the join index table PartKeyLineItem, has the same primary index as that of the Part table, so the rows are stored on the same AMP).

This avoids the large redistribution of LineItem, but not the join processing.

General Procedure for Defining a Single-Table Join Index

A general procedure for defining a single-table join index follows.

Step	Action
1	Define a join index on the table to be partitioned.
2	Define a *column_1_name* for each *column_name* in the primary base table to be included in the single-table join index. Do *not* specify *column_2_name*s, *joined_table* clauses, or WHERE clauses. These are not allowed in single table joins. See "Join Index Syntax Terminology" on page 11-14 for definitions of *column_1_name* and *joined_table*.
3	Define the primary index on a different column (or columns) than the primary base table to enhance the join selectivity.
4	If the physical database design warrants, use CREATE INDEX to create one or more NUSIs on the join index.

Example 6: Aggregate Join Index

Introduction

This example shows how a simple aggregate join index can be used to create prejoins with aggregation on one or more of its columns while retaining full database normalization.

Table Definitions

This example set uses the following table definitions.

```
CREATE TABLE customer
  (c_custkey    INTEGER NOT NULL,
   c_name       CHARACTER(26) CASESPECIFIC NOT NULL,
   c_address    VARCHAR(41),
   c_nationkey  INTEGER,
   c_phone      CHARACTER(16),
   c_acctbal    DECIMAL(13,2),
   c_mktsegment CHARACTER(21),
   c_comment    VARCHAR(127)
  )
UNIQUE PRIMARY INDEX (c_custkey);

CREATE TABLE orders
  (o_orderkey      INTEGER NOT NULL,
   o_custkey       INTEGER,
   o_orderstatus   CHARACTER(1) CASESPECIFIC,
   o_totalprice    DECIMAL(13,2) NOT NULL,
   o_orderdate     DATE FORMAT 'YYYY-MM-DD' NOT NULL,
   o_orderpriority CHARACTER(21),
   o_clerk         CHARACTER(16),
   o_shippriority  INTEGER,
   o_comment       VARCHAR(79)
  )
UNIQUE PRIMARY INDEX (o_orderkey);
```

Example Query Statement

Consider the following aggregate join query.

```
SELECT COUNT(*), SUM(o_totalprice)
FROM orders, customer
WHERE o_custkey = c_custkey
AND o_orderdate > DATE '1998-09-20'
AND o_orderdate < DATE '1998-10-15'
GROUP BY c_nationkey;
```

Query Plan: No Aggregate Join Index Defined

Without an aggregate join index, a typical execution plan for this query might involve the following stages.

Stage	Process
1	Redistribute orders into a spool file.
2	Sort the spool file on o_custkey.
3	Merge join the sorted spool file and the customer file.
4	Aggregate the result of the merge join.

Aggregate Join Index Definition

Suppose you define the following aggregate join index, which aggregates o_totalprice over a join of orders and customer.

```
CREATE JOIN INDEX ord_cust_idx AS
SELECT c_nationkey, SUM(o_totalprice(FLOAT)) AS
 price, o_orderdate
FROM orders, customer
WHERE o_custkey = c_custkey
GROUP BY c_nationkey, o_orderdate
ORDER BY o_orderdate;
```

Query Plan for Aggregate Join Index

The execution plan produced by the Optimizer for this query includes an aggregate step on the aggregate join index, which is much smaller than either one of the join tables. You can confirm this by performing an EXPLAIN on the query.

```
EXPLAIN
SELECT COUNT(*), SUM(o_totalprice)
FROM orders, customer
WHERE o_custkey = c_custkey
AND o_orderdate > DATE '1998-09-20'
AND o_orderdate < DATE '1998-10-15'
GROUP BY c_nationkey;

*** Help information returned. 18 rows.
 *** Total elapsed time was 3 seconds.
```

```
Explanation
  1) First, we lock a distinct TPCD."pseudo table" for read on a
     RowHash to prevent global deadlock for TPCD.ord_cust_idx.
  2) Next, we lock TPCD.ord_cust_idx for read.
  3) We do a SUM step to aggregate from join index table
     TPCD.ordcustidx by way of an all-rows scan with a condition of (
     "(TPCD.ord_cust_idx.O_ORDERDATE > DATE '1998-09-20') AND
     (TPCD.ord_cust_idx.O_ORDERDATE < DATE '1998-10-15')"), and the
     grouping identifier in field 1.  Aggregate Intermediate Results
     are computed globally, then placed in Spool 2.  The size of Spool
     2 is estimated to be 1 row.
  4) We do an all-AMPs RETRIEVE step from Spool 2 (Last Use) by way of
     an all-rows scan into Spool 1, which is built locally on the AMPs.
     The size of Spool 1 is estimated with no confidence to be 1 row.
     The estimated time for this step is 0.17 seconds.
  5) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
   statement 1.
```

Example 7: Aggregate Join Index With EXTRACT Function

Introduction

Join index definitions, both simple and aggregate, support the EXTRACT function. This example illustrates the use of the EXTRACT function in the definition of an aggregate join index.

Aggregate Join Index Definition

The index is defined as follows.

```
CREATE JOIN INDEX ord_cust_idx_2 AS
SELECT c_nationkey, SUM(o_totalprice(FLOAT)) AS
 price, EXTRACT(YEAR from o_orderdate) AS o_year
FROM orders, customer
WHERE o_custkey = c_custkey
GROUP BY c_nationkey, o_year
ORDER BY o_year;
```

The aggregation is based only on the year of o_orderdate, which has fewer groups than the entire o_orderdate, so ord_cust_idx_2 is much smaller than ord_cust_idx.

On the other hand, the use for ord_cust_idx_2 is more limited than ord_custidx. In particular, ord_cust_idx_2 can only be used to satisfy queries that select full years of orders.

Example Query Statement

While the join index defined for this example, ord_cust_idx_2, cannot be used for the query analyzed in “Query Plan for Aggregate Join Index” on page 11-65, the following query *does* profit from its use because dates are on year boundaries.

```
SELECT COUNT(*), SUM(o_totalprice)
FROM orders, customer
WHERE o_custkey = c_custkey
AND o_orderdate > DATE '1998-01-01'
AND o_orderdate < DATE '1998-12-31'
GROUP BY c_nationkey;
```

Join Index Design Tips

Introduction

This topic describes various tips that you can use to optimize the functionality of your join indexes.

When to Consider Defining a Join Index

Join indexes are not suited for all applications and situations. The usefulness of a join index, like that of any other index, depends on the type of work it is designed to perform. Always prototype any join index and evaluate its usefulness to the applications it is designed to support before adding it to your production environment. The overhead of updating join index tables can outweigh their benefit in some situations.

The following situations all make a join index a likely performance enhancer.

- Frequent joins of large tables with other large or moderately-sized tables that result in a significant number of the rows from both tables being joined.
- Frequent joins of tables of high degree (having many columns) for which the same relatively small set of columns is repeatedly requested.
- An alternate partitioning sequence for a vertical subset of data in one of the base tables (a so-called “single-table join index”) would remove the necessity of redistributing rows for a frequently made join.
- The overhead in time and storage capacity for the creation and maintenance of a join index does not outweigh its retrieval benefits.

Use Outer Joins to Define Simple Join Indexes

Because join indexes generated from inner joins do not preserve unmatched rows, you should always consider using outer joins to define simple join indexes, noting the following restrictions.

- Inequality conditions are not supported under any circumstances for ON clauses in join index definitions.
- Outer joins are not supported under any circumstances for aggregate join indexes.

Defining join indexes on outer joins enables them to satisfy queries with fewer join conditions than those used to generate the index. See “Example 1: Defining a Simple Join Index on a Binary Join Result” on page 11-49 and “Example 2: Using Outer Joins to Define Join Indexes” on page 11-52 for a demonstration of this property.

Collect Statistics on the Indexes You Define for Join Indexes

Keeping fresh statistics on join indexes is just as critical for join indexes as it is for base tables for optimizing the query plans generated by the Optimizer.

Base a Join Index on Foreign Key = Primary Key Conditions

To avoid storing redundant data, base join index definitions on foreign key = primary key conditions. Add either the primary key or the foreign key to the index definition, but not both because the Optimizer has the intelligence to derive either from the other.

Add Join Constraints That Facilitate Joining to Other Tables

Include join constraint columns that support joining to tables not defined in the join index. It makes no difference whether you assign these columns to the fixed part of the join index or to the repeating part.

For example, you might carry the join columns c_nationkey and l_partkey in the join index to facilitate joining the Nation and Parts tables to the join index.

When the join index is defined using an outer join, use the outer table join column rather than the inner table join column to enhance performance.

Specify a Value-Ordered Sort Key for Range Queries

Sorting a join index by data values, as opposed to row hash values, is especially useful for range queries that involve the sort key.

Note the following two limitations.

- Value ordering is limited to a single numeric column with a maximum value of four bytes.
- The column you specify in the ORDER BY clause must be drawn from the set of fixed columns. You cannot order by a column from the set of compressed columns.

In the following example, the join index rows are hash-distributed across the AMPs on the primary index, c_name. Once assigned to an AMP, the join index rows are value-ordered using c_custkey as the sort key.

```
CREATE JOIN INDEX ord_cust_idx AS
SELECT (o_custkey, c_name), (o_status, o_date, o_comment)
FROM Orders
LEFT JOIN Customer ON o_custkey = c_custkey
ORDER BY o_custkey
PRIMARY INDEX (c_name);
```

Consider the Tradeoffs for Any Join Index

Join indexes can also have properties that make other solutions more optimal. Remember: it is always important to test and validate join indexes on a test system before putting them into a production environment. Keep these potential downside characteristics in mind whenever you consider creating a join index that you think might be useful for your application workload.

- Balance usage broadness with access efficiency. The more columns defined for a join index, the longer the access time for processing the index. The more tables included in a join index, the longer and more burdensome the required maintenance time.
- You cannot use the MultiLoad, FastLoad, and Restore utilities against tables that have join indexes defined on them. See "Load Utilities" on page 11-41 and "Archive and Restore" on page 11-42.
- You cannot define triggers on tables that have join indexes defined on them. See "Triggers" on page 11-41.

Join Index Benefits and Costs

Introduction

Before you create a join index to optimize one or more of your frequently repeated queries, you should consider the following issues carefully.

- Cost of disk resources required to store a join index
- Cost of creating a join index
- Cost of maintaining a join index
- Benefits of reduced performance times for the queries the join index is designed to facilitate.

Definition: Cost

Cost is expressed in two different ways in this chapter.

- With respect to disk resources, cost is expressed in terms of bytes. In other words, the cost is one of space or size, not time. Like fallback, join indexes are space-intensive because they require separate tables, often of fairly high cardinality, for storage.
- With respect to creation and maintenance, cost is expressed in terms of time. Two different measures of temporal cost are used: elapsed time and CPU path per transaction.

Elapsed time is a fairly simple measure, but it is not comparable across different system configurations. It has the advantages of conceptual simplicity and ease of measurement. With respect to join index creation, it is a measure of how long it takes to create the index. With respect to maintenance, it is a measure of how long it takes to update a join index table beyond the time required to perform the update of the base table. In this sense, the term *update* refers generically to the data manipulation operations insert, update, and delete.

CPU path per transaction is a sophisticated measure that is comparable across different system configurations. It has the disadvantages of being both difficult to measure without the proper tools and difficult to understand conceptually.

Whether creation and maintenance cost is measured as elapsed time or as CPU path per transaction, the same proportional relationship holds: the smaller the number, the better.

The total cost for a join index is the sum of its creation and maintenance costs.

Definitions: Benefit and Benefit Percentage

Benefit is a measure of the difference in elapsed time for the same query performed with and without a join index.

Benefit percentage is a normalized expression of benefit. It is defined as the product of benefit and 100 divided by the elapsed query time without a join index.

The larger the number, the better.

Definition: Payback

Payback is an expression of the number of queries that must be run to achieve a break even point for the total cost of the join index. Its value is derived from dividing total costs by benefit.

The smaller the number, the better.

Cost of Disk Resources

A join index table has the same properties as a base table except that it cannot be queried directly.

Because it is essentially a base table, the cardinality of a simple join index is generally of the same order as its component join table with the highest cardinality, with adjustments being necessary for compression and other miscellaneous issues. The cardinality of an aggregate join index is typically much smaller than that of any of its component join tables.

The degree of either type of join index table is entirely dependent on how it is defined.

Irrespective of any other factors, join index tables extract a cost in terms of increased disk space requirements. If a join index table is fallback-protected, then it exerts twice again the burden in disk space.

The disk resources required for a join index are described in "Hash and Join Index Storage" on page 11-74.

Cost of Join Index Creation

The cost of creating a join index is described in terms of time: how long does it take to create the index? Equally important in many situations is the question of how many times a query for which a join index was designed must be performed to offset the creation time by recovering response time.

Different types of join index require a longer or shorter time to create. The time resources required to create various types of join indexes are described in "Creation Costs of Join Indexes" on page 11-79.

Cost of Join Index Maintenance

The largest resource burden for any join index is incurred by its maintenance: inserting, updating, and deleting rows in the join index table. Similar to creation costs, various types of join indexes incur very different maintenance costs.

The time resources required to maintain a join index are described in "Maintenance Costs of Join Indexes" on page 11-83.

Cost/Benefit Analysis of Join Indexes

Any query performs faster when it uses a join index than it does without a join index. The issue that must be examined is whether or not the benefit of the decreased response times provided by a join index offset the costs of its creation, maintenance, and storage.

The relative costs and benefits of defining a join index are described in "Cost/Benefit Analysis for Join Indexes" on page 11-99.

Hash and Join Index Storage

Introduction

The storage organization for both hash and join indexes supports a compressed format to reduce storage space.

If you know that the hash or join index contains groups of rows with repeating information, then its DDL can specify repeating groups, indicating the repeating columns in parentheses. The column list is specified as two groups of columns, with each group in parentheses. The first group contains the repeating columns, and the second group contains the non-repeating columns.

Hash and join indexes can be stored ordered by value of a 4-byte column. This ordered storage provides better performance for queries that contain selection constraints on the value ordering column. For example, suppose a common task is to look up sales information by sales_date. You can create a join index on the sales table and order it by sales_date. The benefit is that queries that select sales by sales_date only need to access those data blocks that contain the value or range of values that the queries specify.

Physical Hash and Join Index Row and Compression

A physical hash or join index row has two parts:

- A required fixed part that is stored only once.
- An optional repeated part that is stored as often as needed.

For example, if a logical join result has *n* rows with the same fixed part value, then there is one corresponding physical join index row that includes *n* repeated parts in the physical join index. A physical hash or join index row represents one or more logical join index rows. The number of logical hash or join index rows represented in the physical row depends on how many repeated values are stored.

When the number of repeated values associated with a given fixed value exceeds the system row size limit, the hash or join index row is automatically split into multiple physical rows, each having the same fixed value but different lists of repeated values. Note that a given logical hash or join index result row cannot exceed the system 64K row size limit.

Guidelines for Using Compression With Hash and Join Index Columns

A column set with a high number of distinct values cannot be compressed because it rarely (and in the case of a primary key, never) repeats. Other column sets, notably foreign key and status code columns, are generally highly nondistinct: their values repeat frequently. Compression capitalizes on the redundancy of frequently repeating column values by storing them once in the fixed part of the index row along with multiple repeated values in the repeated part of the index row.

Typically, primary key table column values are specified as the repeating part and foreign key table columns are specified in the fixed part of the index definition.

"Example 2: Using Outer Joins to Define Join Indexes" on page 11-52 and "Example 3: Different Join Index Created Using an Outer Join" on page 11-54 are examples of how you can use compression to take advantage of repeated column values.

Determining the Space Overhead for a Hash or Join Index

The following equation provides an estimate of the space overhead required for an uncompressed hash or join index. Double the result if you define fallback on the index.

$$\text{Index Size} = U \times (F + O + (R \times A))$$

where:

This symbol...	Defines the ...
F	length of the fixed field *column_1_name.* For a hash index, this length includes the base row identifier information in the fixed field. Add 2 bytes if the base row is defined with a PPI.
R	length of a single repeating field *column_2_name.* For a hash index, this length includes the base row identifier information in the repeating field. Add 2 bytes if the base row is defined with a PPI.
A	average number of repeated fields for a given value in *column_1_name.*
U	number of unique values in the specified *column_1_name.*
O	row overhead. **FOR this type of index ... / Assume this many row overhead bytes ...** join: 14 hash: 12

For definitions of *column_1_name* and *column_2_name*, see "Join Index Syntax Terminology" on page 11-14.

The following equation provides an estimate of the space overhead required for a compressed hash index. Double the result if you define fallback on the index.

$$\text{Compressed hash index size} = N \times (F + O)$$

where:

This symbol …	Defines the …
N	cardinality of the base table.
F	length of the fields, including base row identifier information. Base row headers contain an additional 2 bytes for PPI tables.
O	row overhead. Assume 12 bytes.

Considerations for Measuring Disk Space

Keep the following rules of thumb in mind whenever you perform these calculations.

- Implement your measurements over a very large number of rows to avoid distorting the figures with table overhead issues.
- An INTEGER field uses 4 bytes.
  A SUM(INTEGER) field uses 8 bytes because it is always cast to FLOAT.
- If you have not explicitly specified a COUNT field in your aggregate join index definition, add 4 bytes to the definition to account for the required COUNT field.

Example Measurement 1

Table A and Table B both have 60 million 100-byte rows. An aggregate join index with two INTEGER columns and one SUM(INTEGER) column.

From these figures, the computed size of the aggregate join index is as follows:

$$\text{Row size} = (2 \times 4 \text{ INTEGER bytes}) + 4 \text{ COUNT bytes} + 8 \text{ SUM bytes} + 14 \text{ overhead bytes} = 34 \text{ bytes}$$

This calculation assumes that Table A and Table B are both defined with NPPIs. If a table is defined with a PPI, then you must add 2 additional bytes per row to account for the partition number field.

There is one join index row for every matching row in the base tables. The permanent space specification is as follows:

Current Permanent Space	Peak Permanent Space	Maximum Permanent Space
2 041 595 904 bytes	2 041 595 904 bytes	0 bytes

Using these figures, you can compute a measured row size:

$$\text{Row size} = \frac{2\ 041\ 595\ 904}{60\ 000\ 000} = 34.03 \text{ bytes}$$

Because the measured and computed disk spaces are identical within a narrow confidence interval, it is safe to conclude that the disk space formula is accurate for an aggregate join index.

Example Measurement 2

This example examines the size of a simple join index created for the same tables used in "Example Measurement 1" on page 11-76. There is a savings of 8 bytes because there is no COUNT column and no SUM(INTEGER) column.

The computed row size is as follows.

$$\text{Row size} = 34 \text{ bytes} - (4 \text{ INTEGER bytes} + 4 \text{ SUM(INTEGER) bytes}) = 26 \text{ bytes}$$

The permanent space specification is as follows.

Current Permanent Space	Peak Permanent Space	Maximum Permanent Space
1 561 216 000 bytes	1 561 216 000 bytes	0 bytes

Using these figures, you can compute a measured row size:

$$\text{Row size} = \frac{1\ 561\ 216\ 000}{60\ 000\ 000} = 26.02 \text{ bytes}$$

As with "Example Measurement 1" on page 11-76, the measured and computed disk spaces are identical, and you can conclude that the disk space formula is accurate for a simple join index.

A Note on the Accuracy of These Figures for the Current Release

The numbers presented in this section of the chapter were computed for an earlier release of Teradata and might not reflect current performance measures. Where differences exist, they reflect performance improvements in the current release in every case.

These remarks apply to the following topics:

- "Creation Costs of Join Indexes" on page 11-79
- "Maintenance Costs of Join Indexes" on page 11-83
- "Maintenance Cost as a Function of Number of Hits per Data Block" on page 11-85
- "Maintenance Cost as a Function of Row Size" on page 11-88
- "How to Extrapolate Approximate Join Index Performance from This Data" on page 11-93
- "Maintenance Cost as a Function of Insert Method" on page 11-95
- "Cost/Benefit Analysis for Join Indexes" on page 11-99

Creation Costs of Join Indexes

Introduction

The major cost of a join index is in its maintenance. However, if your processing environment requires that join indexes be created and dropped on a regular basis, the cost of creating the index also becomes a critical factor that must be considered carefully in determining its worth.

Because of this creation overhead, you should conscientiously evaluate the benefit of a join index created to enhance your standard queries vis-a-vis its creation cost if you must drop and create your join indexes frequently. If the gross effect of an index is negative with respect to overall system performance, you might not want to create it even if it enhances the performance of particular queries.

Unfortunately, it is not possible to predict join index creation times with any degree of accuracy because there are too many significant variables involved to be able to produce a useful predictive model. This multivariate complexity also prevents the derivation of a useful predictive model for maintenance overhead or even for query time reductions.

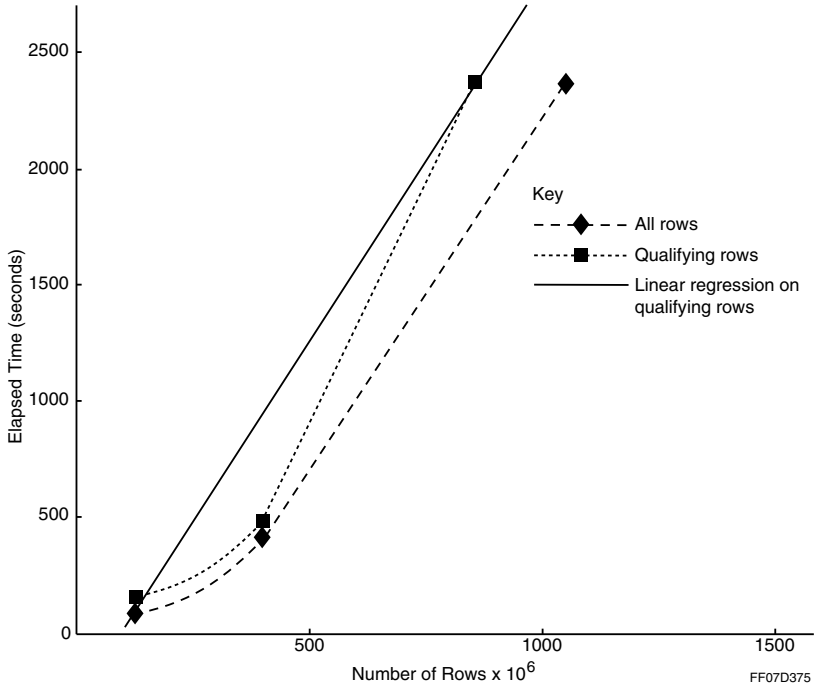
Cost to Create a Join Index

Consider the following example. Empirical testing indicates that one of the most simple relationships is that between the time required to create an in-place aggregate join index and the number of rows in the base tables. Even this simple relationship is complicated by the fact that the resulting creation time is likely to be more closely related to the number of rows that qualify for the join rather than the total number of rows.

The following table indicates a measured relationship between the elapsed creation time for a join index and the number of qualified rows.

Elapsed Time (seconds)	Number of Qualified Rows x 10^6
133	90
415	300
2 344	900

These numbers, along with the total number of rows in each test and a least squares linear regression of number of qualifying rows on elapsed time, are graphed in the following figure.



The equation for this regression is as follows.

$$y = 2.8368x - 255.81$$

This translates into physical terms as follows.

$$\text{Create time} = 2.8 \times \text{Total number qualified rows} - 256$$

where the value for Create time is expressed in elapsed seconds and the order of magnitude for Total number qualified rows is expressed in millions of rows.

The computed coefficient of determination for this regression, R^2, is 0.9819. This means that 98 percent of the variance in the sample data is accounted for by this equation, for which the correlation coefficient is 0.991. In other words, the line is nearly a perfect fit to the data.

A sanity check of the data reveals that this regression might not always be as accurate as it seems to be upon initial scrutiny. For example, consider the 300 million row case. The measured elapsed time value is 415 seconds, but the predicted value is 595.23 seconds, a difference of just over 180 seconds, which is an error of 43 percent.

There are several possible explanations for this discrepancy, but the message to be taken away from this example is that while the model performs well in predicting the general direction of how much time it should take to create a join index with a given number of qualifying rows, the absolute accuracy of the prediction might vary considerably at some points on the curve.

Creation and Elapsed Query Times for Different Join Indexes

The following tables list the creation and elapsed query times for several different kinds of join indexes.

You can use these figures, which are derived from tests on a two node system (see "Test System Configuration" on page E-2 and "Test Table Information" on page E-3) to scale an approximately linear adjustment for your own table sizes and configuration.

Creation Costs and Query Benefits in Terms of Elapsed Query Time

Join Index Type	Creation Cost (seconds)	Elapsed Query Time (seconds)		Query Ratio† (seconds)	Benefit† (seconds)	Benefit Percentage
		Without Join Index	With Join Index			
Single-table aggregate	51	41	< 1	41.00	40	98
2-table in-place aggregate	135	107	< 1	107.00	106	99
4-table in-place aggregate	279	243	< 1	243.00	242	100
Single-table simple	199	80	63	1.27	17	21
2-table in-place	190	116	78	1.49	38	33
2-table in-place outer join	231	147	98	1.50	49	33
2-table foreign aggregate	232	230	< 1	230.00	229	100
2-table ad hoc aggregate	397	357	< 1	357.00	356	100

† Query times of less than 1 second were rounded up to 1.0 seconds for purposes of calculating the displayed value.

Creation Costs and Query Benefits in Terms of CPU Path per Row

Join Index Type	Creation Cost (μseconds)	Query Time CPU Path per Row (μseconds)		Benefit†	Benefit Percentage
		Without Join Index	With Join Index		
Single-table aggregate	9	7	<< 1	40	98
2-table in-place aggregate	7	6	<< 1	106	99
4-table in-place aggregate	10	8	<< 1	242	100
Single-table simple	33	14	11	17	21
2-table in-place	14	10	7	38	33
2-table in-place outer join	16	11	7	49	33
2-table foreign aggregate	19	19	<< 1	229	100
2-table ad hoc aggregate	34	30	<< 1	356	100

† Query times of less than 1 second were rounded up to 1.0 seconds for purposes of calculating the displayed value, which is based on elapsed query time, not CPU path per row time.

<< 1 CPU path per row time was too short to measure.

Maintenance Costs of Join Indexes

Introduction

Join indexes can be expensive to maintain. For many customers, the most important factor in the decision to use a join index is likely to be how much it costs to maintain.

Each time a join-indexed base table column is updated or deleted, the corresponding join index table column must also be updated or deleted. Each time a new row is added to a join-indexed base table, the corresponding join index table columns must also be inserted as a new join index subtable row.

Because of this maintenance overhead, you should always carefully evaluate the benefit of a join index created to enhance your standard queries vis-a-vis its cost to create and maintain.

Types of Join Index Examined for This Analysis

Different types of join indexes incur different costs of maintenance. For the analyses provided here, several different types of join index were used. The types are far from exhaustive, but they provide a fairly representative range of data that you can use to extrapolate roughly how much benefit a join index that uses a particular type of join is likely to provide.

The following table provides a list of the various types of joins defined for the join indexes used in this study. When you think of types of joins, you probably think of join processing types like merge join, nested join, product join, and so on.

The types of joins defined here (in-place, foreign, and ad hoc) are orthogonal to those join processing types. The types of joins and the types of join indexes go hand in hand. In both cases, it is the number of tables that are being redistributed that determines the type of join or join index being described.

Each join type can be used with either a simple or an aggregate join index.

Join Type	Definition
In-place	The joined tables have a common primary index and are joined on that column set. A common example is a logical entity-subentity relationship as seen in an Employee - Employee_Phone relationship, where both tables have Employee Number as a common primary index. As a result, rows to be joined are always on a common AMP and do not have to be redistributed. This is the least expensive join of the three types examined.
Foreign	The tables are joined on a primary key - foreign key column set relationship, where the primary key column set is also the primary index column set. The primary index of the other join table is typically a foreign key in the first table. A common example is a join between an Employee table and a Department table, where the join is on the common Employee Number column, which is the primary index for the Employee table, but is only a foreign key in the Department table. As a result, the rows to be joined must be redistributed from the AMP having the Department table rows to the AMP having the Employee table rows. This join is more expensive than the homogeneous primary index join, but less expensive than the ad hoc join.
Ad hoc	The tables are joined on a column set that is not a primary index in either table. As a result, rows from both tables must be redistributed to make the join. This join is the most expensive of the three types examined.

Maintenance Cost as a Function of Number of Hits per Data Block

Introduction

The number of hits per block is a measure of how many rows in a data block are "touched" (inserted, deleted, or updated) during an operation on a table. Generally speaking, the greater the number of hits per block, the better the performance.

Data Processing Scenarios for the Hit Rates Studied

The following table provides example data processing scenarios that correspond with the hit rates studied.

Number of Hits per Data Block	Percentage of Rows Touched	Scenario
1	0.2	Update a table that contains 500 days of data (roughly 1.5 years) with one typical day of data.
5	1.0	Insert into a table that contains 104 weeks of data (2 years) one typical week of data.
20	4.0	Delete from a table that contains 104 weeks of data (2 years) one typical month (4 weeks) of data

Standard Test Procedure

The standard procedure for these tests was to make any maintenance changes to the left table in the join.

Maintenance Costs in Terms of Elapsed Time

The following table indicates the comparative effects of the number of hits per data block on maintenance costs (as measured by elapsed time) for various types of join index. Test rates of 1, 5, and 20 hits per data block are indicated for insert, delete, and update operations on the join index table.

Maintenance Costs for Standard Join Index Updates in Terms of Elapsed Time (seconds)									
Join Index Type	Maintenance Operation Performed								
	Insert			Delete			Update		
	Number of Hits per Data Block								
	1	5	20	1	5	20	1	5	20
None	132	152	200	92	126	138	90	116	122
2-table in-place aggregate	130	153	202	116	159	166	137	162	172
4-table in-place aggregate	158	191	244	155	194	209	151	177	187
Single-table simple	212	266	319	209	269	286	207	249	264
2-table in-place	128	157	214	121	374	931	190	363	1071
2-table in-place outer join	177	228	283	172	254	582	206	309	628
2-table foreign PI aggregate	149	192	242	200	276	592	252	464	1057
2-table ad hoc aggregate	268	290	348	262	300	304	263	283	291

Note that elapsed times increase as a function of increased hits per data block because more rows are touched. At the same time, the CPU path per row times *decrease* because the elapsed times increase at a lesser rate than the increase in the number of rows touched. These measurements are not tabulated.

Maintenance Costs in Terms of CPU Path Length Per Transaction

The following table shows different numbers derived from the same tests, this time expressed in terms of CPU path length per transaction. For this data, the term *transaction* equates to *qualifying row*. This number is a measure of the amount of CPU time a transaction requires; roughly, the number of instructions performed per transaction.

CPU path per transaction is a better way to compare various manipulations than elapsed time for two reasons:

- CPU path per transaction figures include both CPU utilization information and elapsed time.
- Because the measure is normalized, it factors out configuration-specific variables such as number of CPUs per node and number of nodes, making the numbers comparable across all configurations.

Maintenance Costs for Standard Join Index Updates in Terms of Standard CPU Path per Transaction Ratios

Join Index Type	Maintenance Operation Performed								
	Insert			Delete			Update		
	Number of Hits per Data Block								
	1	5	20	1	5	20	1	5	20
Single-table aggregate	1.0	1.0	1.0	1.3	1.2	1.2	1.9	1.8	1.5
2-table in-place aggregate	1.1	1.2	1.2	2.0	1.6	1.5	2.5	2.2	1.8
4-table in-place aggregate	1.3	1.4	1.4	2.3	1.9	1.8	2.9	2.7	2.3
Single-table simple	1.0	1.0	1.1	3.2	6.3	13.4	3.8	9.7	22.1
2-table in-place simple	1.2	1.3	1.4	3.4	4.3	9.0	4.5	7.6	13.7
2-table in-place outer join	1.1	1.2	1.2	3.3	4.7	9.0	4.9	11.0	23.1
2-table foreign aggregate	3.8	2.9	2.6	6.9	4.5	3.8	9.3	7.6	5.3
2-table ad hoc aggregate	4.0	3.1	2.8	6.9	4.6	3.9	9.5	7.8	5.7

Maintenance Costs for Special Cases

Maintenance data for some more unusual types of DML processing provide additional interesting information. The following table presents the data for two such cases:

- Inserts and deletes to the right table in an in-place outer join
- Updates to a 2-table in-place aggregate join index table using a non-primary index column.

Maintenance Costs for Special Case Join Index Updates in Terms of Standard CPU Path per Transaction Ratios

Join Index Type	Maintenance Operation Performed								
	Insert			Delete			Update		
	Number of Hits per Data Block								
	1	5	20	1	5	20	1	5	20
In-place outer join	1.2	1.4	1.4	2.8	3.8	8.4	Not applicable		
In-place aggregate	Not applicable						2.8	2.2	1.9

Maintenance Cost as a Function of Row Size

Introduction

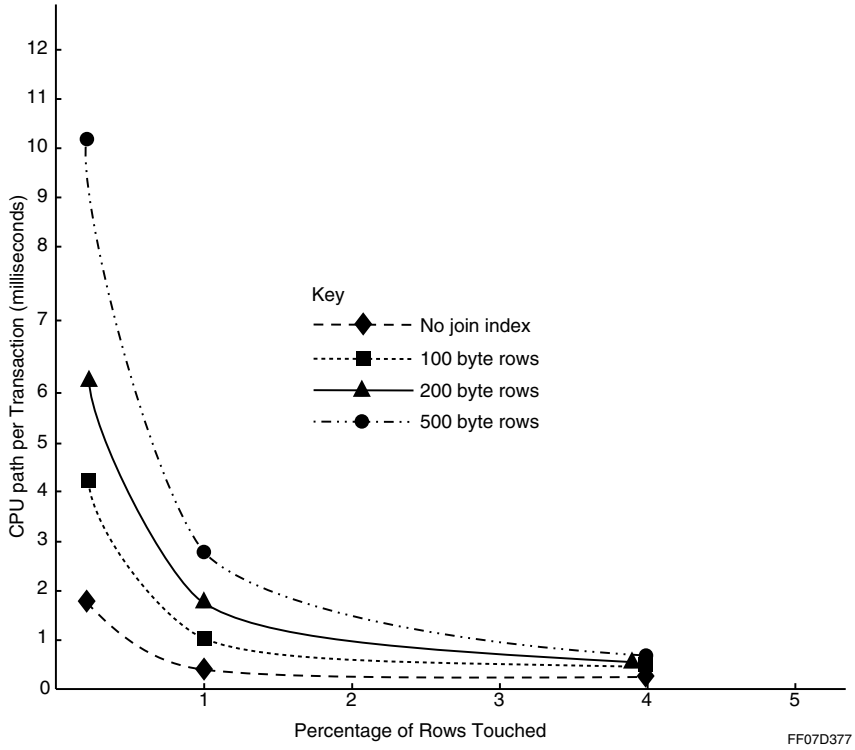
The size of join index rows is another important factor to consider when evaluating the maintenance overhead of a join index. Given a constant disk space, as the size of the rows increases, the number of rows decreases, while the number of data blocks remains the same. The result is that the number of hits per data block decreases.

Data supporting these statements is provided by the following table.

Measured Variable	Percentage of Rows Affected	Row Size (bytes)		
		100	200	500
Total number of rows	Not applicable	4.50×10^7	2.25×10^7	9.00×10^6
Number of rows per data block		650	325	130
Total number of data blocks		6.9×10^4		
Number of rows touched	0.2	9.00×10^4	4.50×10^4	1.80×10^4
	1.0	4.50×10^5	2.25×10^5	9.00×10^4
	4.0	1.80×10^6	9.00×10^5	3.60×10^5
Number of hits per data block	0.2	1.3	0.7	0.3
	1.0	6.5	3.3	1.3
	4.0	26	13	5.2

Maintenance Costs in Terms of CPU Path Per Transaction

CPU path per transaction times increase steadily as a function of increased row size, while the number of transactions, or qualified rows, decreases drastically. This can be seen in the following graph, where CPU path per transaction is presented as a function of percentage of rows touched during an update operation.



The data supporting this graph, as well as data for insert and delete operations, is shown in the following table.

Maintenance Cost in Terms of CPU Path per Transaction (milliseconds) as a Function of Row Size

Operation	Percentage of Rows Affected	Type of Join Index Measured			
		None	Two-Table In-Place Aggregate		
		Row Size (bytes)			
		Not Applicable	100	200	500
Delete	0.2	2.51	5.11	6.69	11.59
	1.0	0.83	1.36	2.06	3.79
	4.0	0.26	0.39	0.60	1.28
Insert	0.2	4.11	4.69	4.97	7.24
	1.0	1.21	1.48	2.05	3.24
	4.0	0.37	0.45	0.66	1.46
Update	0.2	1.78	4.47	5.80	10.32
	1.0	0.45	1.00	1.41	2.68
	4.0	0.17	0.31	0.43	0.81

Maintenance Costs in Terms of Elapsed Time

A corollary result, as indicated by the data in the following table, is that the total elapsed time decreases with increasing row size for all join index cases. The table shows the elapsed time in seconds for delete, insert, and update operations against a 2-table in-place aggregate join index using various constant row sizes, including none for the case where no join index is defined.

Maintenance Cost in Terms of Elapsed Time (seconds) as a Function of Row Size

Operation	Percentage of Rows Affected	Type of Join Index Measured			
		None	Two-Table In-Place Aggregate		
		Row Size (bytes)			
		Not Applicable	100	200	500
Delete	0.2	92	155	126	101
	1.0	126	194	183	143
	4.0	138	209	193	178
Insert	0.2	132	158	100	66
	1.0	152	191	164	125
	4.0	200	244	219	174
Update	0.2	90	151	122	100
	1.0	116	177	170	135
	4.0	122	187	179	165

Maintenance Costs in Terms of I/Os Per Transaction

The following table provides maintenance costs in terms of the number of I/Os incurred per transaction as a function of row size for the same set of operations, row sizes, and join index examined. Note that I/Os also increase as a function of increasing row size.

Maintenance Cost in Terms of Number of I/Os per Transaction as a Function of Row Size					
Operation	Percentage of Rows Affected	Type of Join Index Measured			
		None	Two-Table In-Place Aggregate		
		Row Size (bytes)			
		Not Applicable	100	200	500
Delete	0.2	2.0	3.8	6.9	15.7
	1.0	0.5	0.9	1.7	3.8
	4.0	0.1	0.2	0.5	1.1
Insert	0.2	3.9	4.6	5.5	10.0
	1.0	1.0	1.2	2.0	3.5
	4.0	0.2	0.3	0.5	1.3
Update	0.2	2.1	3.8	6.8	15.7
	1.0	0.5	0.8	1.6	3.8
	4.0	0.1	0.2	0.4	1.0

Maintenance Costs in Terms of Number of Point-to-Point Transactions

Finally, the number of BYNET point-to-point transactions also increases as a function of increasing row size, as indicated by the following table.

Maintenance Cost in Terms of BYNET Point-to-Point Messages per Transaction as a Function of Row Size					
Operation	Percentage of Rows Affected	Type of Join Index Measured			
		None	Two-Table Primary Index Aggregate		
		Row Size (bytes)			
		Not Applicable	100	200	500
Delete	0.2	3.2	3.2	3.6	4.0
	1.0	1.0	1.0	2.0	3.6
	4.0	0.3	0.3	0.5	1.4
Insert	0.2	3.2	4.2	5.1	4.2
	1.0	1.2	1.5	2.8	4.1
	4.0	0.3	0.4	0.8	1.8

Maintenance Cost in Terms of BYNET Point-to-Point Messages per Transaction as a Function of Row Size					
Operation	Percentage of Rows Affected	Type of Join Index Measured			
		None	Two-Table Primary Index Aggregate		
		Row Size (bytes)			
		Not Applicable	100	200	500
Update	0.2	1.7	1.6	1.9	2.2
	1.0	0.5	0.5	0.9	1.4
	4.0	0.1	0.1	0.2	0.6

Summary

Summarizing these results, each join index maintenance transaction performs more work as a function of increased row size as measured by the following list of variables.

- CPU path per transaction
- I/Os per transaction
- BYNET transactions per transaction

How to Extrapolate Approximate Join Index Performance from This Data

Introduction

This topic provides an example of how you can extrapolate the data presented in this chapter to roughly approximate the behavior of join indexes in your own production environment. Keep in mind that this is a very simplistic exercise that is provided only to permit you to make reasoned guesses about how a given join index will behave in a given system configuration.

Scenario

You want to delete 800 000 200-byte rows from a base table with a cardinality of 400 million rows. There is a two-table in-place aggregate join index defined on this base table.

The system has 8 nodes, but is otherwise identical to the test configuration used to produce the numbers presented in this chapter in terms the number of PEs, AMPs, and so on (see “Test System Configuration” on page E-2).

Procedure

The following procedure normalizes the data presented in the tables of elapsed maintenance times for different data manipulation operations in terms of number of hits per data block (see the table on page 11-86) and row size (see the table on page 11-90) for a system configuration other than the one used to derive the original figures.

Step	Action
1	Calculate the percentage of rows being deleted from the base table. $\%\ \text{rows deleted} = \frac{800\,000}{400\,000\,000} \times 100 = 0.2\%$
2	Consult the table on page 11-90 and find that deleting 0.2% of the rows from a table with an in-place aggregate join index and 200-byte rows takes 126 seconds.
3	Consult the table on page 11-88 to determine that there were 225 million 2-byte rows in the data that produced the test result numbers.
4	Calculate how long it should take to complete the deletion of 800 000 rows from the table. $\text{Elapsed time to delete} = \frac{400\,000\,000}{22\,500\,000} = 17.8 \text{ seconds}$
5	Scale the time linearly to account for the number of rows. $\text{Elapsed time} = 126 \times 17.8 \cong 2240 \text{seconds} \cong 38 \text{ minutes}$
6	Adjust the result for an 8-node system. $\text{Adjusted rate} = \frac{8}{2} = 4 \text{ times faster}$
7	Compute the elapsed time to delete the 800 000 rows from your system based on the figures measured from the 2-node test system. $\text{Adjusted elapsed time} = \frac{38}{4} \cong 10 \text{ minutes}$

Maintenance Cost as a Function of Insert Method

Introduction

For myriad reasons, some methods of inserting rows into tables are much more efficient than others. Four different methods of insertion are examined here for their efficiency in lowering the maintenance cost of join indexes. The following methods are reported.

- Case 1: FastLoad and INSERT/SELECT the rows
- Case 2: Drop the join index, insert the rows into the base table, recreate the join index
- Case 3: Tpump the rows into the base table
- Case 4: Insert the rows into the base table using SQL

Case 1

This method uses the following procedure.

Step	Action
1	FastLoad the rows into an empty table.
2	INSERT/SELECT the rows from the freshly loaded table into the table that has a join index defined.

This procedure uses block-at-a-time optimization and is the fastest of the methods examined.

Case 2

This method uses the following procedure.

Step	Action
1	Drop the join index.
2	Insert the rows into the base table using any method.
3	Recreate the join index.

The elapsed time for this method averages to about 1.3 times the elapsed time measured for the method provided in Case 1.

Case 3

This method uses the Tpump utility to insert rows into the base table with a join index. Tpump has several procedural advantages that make it an attractive option, including restartability and resource throttleability.

Because Tpump is a row-at-a-time operation, however, its performance for this application is not optimal. Measurements confirm that Tpump, when performed using an non-optimized run, is more than 200 times slower than the method of Case 1 with respect to elapsed time measurements. Optimization of the run using techniques like sorting input data in different orders can reduce this figure by an order of magnitude in many circumstances, bringing the cost differential down to a more reasonable 20:1 ratio for a 1 hit per data block situation. As the number of hits per data block increases, the cost differential increases because methods that use block-at-a-time methods become increasingly efficient.

Case 4

This method uses the SQL INSERT statement to insert the rows into the base table. The data presented in "Maintenance Cost as a Function of Number of Hits per Data Block" on page 11-85 confirms that this method is slightly better than the method presented in Case 2.

Comparative Elapsed Times to Insert

The following table indicates the comparative elapsed times required to insert the same number of 100 byte rows into a base table and join index table.

Insertion Method	Elapsed Time (seconds)	
Case 1	FastLoad into empty table	26
	INSERT/SELECT into base table with join index	99
	Total	125
Case 2	Drop join index	0
	INSERT/SELECT into base table without join index	59
	Recreate join index	104
	Total	163
Case 3	Tpump rows into base table with join index	25 896

Summary Evaluation

The following table presents a summary evaluation of the insert methods examined in this topic.

Insertion Method	Evaluation
Case 1	Greatest speed
Case 2	Least advantageous method
Case 3	• Relatively slow • Offers other advantages
Case 4	• Simple to perform • Greatest efficiency for a small number of rows

Generalizations Derived from These Tests

The following generalizations are made from these test results.

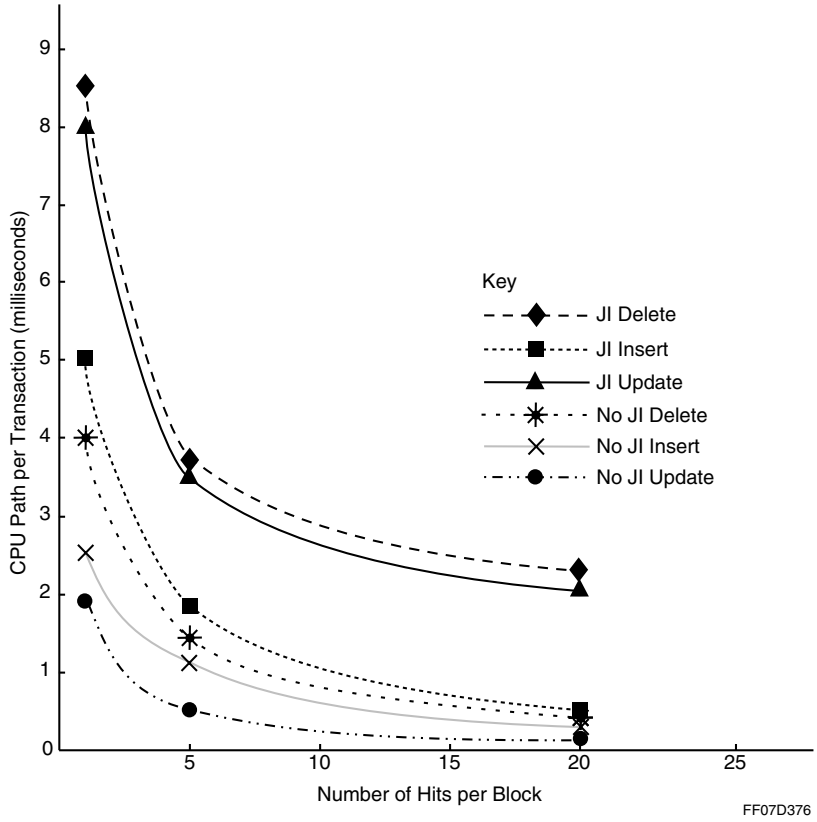
- Maintenance costs for aggregate join indexes are much lower than maintenance costs for comparable simple join indexes.
- Maintenance costs for no join index are lower than maintenance costs for foreign and ad hoc join indexes.

  For example, maintenance costs for a 2-table in-place aggregate join index are 1.1 to 2.5 times greater than maintenance costs computed for just the base table without a join index.
- Maintenance costs vary with the type of join index defined.

  For example, with an in-place aggregate join index, the higher the number of hits per data block, the less the overhead incurred. This effect is marginal for inserts, for which maintenance costs are already minimal.

  On the other hand, for an in-place simple join index at 1 hit per data block, deletes cost 3.4 times more and updates cost 4.5 times more than the case where no join index is defined.

In contrast to the in-place aggregate join index, maintenance costs *worsen* as the number of hits per data block increase, as indicated by the following graph of CPU path per transaction as a function of number of hits per data block.



In the case of an in-place simple join index, inserts cost less than 1.4 times the maintenance required when no join index is defined.

In all cases, inserts are never more than 4 times more expensive than the maintenance cost when no join index is defined.

Cost/Benefit Analysis for Join Indexes

Introduction

This topic introduces the concepts of cost/benefit analyses for various types of join indexes. Several different computations are described to support these analyses including the following.

- Cost
- Benefit
- Benefit percentage
- Payback
- Query ratio

Join Index Benefits Summary

The benefits of join indexes vary considerably among the various types of joins used as well as between simple and aggregate types. The following bulleted list summarizes the conclusions to be drawn from the performance analyses performed.

- All queries that require join processing benefit from join indexes; often dramatically, and sometimes spectacularly.
- In all tests performed, aggregate join indexes strongly outperform simple join indexes, both with respect to their performance and with respect to their maintenance burden. Queries using an aggregate join index frequently run hundreds of times faster than the same queries against the same tables when no join index is defined.
- The benefits of simple join indexes are typically more modest. Queries using a simple join index typically run from 1.3 to 4.9 times faster than the same queries against the same tables when no join index is defined.
- In-place join indexes handily outperform any join indexes that redistribute rows.

Computing the Benefits of Join Indexes

This topic describes a measure you can use to calculate the benefit of a join index.

Begin the calculation by computing some preliminary measures, as described in the following set of equations.

Define the benefit of a join index as follows.

$$\text{Benefit} = ET_{wo} - ET_{w}$$

where:

Syntax element ...	Specifies elapsed query time ...
ET_{wo}	without a join index defined for the query.
ET_{w}	with a join index defined for the query.

Benefit is a simple measure that measures the advantage of a query in terms of the difference between its elapsed completion time when a join index is not defined and its elapsed completion time when a join index *is* defined.

Define the benefit percentage as follows.

$$\text{Benefit Percentage} = (ET_{wo} - ET_{w}) \times \frac{100}{ET_{wo}}$$

The benefit percentage for a query is just a normalized form of the raw benefit. It provides an easily understood measure of the reduction in processing time gained by creating the join index.

Rearrange terms to determine the elapsed time to process a query with the join index defined. This measure is necessary to determine if the cost of creating the join index exceeds its usefulness in reducing response time.

$$ET_{w} = ET_{wo} - \text{Benefit Percentage} \times \frac{ET_{wo}}{100}$$

Computing the Query Ratio

Define the query ratio as follows.

$$\text{Query Ratio} = \frac{\text{Elapsed query time without join index}}{\text{Elapsed query time with join index}}$$

The query ratio is a normalized measure of how much faster a query executes with a join index than without a join index. For example, consulting the table on "Creation Costs and Query Benefits in Terms of Elapsed Query Time" on page 11-81, you find a column of calculated query ratios.

You interpret these ratios as follows. Consider the query ratio for the creation cost of a single-table aggregate join index. The reported query value for this join index is 41. This means that the test query runs 41 times faster with the single-table aggregate join index defined than it does without it.

Computing the Payback Factor

This topic explains how to calculate the payback factor for a join index. Note that the term Costs is something you must measure. It is the time required to compute the join index being investigated.

$$\text{Payback Factor} = \frac{\text{Costs}}{\text{Benefit}}$$

where:

Syntax element …	Specifies …
Costs	the processing time required to create the join index.

Because the term *payback factor* represents a number of queries, it is always rounded up to represent a whole number. The smaller the payback factor, the sooner benefits accrue from using the join index.

Example of Computing a Payback Factor for a Simple Join Index

Suppose you have a frequently performed, nonaggregate join query that you think might benefit from a join index. You decide to test the join index and collect the following data.

Parameter	Measured Value (seconds)
ET_{wo}	416
ET_{w}	226
Cost	1055

What is the payback factor for creating this join index?

$$\text{Benefit} = 416 - 226 = 190 \text{ seconds}$$

$$\text{Benefit Percentage} = 190 \times \frac{100}{416} = 46\%$$

The interpretation of the benefit percentage is that the time to complete the query is cut nearly in half.

$$\text{Payback Factor} = \frac{1055}{190} = 6$$

The interpretation of the payback factor is that the cost of creating the join index is recovered if the query is run six or more times.

Example of Computing a Payback Factor for an Aggregate Join Index

In the scenario supporting this example, a 30 day period is used for the analysis. You do not regularly create and drop join indexes in your production environment, but you want to calculate the cost of creating the index.

The query to be supported by the join index joins two tables and uses aggregation. The join is made on the primary index of both tables. This is the definition of an in-place aggregate join index.

You have a 2-node production system with the same configuration described in Appendix E: "Supporting Information for Join Index Performance Analysis." The following table indicates the demographics of your data.

Data Characteristic	Table A	Table B
Row length (bytes)	100	
Cardinality	450 million rows	600 million rows
Number of rows for each value in the aggregating column	10 000	
Number of rows updated weekly	900 000	Not applicable
Data block size	64 Kbytes	

How many times must the query be performed to recover the cost in the first month? Here is the procedure.

Step	Action
1	Estimate the benefit. The benefit should be approximately one order of magnitude greater than the figure generated by this study because there are that many more rows in the base tables, while all other factors are identical. Examine the table on page 11-81. The measured benefit is 106 seconds. Multiply this by 10 to produce the estimated benefit for your configuration, which is 1060 seconds.
2	Estimate the cost of creation. Examine the table on page 11-81. The cost of creation for an in-place aggregate join index is seen to be 135 seconds. Scale this value up by an order of magnitude for the same reason as Step 1. The estimated cost of creation is 1350 seconds.

Step	Action
3	Estimate the approximate number of hits per block. You update 900 000 100-byte rows in a 450 million row table weekly. Calculate the number of blocks these 450 million 100-byte rows occupy. The equation is as follows. $\text{Number of blocks} = \frac{450\,000\,000 \times 100}{65\,024} \cong 700\,000$ Determine the approximate hit rate. With approximately 700 000 blocks and 900 000 rows to be updated weekly, the hit rate is determined as follows. $\text{Number of hits per block} = \frac{900\,000}{700\,000} \cong 1$
4	Determine the update cost. In a 30 day period, the table will be updated four times. By examining the table on page 11-86, you discover that the cost of this update is 151 seconds for the test system. Scale this number up by an order of magnitude and multiply it by four weeks to obtain the approximate update cost. $\text{Update cost} = 4 \times 1510 = 6040 \text{ seconds}$
5	Calculate the payback factor. $\text{Payback} = \frac{1\,350 + 6\,040}{1\,060} = 7 \text{ queries}$

If the query for which this join index was designed is run twice weekly, it recovers the time spent creating and maintaining it.

Two More Brief Examples

The following examples work through the following join indexes.

- In-place simple join index
- Ad hoc aggregate join index

Follow the identical procedure to that used in the detailed example.

For the in-place simple join index, the calculation is as follows.

$$\text{Payback} = \frac{1900 + (2064 \times 4)}{380} = 27 \text{ queries}$$

To recover its cost in a 30 day period, this query must be run slightly more than once per business day.

For the ad hoc aggregate join index, the calculation is as follows.

$$\text{Payback} = \frac{3970 + (2510 \times 4)}{3560} = 4 \text{ queries}$$

To recover its cost in a 30 day period, this query must be run once per week.

Join Index Costs Summary

Creation and maintenance costs for join indexes can be a resource burden because of their processing overhead. In the case of join index maintenance, the burden is an ongoing process that lasts for the life of the index.

Costs vary considerably among the various types of joins used as well as between simple and aggregate types. The following bulleted list summarizes the conclusions to be drawn from the performance analyses performed.

- Maintenance overhead for in-place aggregate join indexes ranges from 1.0 to 2.9.
- Maintenance overhead for aggregate join indexes that redistribute rows ranges from 2.6 to 9.5.
- Maintenance overhead for simple join indexes ranges from 1.0 to 23.1, increasing as the number of hits per data block increases.
- Maintenance overhead for aggregate join indexes decreases as the number of hits per data block increases because of the efficiencies of block-at-a-time operations in the file system.
- Maintenance overhead for insert operations is significantly less than the maintenance overhead required for update and delete operations.

Section 4:

Physical Database Design, Part 2: Integrity

Chapter 12:

Designing for Database Integrity

This chapter focuses on an important database design issue related to the integrity of databases: database constraints. First and foremost, enforcing database integrity means getting the correct results. This is fundamental.

Constraints are a physical implementation of the business rules under which an enterprise operates. Their purpose is to ensure not only that bad data does not get into the database, but also that intertable relationships do not become corrupted by the improper deletion or updating of data from the existing database.

The emphasis of this chapter is on declarative constraints: constraints that are part of the definition of the database itself[1]. The principal types of declarative constraints described are as follows:

- Column-level constraints
- Table-level constraints
- Database constraints

The recommendation for constraints is to specify them anywhere you can. If you cannot trust your data, there is little point in maintaining databases that contain it.

If performance issues make declarative constraints an unattractive option, then you should *always* implement integrity constraints by some other mechanism. The most important consideration must be that database integrity is consistently enforced.

---

1 It is also possible to implement procedural support for integrity using database features such as triggers and stored procedures. The principal drawbacks to procedural enforcement of database integrity are the added programming requirements and, critically, the increased opportunity for errors in enforcement introduced by these methods.

The least favorable method for enforcing database integrity is through application programming. Besides the increased programming burden and its likely introduction of errors into integrity enforcement, application programming introduces the additional fault of application specificity. If one application enforces database integrity and another does not, or if two programs enforce integrity in different, perhaps contradictory ways, then a still greater chance of corrupting the database results. Worse still, application-based database integrity cannot affect ad hoc inserts, deletes, and updates to the database, and as a result place still further burdens on the DBA to find other mechanisms of preventing database corruption.

The point of relational databases is to be application-neutral, thus serving all applications equally well.

Integrity Issues

Why Integrity Constraints?

Integrity constraints are a method of restricting database updates to a set of specified values or ranges of values. They are one method of implementing business rules in a logical database design, triggers being the other most commonly used method.

Teradata also provides a feature known as a Referential Constraint, which provides the Optimizer with a means for devising better query plans, but which does *not* enforce the referential integrity of the database. See "Referential Constraints" in the description of the CREATE TABLE column definition clause in the chapter "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4* for more information.

Definitions

Term	Definition
Business rule	A component of the business model that defines specific conditional modes of activity. Business rules are expressed in natural language and are represented in a relational database by integrity constraints. They are the ultimate determining factor for proper database design.
Constraint	A predicate that must evaluate to true if a database delete, insert, or update is to be permitted.
Integrity constraint	A component of the logical database model that formalizes business rules by specifying the boundary conditions and ranges permitted for various database parameters. Integrity constraints are one of the four components of an integrity rule and are expressed in the language of databases: tables, columns, rows, and so on.

Term	Definition
Integrity rule	A set of rules for ensuring the integrity of a database. Each integrity rule is composed of four components: • A name • An integrity constraint set • A checking time • A violation response The checking time specifies the processing point at which the constraint is checked. In the ANSI SQL standard, a checking time has either of the following two possible values: • Immediate • Deferred All Teradata constraints have an immediate checking time. Deferred constraint checking, which is never a good way to ensure integrity, is not permitted. The violation response specifies the action to be taken when an integrity constraint is violated. In the ANSI SQL standard, a violation response has either of the following two possible values: • Reject • Compensate All Teradata constraints have a reject violation response. Compensate violation responses are not permitted. Integrity rules are specified by the SQL CREATE TABLE and ALTER TABLE statements.

Databases, Tables, Rows, and Constraints Are Formal Logical Entities

Introduction

The pursuit of theory is frequently unnecessary in the day-to-day practice of administering relational database management systems, but a small dose of theory is essential to understanding the issues you encounter when designing and maintaining databases. Concepts that otherwise appear to be complicated and confusing turn out to be rather obvious once the underlying principles have been mastered. As an anonymous wit once observed, "the gap between theory and practice is not as wide in theory as it is in practice." In other words, without understanding the formal building blocks of relational technology, it is easy to land oneself in considerable trouble in real world practice.

The extent to which practical relational database management can be formalized using the principles of set theory[2] and formal logic is seldom appreciated. When an everyday practice such as database management can be abstracted to such a degree, it becomes trivial to determine whether a particular operation against the database is logically correct or not. The importance of this capability is far from trivial. Indeed, it has been argued that *any* error in logic is a major error, and this contention is surely true with respect to any database upon which the correctness of decision making for an enterprise relies, because if the information in the database is not correct, it is not just worthless, it seriously endangers the viability of the enterprise it is designed to support.

2 The correspondence between relational set theory and relational database theory is not always direct. While it is true that relational database theory is based on the relations of set theory, it necessarily introduces the additional constructs listed below:

- Database relations are typed, while set theory relations are not.
- Database relations are not ordered left-to-right, while set theory relations are.
- Dependency theory is critical to database relations, while no corresponding theory exists for set theory relations.
- Relation variables are critical to database relations, while no corresponding theory exists for set theory relations. As a result, the concept of integrity is foreign to set theory relations.

Terminology

Before proceeding further, some definitions for terms used in predicate logic and the predicate calculus are in order.

Term	Definition
Existential quantifier	The symbolic quantifier ∃ of predicate logic, signifying the logically equivalent English language phrases “for some,” “for any,” or “there exists.”
Identity predicate	The symbolic operator = of predicate logic, signifying the logically equivalent English language phrases “is identical to” or “is equal to.”
Inference rules	The rule set of a formal system that determines the steps of reasoning that are valid for proving logical propositions.
Predicate	A truth-valued function. The attributes of a relation (columns), as well as the relation heading itself, can be represented formally as logical predicates. This is true whether an explicit constraint is defined over the column or not, because when no explicit constraint is defined, the implicit constraint specified by the data type for the column specifies its minimum, essential, constraint. You cannot, for example, insert the character string ‘character string’ into a column typed as INTEGER without first converting the string into an integer value. This type of constraint is known as a domain constraint (see “Domain Constraints” on page 12-12).
Predicate calculus	The set of inference rules by which propositions in predicate logic are proven.
Predicate logic	The study of statement validity using the truth-functional operators of the propositional calculus, the universal and existential quantifiers, and the identity predicate.
Proposition	A verbal statement that can be proven unequivocally to be either true or false. In a table, all rows in any database are assumed to be true propositions by default, because if they were false, they would have been prevented from entry in the database by the various integrity constraints, both implicit and explicit, defined on that database. This important property is sometimes called the Closed World Assumption (see “The Closed World Assumption” on page 12-8).

Term	Definition
Truth-valued function	A function that evaluates unequivocally to either true or false.
Universal quantifier	The symbolic quantifier ∀ of predicate logic, signifying the English language phrase “for all.”

Relations and Relation Variables

As the relational model has matured, careful analysis has continued to reveal facets of its definition that are obvious in hindsight, but had remained undiscovered and unstated until recently. An important example of such a revelation of the obvious is the relation variable, or relvar, first described by Date and Darwen (1998), which actually derives from a statement made by Codd (1970), where he writes that a database is “a collection of time-varying relations … of assorted degrees. As time progresses, each *n*-ary relation may be subject to insertion of additional *n*-tuples, deletion of existing ones, and alteration of components of any of its existing *n*-tuples.” Note that relations, as defined in set theory, do *not* vary over time. A relation is a value, and values do not vary as a function of time. Variables, on the other hand, take on different values as a function of time, so it is formally correct to state that the values of a relation variable change as a function of time. What Codd almost certainly meant to say was something like this: a database is a collection of time-varying relation variables of assorted degrees.

In the context of business rules and their relationship with integrity constraints, the concept of the relvar is useful because it provides a formal, yet simple, framework in which to situate the idea of integrity constraints. Note that the relvar concept applies equally well to views (virtual relvars) and base tables (base relvars).

The first principle captured by the concept of the relvar is that the value of a relation is orthogonal to its defining variable. The easiest way to explain this is by analogy. A variable in any programming language represents the possible values that can be taken for that variable, but it is not those values: it is a place holder for them. The values represented by the program variable can take on any number of values, but the variable itself is always an abstraction.

Think of a relvar as the heading definition for the attributes of a relation: it is the ANDing of all the column headings, including their constraints, defined for the relation.

The relation value, then, is all the data held by the relation. Each time the relation is updated, its value changes, but its relvar does not.

For example, consider the following equation:

$$x + y = 7$$

If you set the variable x=4, then y must be 3. If you then change the value of the variable x to 3, then you have updated x, but you have not changed the value of

4 to 3, just the value of the variable *x* from 4 to 3, or any other valid number over the domain of *x*. Another way to think of this is as follows: values are not time varying. A 3 is always a 3 and never a 4. On the other hand, a variable can represent any number of different values over the course of time.

The integrity of this update relationship is maintained by the relvar predicate, which is defined by the logical ANDing of all the integrity constraints defined on the relation in question. The result of enforcing the relvar predicate is what Date and Darwen term The Golden Rule: no statement can leave any relvar with a value that violates its relvar predicate. In other words, the integrity of the database cannot be violated.

It follows that tuples (“rows” in everyday language) are also predicates that represent various propositions about the relation they constitute. When you think of a row as a proposition, it immediately follows that it must also be a *true* proposition because, by definition, false propositions (wrong, or corrupt data in everyday language) are not permitted in a database that enforces integrity constraints (see “The Closed World Assumption” on page 12-8). If they were, the database would not represent facts and the Golden Rule would be violated. In other words, if a given tuple is an element of a particular relvar, then that tuple, by default, satisfies its predicate by evaluating to true.

Unfortunately, the representation of the real world by a database is only as good as its input, and any database will permit the insertion of factually wrong data as long as that data does not violate the constraints specified for the base table or view through which the table is updated. In other words, a database can only enforce consistency, not truth. All truths are consistent, but not all consistent things are true. This is an important consideration: the database itself cannot know whether its information corresponds to real world facts. It can, however, constrain the data with which it is updated by ensuring that no business rules are violated, and it is these constraints that provide the mechanism for ensuring the success of the data integrity rules enforced by relational database management systems (see “Business Rules” on page 12-11).

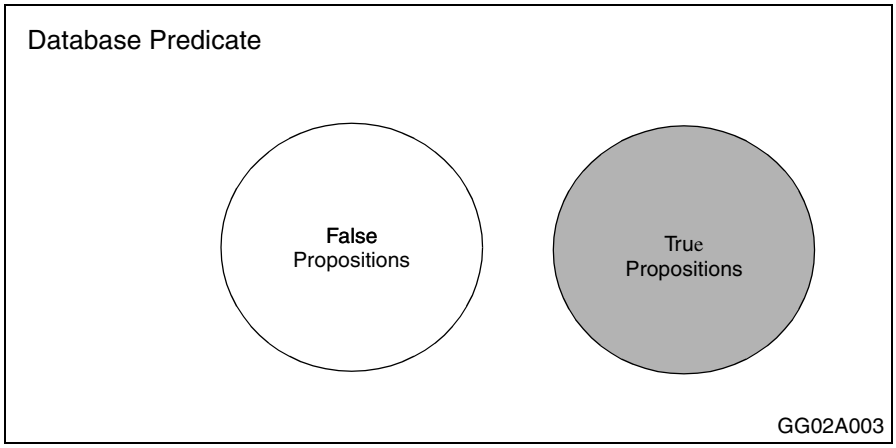
The Logic of Database Integrity

To ensure that no row in the database represents a logically false proposition, each table in the database must be capable of evaluating its own predicate, or relvar, for its truth value, and the only way it can enforce this integrity is to enforce the set of business rules defined by the enterprise it supports. In practice, this means that the predicate for any table is its sole criterion for update acceptability (McGoveran and Date, 1994). From this, it follows that the formal meaning of any table is defined by the logical ANDing of its component column and table constraints. This logically ANDed constraint set is also the best approximation the database management system can make to any table predicate.

From this, it follows that the meaning of an entire database, its predicate, can be represented formally as the logical ORing of its table constraints logically ANDed with the set of all database constraints.

The Closed World Assumption

The Closed World Assumption, or CWA, asserts that if an otherwise valid tuple does not appear in the body of a relation, then the proposition representing that tuple must be false. In logic, such an assertion is called a completion axiom. It follows that a relation contains all of, and only those, tuples whose corresponding propositions are true. In other words, a relation body contains only those tuples that satisfy the completion axiom for its relvar. By generalization, the complete set of completion axioms for a given database defines its CWA, and the set of all tuples in a given database must satisfy its CWA. The following Venn diagram illustrates this point.



Note that, by the definition of the CWA, the sets of false and true propositions for a database do not intersect. See "The Closed World Assumption Revisited" on page 13-5 for an application of the CWA to missing information in relational databases and the paradox that nulls present to the definition of a well-formed completion axiom.

An Illustration of How Databases Are Built From Logical Propositions

Introduction

The following example illustrates how a real database is built from logical propositions.

An Airline Reservation System

As a concrete example of the logical underpinnings of database constraints, consider an airline reservation. The following proposition can be stated about any reservation, and a particular instance of the proposition can be represented by a unique row in a table:

> The reservation identified by reservation number (Reservation_Number), made for flight (Arrival_Flight_Number), has a scheduled arrival date (Arrival_Date) and time (Arrival_Time) at gate (Arrival_Gate_Number) and a scheduled departure flight of (Depart_Flight_Number) with departure date (Depart_Date) and time (Depart_Time) from gate (Depart_Gate_Number).

The variables enclosed within parentheses in this proposition are placeholders for the entire domain of values they represent. When you substitute actual values for these placeholders, you produce a proposition about a specific airline reservation. For example, suppose you have the following set of values:

```
(88079, 317, 10/19/2001, 13:59, 60, 1138, 10/25/2001, 05:40, 82)
```

When these values are substituted into the proposition framework, they produce the following proposition:

> The reservation identified by reservation number 88079, made for flight 317, has a scheduled arrival date 10/19/2001 and time 13:59 at gate 60 and has a scheduled departure flight of 1138 with departure date 10/25/2001 and time 05:40 from gate 82.

This proposition, which represents an instantiation of the table predicate, might be represented in a Flight_Reservations table by the following row:

Res_Num	A_Flt_Num	A_Date	A_Time	A_Gate	D_Flt_Num	D_Date	D_Time	D_Gate
PK								
88079	317	10/19/2001	13:59	60	1138	10/25/2001	05:40	82

Several obvious constraints can be developed for the columns and table supporting this proposition. Stated as simple sentences, these are:

- Column constraints:
  - Reservation numbers must be unique values drawn from a defined integer domain.
  - Arrival and departure flight numbers must be unique values drawn from a defined integer domain.
  - Arrival and departure dates must be unique values drawn from a defined date domain.
  - Arrival and departure times must be unique values drawn from a defined time domain.
  - Arrival and departure gate numbers must be unique values drawn from a defined integer domain.
- Table constraints:
  - Departure dates must be greater than or equal to arrival dates.
  - Departure times must be greater than arrival times if they occur on the same date.
  - Arrival flight numbers and departure flight numbers cannot be equal if they occur on the same date.

And so on.

Business Rules

Business rules are the codified guidelines by which an enterprise operates. They can range from the most simple (“all employees must be at least 16 years old”) to very complex (“if a petroleum cargo is worth less than one million US dollars and is being shipped to an offshore destination, then it must be carried by a supertanker having a deadweight greater than 400 thousand tons and it must be insured by a US carrier for its exact value in US dollars”).

As the name suggests, business rules are derived from the enterprise business model and are literally the rules by which the corporation engages in commerce. Because business rules are stated as natural language propositions, they must be transformed into appropriate database constraints and stored in the data dictionary as table attributes in order to enable the database management system to enforce them.

Business rules are captured during the interview process (see “Interviewing for Requirements” on page 3-12) and specified during the ATM process (see “Column Names and Constraints” on page 7-10 and “Constraints Form” on page 7-20).

Integrity Constraint Types

Introduction

Business rules are expressed in relational databases by means of various types of integrity constraints. Four types of integrity constraint are supported:

- Domain
- Column
- Table
- Database

These four types of constraints are implemented in distinct ways in Teradata, though the differences between column and table constraints are subtle.

Domain Constraints

Fundamentally, a domain is a data type. Data types act as simple constraints by not allowing a user to, for example, insert a 20 character data string into a field typed INTEGER (5).

The domain constraints for your databases are developed from the ATM Domains form (see "Domains Form" on page 7-17).

Column-Level Constraints

Column-level constraints define more elaborate integrity rules for columns than those defined by domain constraints. For example, a column constraint on a column typed as INTEGER(5) might further specify that only values between 0 and 49999 or between 99995 and 99999 are permitted for that column.

A column constraint specifies a simple predicate that applies to one column only. For example, if some business rule states that employee numbers in the Employee table must be between 10001 and 32001 inclusive. You could specify this rule in the CREATE TABLE statement used to define the EmpNo column in the Employee table as follows:

```
CONSTRAINT EmpNo CHECK (EmpNo >= 10001 AND EmpNo <= 32001)
```

Notice that the only column referred to by this constraint is the employee number column, EmpNo.

The column constraints for your databases are developed from the ATM Constraints form (see "Constraints Form" on page 7-20).

Table-Level Constraints

Table-level constraints specify sets of relationships among values within a row. A table constraint can be either single row or multirow.

For example, you might specify a table constraint on the Flight_Reservations table (see "An Airline Reservation System" on page 12-9) that requires the value for the scheduled arrival date, A_Date, to be less than or equal to the value for the scheduled departure date, D_Date. This is a single-row table constraint. The SQL definition for this constraint looks like this:

```
CONSTRAINT ArrivDateCheck CHECK (A_Date >= D_Date)
```

Notice that this constraint refers to two columns, A_Date and D_Date; therefore, it must be a table constraint.

You probably want to ensure that the value for Res_Num is unique and non-null, so you would define a uniqueness constraint on the table for reservation numbers. In this case, Res_Num is the primary key for the Flight_Reservations table, so the constraint is called a primary key constraint. The SQL definition looks like this:

```
Res_Num INTEGER NOT NULL CONSTRAINT pKey PRIMARY KEY
```

Primary key constraints are a subset of uniqueness constraints: all primary keys are also unique, but all unique fields are not primary keys. Because any unique column set is, by definition, also a candidate key, uniqueness constraints are sometimes referred to as key constraints. The primary key constraint on Res_Num is also a multirow table constraint because it enforces the rule that every row in the table has at least one unique value: its reservation number.

Suppose reservation number is not the primary key for the Flight_Reservations table, but your business rules require the values for Res_Num to be unique. To do this, you write a constraint that enforces uniqueness on the Res_Num column. The SQL definition looks like this:

```
Res_Num INTEGER NOT NULL CONSTRAINT ResNumUnique UNIQUE
```

The table constraints for your databases are developed from the ATM Constraints form (see "Constraints Form" on page 7-20).

Database Constraints

Database constraints specify some sort of functional determinant between two or more tables. The most commonly observed database constraint is the primary key-foreign key relationship whose enforcement is referred to as maintaining referential integrity. This constraint specifies that you cannot delete a row having primary key value *x* from table X as long as any foreign key value in table Y has value *x* on the column set that defines the referential integrity relationship between those columns in tables X and Y. In other words, if you have foreign key value *x*, then you must also have primary key value *x* if referential integrity is defined for those keys and tables.

The specific constraint used to define the common PK-FK referential integrity constraint is a `FOREIGN KEY … REFERENCES … table_name … primary_key_name` constraint.

Though less frequently used, it is also possible to specify and enforce database constraints on non-PK-non-FK column relationships if the columns defining those relationships are alternate keys.

The specific constraint used to define an alternate key referential integrity constraint is a `REFERENCES … table_name … alt_key_name` constraint, where *alt_key_name* refers to an alternate key column set in the parent table.

The database constraints for your databases are developed from the ATM Constraints form (see "Constraints Form" on page 7-20).

Constraint Specifications

Introduction

A name and a data type must be specified for each column defined for a table. Each specified column can be further defined with one or more attribute definitions.

There are several different specifications for constraints, some of which apply to multiple categories of constraints (see "Integrity Constraint Types" on page 12-12).

The following subclauses are SQL column definition attributes that specify column-level integrity constraints:

- CHECK constraint definition clause on a single column
- PRIMARY KEY constraint definition clause on a single column
- REFERENCES constraint definition clause on a single column
- UNIQUE constraint definition clause

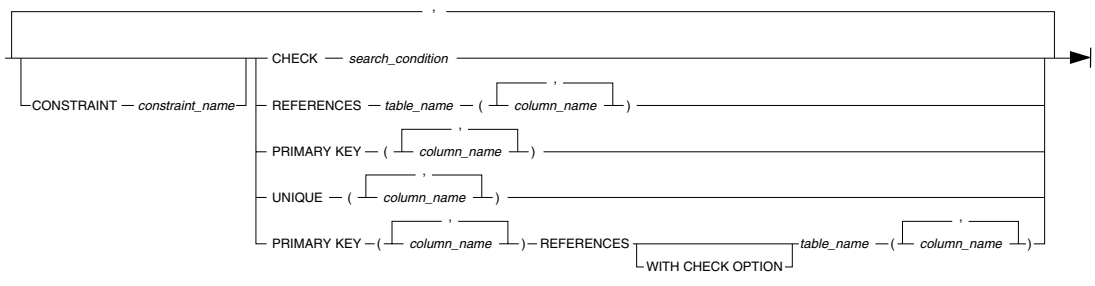
The following subclauses are SQL table definition attributes that specify table-level and intertable integrity constraints:

- CHECK constraint definition clause on multiple columns
- REFERENCES constraint definition clause on multiple columns
- PRIMARY KEY constraint definition clause on multiple columns
- UNIQUE constraint definition clause on multiple columns
- FOREIGN KEY constraint definition clause

  FOREIGN KEY constraint definitions must also specify a REFERENCES clause.

Constraint Syntax

The syntax for specifying the various constraint types is indicated by the following diagram. Note that the WITH NO CHECK OPTION phrase is not shown because Referential Constraints are not used to ensure database integrity.



Naming Constraints

Constraint specifications can, and should, be named. Naming constraints is a good practice to follow because it permits a programmer to debug an embedded SQL or stored procedure application by fetching the name of a violated constraint from the SQLSTATE area. This makes debugging simpler because it can be very difficult to determine which constraints belong to which system-defined names.

Constraint names must conform to the rules for Teradata object names and be unique among all other constraint, primary index, and secondary index names specified on the table.

The characters used to name a constraint can be any of the following:

- Uppercase and lowercase alphabetic characters
- Integers
- Any of the following special characters:
  - -
  - #
  - $

Unnamed constraints are not assigned system-generated names.

CHECK Constraints

CHECK constraints are the most general type of SQL constraint specifications. Depending on its position in the CREATE or ALTER TABLE text, a CHECK constraint can apply either to an individual column or to an entire table.

The following rules apply to all CHECK constraints:

- CHECK constraints can be defined at column-level or at table-level.
- The specified predicate for a CHECK constraint can be any simple boolean search condition.

  Subqueries, aggregate expressions, and CASE expressions are not valid search conditions for CHECK constraint definitions.
- You cannot specify CHECK constraints for volatile tables.
- A combination of table-level, column-level, and WITH CHECK OPTION on view constraints can create a constraint expression that is too large to be parsed for INSERT and UPDATE statements.
- Unnamed CHECK constraints with identical wording and case are considered to be duplicates, and the system returns an error when you specify them.

  For example, the following CREATE TABLE statement is valid because the case of f1 and F1 is different:

  ```
  CREATE TABLE t1 (f1 INTEGER, CHECK (f1>0), CHECK (F1>0));
  ```

  The following CREATE TABLE statement is not valid and returns an error.

  ```
  CREATE TABLE t1 (f1 INTEGER, CHECK (f1>0), CHECK (f1>0));
  ```

- The principal difference between defining a CHECK constraint at column-level or at table-level is that column-level constraints cannot reference other columns in their table, while table-level constraints, by definition, *must* reference other columns in their table.

The following rules apply to column-level CHECK constraints:

- You can specify multiple constraints on a single column.

  If more than one *unnamed* CHECK constraint is defined for a column, then the system combines them into one column constraint.

  However, each *named* column-level CHECK constraint is handled separately, as if it were a table-level named CHECK constraint.
- A column-level CHECK constraint cannot reference any other columns in its table.

The following rules apply to table-level CHECK constraints:

- A table-level constraint must reference at least two columns from its table.
- Table-level CHECK constraint predicates cannot reference columns from other tables.
- A maximum of 100 table-level constraints can be defined for any table.

REFERENCES Constraints

REFERENCES constraints permit you to specify foreign key-like relationships between a unique column set in the current table and an alternate key column set in a different table.

A REFERENCES clause is also required to modify a FOREIGN KEY constraint by identifying the primary key set being referenced in a parent table.

You can also use a REFERENCES clause to specify any of the following referential constraint types:

- Standard referential integrity
- Batch referential integrity
- Referential constraint

The following table summarizes the differences among the different referential constraints:

Referential Constraint Type	DDL Definition Clause	Does It Enforce Referential Integrity?	Level of Referential Integrity Enforcement
Referential integrity constraint	REFERENCES	Yes	Row
Batch referential integrity constraint	REFERENCES WITH CHECK OPTION	Yes	Implicit transaction
Referential constraint	REFERENCES WITH NO CHECK OPTION	No	None

The various types have different applications as described in the following table:

Referential Constraint Type	Application
Referential integrity constraint (see "Standard Referential Integrity" in "CREATE TABLE" in the chapter "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*)	• Tests each individual inserted, deleted, or updated row for referential integrity. • If insertion, deletion, or update of a row would violate referential integrity, then AMP software rejects the operation and returns an error message. • Permits special optimization of certain queries.
Batch referential integrity constraint (see "Batch Referential Integrity" in "CREATE TABLE" in the chapter "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*)	• Tests an entire insert, delete, or update batch operation[a] for referential integrity. • If insertion, deletion, or update of any row in the batch violates referential integrity, then parsing engine software rolls back the entire batch and returns an abort message. • Permits special optimization of certain queries.
Referential constraint (see "Referential Constraints" in "CREATE TABLE" in the chapter "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*)	• Does not test for referential integrity. • Assumes that the user somehow enforces referential integrity in a way other than the normal declarative referential integrity constraint mechanism. • Permits special optimization of certain queries.

a. In this context, a batch operation is defined as an implicit SQL transaction.

Referential Constraints do not enforce primary key-foreign key constraints between tables, so they avoid the overhead of RI enforcement by the system as practiced by standard and batch referential integrity constraints. Their only purpose is to provide the Optimizer with a means for devising better query plans. Referential Constraints should be used only for situations for which referential integrity is either not important or is enforced by other means, because its use implicitly instructs the system to trust the validity of all DML statements made against the affected columns and not to check the specified referential integrity relationships.

Caution If referential integrity errors occur, then data corruption can occur. Erroneous results can be returned if the DML statement contains a redundant RI join and the primary key-foreign key rows do not match.

See "CREATE TABLE Column Definition Clause" in *Teradata RDBMS SQL Reference, Volume 4* for more information.

The following rules apply to both column- and table-level REFERENCES constraints.

- Each column in the REFERENCES clause should correspond with a column of the referenced table, and the same column name should not be specified more than once.
- The referencing column list should contain the same number of column names as the referenced column list. The ith column of the referencing list corresponds to the ith column identified in the referenced list. The data type of each referencing column should be the same as the data type of the corresponding referenced column.
- The user issuing the CREATE TABLE statement containing a REFERENCES clause must either have the REFERENCE privilege on the referenced table or on all specified columns of the referenced table.
- If REFERENCES is specified in a *column_constraint*, then *table_name* defines the referenced table. *table_name* must be a base table, not a view.
- Referential constraints are not supported for global temporary or volatile tables.

The following rules apply to column-level REFERENCES constraints only.

- If *column_name* is omitted, then the referenced table must have a single column primary key, and the specified foreign key column references that primary key column of the referenced table.
- If *column_name* is specified, then it must refer to the single column primary key of the referenced table or to a single column alternate key in the referenced table defined as UNIQUE.

The following rule applies to table-level REFERENCES constraints only.

- A maximum of 100 table-level constraints can be defined for any table.

FOREIGN KEY Constraints

FOREIGN KEY constraints specify the foreign key columns, if any, in a table definition. All FOREIGN KEY constraints must be followed by a REFERENCES constraint clause that specifies the parent table column set for the foreign key. The database system uses foreign key constraints to enforce referential integrity (see "The Referential Integrity Rule" on page 5-37).

All FOREIGN KEY constraints are table-level constraints by definition.

The following rules apply to foreign key constraints.

- A maximum of 64 foreign keys can be defined on or about a table.

  This limit includes both references to and from the table. For example, if 10 tables (t_1, t_2, … t_{10}) reference table t_n, then only 54 (64 - 10 = 54) foreign keys can be defined on table t_n.

  Column-level CHECK constraints that reference alternate keys in other tables do not count against this limit.
- Each individual foreign key can be defined on a maximum of 16 columns.
- A maximum of 100 table-level constraints can be defined for any table.

PRIMARY KEY Constraints

PRIMARY KEY constraints specify the primary key columns in a table definition. The database system uses primary keys to enforce both row uniqueness and referential integrity.

Whether a PRIMARY KEY constraint is treated as a column-level constraint or a table-level constraint depends on whether the primary key is simple or composite.

The following rules apply to primary key constraints.

- Only one primary key can be defined for a table.
- Defining a primary key for a table is never required, though it is very strongly recommended as part of a policy of enforcing data integrity.
- A primary key can be defined on a maximum of 16 columns.
-

PRIMARY KEY constraints are treated as …	When the primary key is defined on this many columns …
column-level constraints	1
table-level constraints	> 1

  A maximum of 100 table-level constraints can be defined for any table.
- When a primary key is defined on a column set that is not identical to the unique primary index column set defined for the table, then the primary key is implemented physically as a UNIQUE NOT NULL secondary index.

UNIQUE Constraints

UNIQUE constraints specify that the column set they modify must contain unique values. Teradata implements UNIQUE constraints as unique secondary indexes (USI).

The following rules apply to UNIQUE constraints:

- UNIQUE constraints can be defined at column-level (simple) or at table-level (composite).
- Column-level UNIQUE constraints refer only to the column on which they are specified.
- Table-level UNIQUE constraints can be defined on multiple columns using a column name list.
- A maximum of 100 table-level constraints can be defined for any table.
- The internal implementation of the UNIQUE constraint is as a UNIQUE NOT NULL secondary index.

Constraint Enforcement

The enforcement of a constraint depends on how the base table on which it is defined is accessed. If the base table is accessed directly, then its column and table constraints are always enforced. Date (2001a) calls this "The Golden Rule": No update operation must ever assign to any database a value that causes its database predicate to evaluate to false.

By this definition, the checking time must always be immediate for any update; that is, the constraint check is made at statement boundaries, not deferred to be checked at transaction (COMMIT time) boundaries. If this were not the case, then inconsistent, or false, data could be entered into the database, even if only for a brief time and even if the inconsistency were, as it must be, private to the transaction in question. It would still be possible for a query that followed this inconsistent update within the boundaries of an explicit transaction to report erroneous information. Teradata does not support deferred integrity checking.

If a base table is accessed by means of a view, then the enforcement of any WHERE clause constraints specified in the view definition depends on whether the view is also defined WITH CHECK OPTION or not (see "Integrity Constraints for Updatable Views" on page 12-25).

Views inherit the constraints of their underlying base tables; therefore, base table constraints cannot be violated by updating through a view. However, additional constraints defined by means of a WHERE clause specification in a view definition *can* be bypassed if the view is not also defined WITH CHECK OPTION. The result is not a violation of any constraints defined on underlying base tables, but the insertion of a row that cannot be seen from that view.

Updatable Cursors and Database Integrity

Introduction

Positioned updates using updatable cursors can present integrity problems unless strict locking protocols are observed. Cursor updates[3] must be performed within the same transaction as the SELECT statement that defined the cursor.

Between Transaction Integrity Issues

By default, updatable cursors use READ locks, which are implemented as row hash locks internally. READ locks are adequate for preserving database integrity because they prevent access to the database by users who attempt to either change the definitions of database structures or to update table data. As a result, stored data cannot be changed while an open cursor is also manipulating the same data in such a way that it updates the same fields of the same rows. This is sometimes referred to as repeatable read mode. Programmers can also upgrade the locks used by updatable cursors to WRITE or even EXCLUSIVE locks if they so desire.

The potentially most severe problems for database integrity regarding updatable cursors are presented by the use of ACCESS or CHECKSUM[4] locks, because both of these locks permit rows to be updated by other users while an open cursor is manipulating them, thus providing greater concurrency. CHECKSUM locks provide better integrity than simple ACCESS locks because they compute a checksum value for each row updated by an open cursor (see the chapter "Transaction Processing" in *Teradata RDBMS SQL Reference, Volume 2* for details). When the current transaction commits and an updated row is written to disk, the checksum value for that row is compared with the value computed for the row at the time it was accessed and if the values differ, the update is not permitted.

There is a finite probability that pre- and post-updated rows will produce an identical checksum value, so this option is only advisable for those situations where performance concerns significantly override integrity concerns. The most important fact to understand about the various locking options for positioned updates via updatable cursors is that there is no integrity risk with READ and more restrictive locks, but there is a finite integrity risk with CHECKSUM locks, and an almost certain integrity risk with ACCESS locks.

---

3 As used here, the word *update* also includes deletes.

4 CHECKSUM locks *are* ACCESS locks. The difference is that CHECKSUM locks compute a checksum value for updated rows and compare it with the checksum value computed for the row at the time the cursor accessed it, while ACCESS locks provide no such integrity check.

Within Transaction Integrity Issues

"Between Transaction Integrity Issues" on page 12-23 describes the integrity risks presented by concurrent updating of a table by different transactions. This topic describes the risks associated with cursor conflicts that occur within an individual transaction.

Cursor conflicts occur in the following situations:

- Two cursors are open on the same table at the same time. One attempts a positioned update, but the other has already made a positioned update on the same row, thereby modifying the table.
- While a cursor is open, a searched update is made on a row, and then the cursor attempts a positioned update on that row.
- While a cursor is open, a positioned update is made on a row through that cursor and then a searched update is attempted on the same row.

All three of these updates are dirty because the same row was updated twice without the transaction having been committed.

Note that the system permits all three of the dirty updates to occur and then returns a cursor conflict warning to the requesting program. Resolution of the integrity breach is left to the user.

Because of the multiple possible integrity risks associated with declaring and opening multiple cursors within the same transaction, you should code your applications so those risks cannot occur.

Integrity Constraints for Updatable Views

Introduction

Besides defining constraints on your base tables, you can specify additional constraints on a user-by-user basis through updatable views by specifying constraints in a WHERE clause and enforcing them by specifying a WITH CHECK OPTION clause. Because views inherit the constraints defined for their underlying base tables as well as those defined for any intermediate views, the constraints they inherit from their underlying relations are called derived constraints.

Specifying Integrity Constraints in an Updatable View Definition

Not all views are updatable. You can tailor update constraints at the user level by specifying user-specific constraints in the WHERE clause of a view as long as you also specify a WITH CHECK OPTION clause.

WITH CHECK OPTION pertains only to updatable views. Views are typically created to restrict which base table columns and rows a user can access in a table. Base table column projection is specified by the columns named in the *column_name* list, while base table row restriction is specified by an optional WHERE clause.

The WITH CHECK OPTION clause is an integrity constraint option that restricts the rows in the table that can be affected by an INSERT or UPDATE statement to those defined by the WHERE clause. If you do not specify WITH CHECK OPTION, then the integrity constraints specified by the WHERE clause are not checked for updates. Derived constraints inherited from underlying relations are not affected by this circumvention and continue to be enforced. The problem is that the view that updated the row in a way that violates the WHERE clause cannot view the updated row, so it cannot be updated in the future through that view.

WHEN WITH CHECK OPTION is …	THEN any insert or update made to the table through the view …
specified	only inserts or updates rows that satisfy the WHERE clause.
not specified	ignores the WHERE clause used in defining the view.

The following rules apply to updatable views and the WITH CHECK OPTION.

- If WITH CHECK OPTION is specified, the view is updatable. Any insert or update to the table through the view is rejected if the WHERE clause predicate evaluates to false.
- If WITH CHECK OPTION is not specified in an updatable view, then any WHERE clause contained in the view definition is ignored for any insert or update action performed through the view. In other words, the specified integrity constraints in the view definition are ignored, so you should always specify a WITH CHECK OPTION clause to define constrained updatable views unless you have an extraordinary reason not to.
- You can define nested views that only reference a single base table, which might allow the views to be updatable. In this case, the specification of a WITH CHECK OPTION clause in the view definition permits the WHERE clause constraints for that view, as well as those defined for any underlying views, to be exercised in the constraint on INSERT or UPDATE. See "Updatable View Inheritance" on page 12-28 for more information.

Example 1

The following statement creates a view of the Employee table so that it provides access only to the names and job titles of the employees in Department 300:

```
CREATE VIEW Dept300 (Name, JobTitle) AS
 SELECT Name, JobTitle
 FROM Employee
 WHERE DeptNo = 300
 WITH CHECK OPTION;
```

The WITH CHECK OPTION clause in this example prevents you from using the Dept300 view to update a row in the Employee table for which DeptNo <> 300.

Example 2

This example shows how using a view in an UPDATE, INSERT, or DELETE statement allows you to add, change, or remove data in the base table set on which the view is defined.

Consider the following Staff_Info view, which provides a personnel clerk with retrieval access to employee numbers, names, job titles, department numbers, sex, and dates of birth for all employees except vice presidents and managers:

```
CREATE VIEW staff_info (number, name, position,
department, sex, dob) AS
 SELECT employee.empno, name, jobtitle, deptno, sex, dob
 FROM employee
 WHERE jobtitle NOT IN ('Vice Pres', 'Manager')
 WITH CHECK OPTION ;
```

If the owner of Staff_Info has the insert privilege on the Employee table, and if the clerk has the insert privilege on Staff_Info, then the clerk can use this view to add new rows to Employee.

For example, this statement inserts a row into the underlying Employee table that contains the specified information:

```
INSERT INTO staff_info (number, name, position, department,
sex, dob)
 VALUES (10024, 'Crowell N', 'Secretary', 200, 'F', 'Jun 03
 1960');
```

The constraint on Staff_Info illustrated by the following WHERE clause applies to any insert using this view that includes the WITH CHECK OPTION phrase.

```
…
WHERE jobtitle NOT IN ('Vice Pres', 'Manager')
…
```

Therefore, the preceding INSERT statement would fail if the Position entered for employee Crowell was Vice Pres or Manager.

If this view were defined without the WITH CHECK OPTION, and a user had the update privilege on the table, that user could update a job title to 'Vice Pres' or 'Manager'. The user would be unable to access the changed row through the view.

The following statement changes the department number typed for Crowell in the preceding INSERT statement from 200 to 300:

```
UPDATE staff_info
 SET department = 300
 WHERE number = 10024 ;
```

Performing the following DELETE statement removes the row for employee Crowell from the staff_info table.

```
DELETE
 FROM staff_info
 WHERE number = 10024 ;
```

Views are a useful method for permitting selected users to have restricted access to base table data. However, as the preceding examples suggest, granting another user insert, update, and delete privileges on a view means relinquishing some control over your data. Carefully consider granting such privileges.

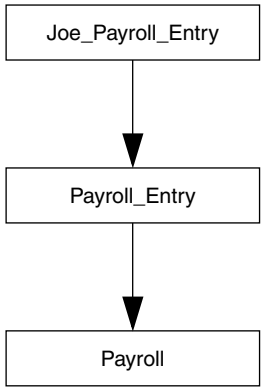
The default is *not* to constrain updated or inserted values unless the view definition explicitly includes WITH CHECK OPTION.

Updatable View Inheritance

All base table integrity constraints are enforced without exception. Another way of stating this is to say that updatable views inherit the integrity constraints of their underlying base tables.

Because views can be nested, it is also true that updatable views inherit the integrity constraints of their underlying updatable views (when such relationships exist) as derived constraints in addition to inheriting the integrity constraints of their underlying base tables.

For example, suppose you have two nested views defined on the Payroll table as follows:



GG02A005

When user Joe logs on, he is assigned the view Joe_Payroll_Entry defined with the following integrity constraint:

```
WHERE dept_no = 1350
WITH CHECK OPTION
```

Joe can only update values for department 1350.

View Joe_Payroll_Entry is defined on top of view Payroll_Entry, which restricts the ability to update payroll for any employee who earns a base salary of 200 000 USD or greater. This constraint is specified by the following WHERE clause in the view definition:

```
WHERE base_salary < 200000
WITH CHECK OPTION
```

Because of this derived constraint, Joe can only update annual salaries less than 200 000 USD in department 1350.

View Payroll_Entry is defined on top of base table Payroll, which restricts the ability to update payroll to only those employees who are US citizens or who have valid work visas. This constraint is specified by the following column-level CHECK constraint in the table definition:

```
visa_code CHARACTER(2)
  CONSTRAINT ok2work
    CHECK (visa_code IN ('US','WV') )
```

Because of this additional derived constraint, Joe can only update salaries of US citizens or individuals with valid work visas who work in department 1350 and earn annual salaries less than 200 000 USD.

References

Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, 13(6):377-387, 1970.

Date, C.J., *What Now How: The Business Rules Approach to Application Development*, Reading, MA: Addison-Wesley, 2000.

Date, C.J., *Constraints and Predicates: A Brief Tutorial, Parts I, II, and III*, http://www.firstsql.com/dbdebunk, 2001a.

Date, C.J., *Integrity Constraints, Parts 1, 2, and 3*, http://www.brcommunity.com, 2001b.

Date, C.J. and Hugh Darwen, *Foundation for Object/Relational Databases: The Third Manifesto*, Reading, MA: Addison-Wesley, 1998.

Date, C.J. and David McGoveran, "A New Database Design Principle," *Database Programming and Design*, 7(7):46-53, 1994.

Pascal, Fabian, *Practical Issues in Database Management: A Reference for the Thinking Practitioner, Chapter 2: The Rule of Rules: Integrity*, Reading, MA: Addison-Wesley, 2000.

von Halle, Barbara, *Business Rules Applied: Building Better Systems Using the Business Rules Approach*, New York: John Wiley and Sons, 2001.

Chapter 13:

Designing for Missing Information

The efficacy and usefulness of the manner in which missing information is handled by SQL is in the eye of the beholder.[1] Perhaps no other topic in relational database management generates as much controversy as does the dispute over the proper way to deal with missing information in relational databases.

The problems resulting from recording and manipulating missing information are concerned principally with the sometimes ambiguous, and often counterintuitive, meaning of nulls as reported by various types of database queries.

This chapter examines the various semantic ambiguities with nulls in order to make designers aware of the potential problems of interpretation they present. The principal thrust is to make you aware of the inconsistencies in the way SQL handles missing information. Careful use of nulls can provide benefits that cannot otherwise be achieved: at the same time, their careless use can present serious problems not only for the integrity of your databases, but also for the accuracy of the information you retrieve from them.

The recommendation for nulls is not to use them if you can avoid doing so. Constrain as many of your columns as possible to be NOT NULL. A carefully considered database design is often all that is required to avoid recording most nulls.

1 Independent of the semantic difficulties SQL presents when dealing with nulls, missing information also makes exploratory data analysis far more difficult than the identical data mining operation performed on data free of missing values. For example, see Wu and Barbará (2002).

Semantics of SQL Nulls

Introduction

The meaning of SQL nulls is well established in both the original vendor SQL dialects and in the succession of ANSI SQL standards, though that meaning is frequently misunderstood. In brief, for most applications, an SQL null simply means that the value of interest is not known. The only exceptions to this occur when nulls are used to represent the empty set[2].

Types of Missing Values

The possible explanations for the meaning of SQL nulls are many. The following list touches on most of the common uses:

- Value is unknown
- Value is not applicable
- Value does not exist
- Value is not defined
- Value is not valid
- Value is not supplied
- Value is the empty set

In the real world, the semantics, properties, and behavior of each of these null types are different, but SQL treats them identically. Codd (1990) has proposed a revision of SQL logic that includes a second type of null: value not applicable, which would mean extending the current 3-valued first-order predicate logic of SQL (3VL, having the possible predicate evaluations TRUE, FALSE, and UNKNOWN, the latter called MAYBE by Codd) to a 4-valued first-order predicate logic (4VL, having the possible predicate evaluations TRUE, FALSE, UNKNOWN, and NOT APPLICABLE, the latter two called MAYBE BUT APPLICABLE (designated with an A-mark) and MAYBE BUT NOT APPLICABLE (designated with an I-mark), respectively).

The problems with the 3VL now supported by SQL are well documented (see Date, 1990ab, 1992abc; McGoveran, 1993-94; Pascal, 2000), and the formal arguments used to justify the deprecation of 3VLs are not repeated in any detail here. Arguments supporting 3VL have also been made (see E.F. Codd in Codd and Date, 1993; Fesperman, 1998; Johnston, 1995ab), and detailed rejoinders from 2VL (no nulls allowed) supporters have been published in response (see C.J. Date in Codd and Date, 1993; Date, Darwen, and McGoveran, 1995, 1997).

2 The empty set is an actual value in mathematics, not a place holder for missing information, but SQL uses the null marker to indicate empty sets.

Summary of Inconsistencies in How SQL Treats Nulls

Introduction

Recall that the defined semantics for SQL nulls is as the simple representation for missing information (see "Semantics of SQL Nulls" on page 13-2). A null is to be interpreted as missing information and nothing more (see footnote 2 on page 13-2 for an important counterexample). There are a number of areas in which the SQL treatment of nulls is inconsistent with this definition, and the purpose of this section is to summarize those inconsistencies.

Inconsistencies in the SQL Treatment of Nulls

The following summary of the inconsistencies found with the treatment of nulls by SQL is intended to assist you with the interpretation of results that might otherwise be misleading.

- The empty set, which is a well-defined value in mathematics, evaluates as null in SQL. For example, the evaluation of an empty character string returns null rather than the mathematically correct empty set. Similarly, aggregation over empty sets report null, which is meant to mean that the value is missing, but there are known results having real values for such operations on empty sets.

  The problem with empty sets is also true for the outer join, where the extended columns in the join result are denoted by nulls, but they are actually empty sets. As a result, nulls are used to represent both empty sets and missing information in the same report.
- Nulls have no value, but sort as if they were equal.
- With the exception of COUNT(*), aggregate operations ignore nulls, while arithmetic operations do not.
- With the exception of CASE, COALESCE, and NULLIF, SQL expressions cannot return or operate on nulls.
- When a CASE, COALESCE, or NULLIF expression returns a null literal, it has INTEGER data type. All other nulls are untyped.

- Null is a relatively common European family name, so its use as a literal default for name columns can produce incorrect query results.
- With the exception of CASE, COALESCE, and NULLIF, nulls are not valid predicate conditions in SQL expressions.

  For example, the following CASE expression is valid:

  ```
  SELECT CASE NULL + 1;
  ```

  but the following predicate condition is not valid:

  ```
  SELECT EmpNo
  FROM Employee
  WHERE EmpNo = NULL + 1;
  ```

Bivalent and Higher-Valued Logics

Introduction

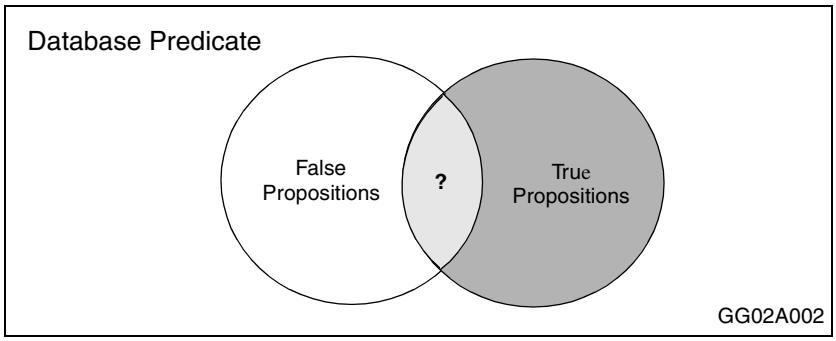
This topic presents a brief overview of the relevant predicate logic underlying the database relational model and the place of nulls in that logic.

First Order Predicate Logic Is Based on Bivalent Logic

In his original statement of the database relational model, Codd (1970) explicitly stated that it was based on the foundation of first order predicate logic. The classic first order predicate logic is based on bivalent logic[3], and though the theory of higher-valued logics has been studied by mathematicians and philosophers, no true multivalued logics are used for real world applications. Upon close examination, it is apparent that the logic supporting SQL is actually a bivalent logic (2VL) with additional ad hoc support for nulls.[4] Indeed, Codd did not add nulls to the relational model until 1979 (see section 2.3 of Codd, 1979).

The Closed World Assumption Revisited

Recall that any tuple in a relation body is assumed to conform to the Closed World Assumption (see "The Closed World Assumption" on page 12-8). With the addition of nulls to the database relational model, the CWA can no longer be assumed to be true. As an example, consider the following Venn diagram.



3 A bivalent logic has exactly two truth values: TRUE and FALSE. Higher-valued logics have three or more truth values. For example, SQL uses a 3VL having the following three truth values: TRUE, FALSE, and UNKNOWN.

4 Note that the inconsistent treatment of missing values in SQL is common to all vendors: it is not an implementation problem per se, but rather a problem with the way SQL itself is defined.

In contrast with the Venn diagram on page 12-8, there is overlap between the set of false propositions and the set of true propositions, represented by the darker shading. Better put, there is potential overlap between those sets. For example, suppose the Employee table has an EmpAge column that permits nulls. The company hires an employee who does not disclose her age in the application form. All other information for the employee is entered. Because the age for this employee is missing, the tuple that represents her can neither be evaluated as true nor as false, so it falls into the ambiguous intersection of the Venn diagram.

At some point, it is discovered that this employee is 15 years old, which violates a constraint on the EmpAge column that requires all employees to be at least 18 years old. The proposition for this employee now evaluates as false, and the tuple that represents her is no longer valid in the database. Nevertheless, that tuple was treated as valid from the time its data was entered until the time the age of the employee was discovered to violate the constraint specified for the EmpAge column, and that information was used to produce any number of reports which are, in retrospect, known to be non-valid.

How Many Logical Operators Must Be Supported?

In a bivalent logic, there are exactly four possible monadic, or single-operand, logical operators, as indicated in the following table. Note that the numberings have no meaning other than to distinguish the operators from one another.

This operator number ...	Performs these mappings of TRUE and FALSE values ...
1	T → T F → T
2	T → F F → F
3	T → F F → T
4	T → T F → F

By generalization, it is also true that there are exactly 16 possible dyadic, or two-operand, logical operators.

The formulas that produce the numbers of monadic and dyadic operators are provided as follows, where *n* represents the valence of the logic in question. For a bivalent logic, its value is 2; for a trivalent logic, its value is 3.

$$\text{Number of monadic operators} = n^n$$

$$\text{Number of dyadic operators} = n^{n^2}$$

By performing the calculations for a trivalent logic, you find that 3VLs have 27 possible monadic operators and 19 683 possible dyadic operators. These figures, of course, are redundant: they represent the number of possible operators, not the number of useful operators.[5] Nevertheless, they indicate that SQL does not support anywhere near the number of logical operators required to support a coherent 3VL. Imagine the difficulty of trying to support the 4VL that Codd (1990) advocates, for which there is a possible 256 monadic operators and over four billion dyadic operators (4 294 967 296 to be exact).

SQL Does Not Provide Adequate Support for a Consistent Trivalent Logic

The message to be taken away from this is that the treatment of nulls by SQL is not only inconsistent, but sometimes even incorrect from a real world perspective. The purpose of this description of the problems inherent in SQL missing values is to help you to understand what the most important inconsistencies are and how you can avoid some of them through careful database design.

---

5 SQL, like most programming languages, overdetermines its operators. For example, the function performed by the = operator in the following two equations is not identical internally, but the language uses the same symbol for the operation in both cases.

- `'value' = 'value'`
- `2 = 2`

The issue raised here is whether SQL supports a sufficient number of trivalent logical operators to fully support a 3VL, and the answer to that question is negative.

Perspectives on the Advantages and Disadvantages of SQL Nulls

Arguments in favor of using nulls to indicate missing information in relational databases are invariably formulated in terms of practicality: information is sometimes missing in the real world and database management systems ought to reflect that reality (see, for example, Fesperman, 1998 and Johnston, 1995abc).

Arguments against the use of nulls to indicate missing information are just as invariably framed in terms of mathematical rigor: the logical systems upon which the relational model is based do not support multivalent logics and the ad hoc support of nulls used to indicate missing information is inherently problematic. Unfortunately, some of the counterproposals to using nulls are themselves nearly as problematic.

This section briefly outlines the various arguments advanced in support of both positions, noting the problems with each, and arguing some reasonably conservative recommendations on how to avoid many of the problems that nulls present to the unwary user.

The case for nulls is given in "Arguments in Favor of Using Nulls to Record Missing Information" on page 13-9, while the cases for alternative methods of dealing with missing information are given in "Alternative Proposals for Dealing With Missing Information in Relational Databases" on page 13-11.

Arguments in Favor of Using Nulls to Record Missing Information

Introduction

This section describes the most fundamentally sound and most commonly expressed reasons given to justify the use of nulls to record missing information in SQL databases.

The Position of E.F. Codd on Nulls

Codd (1990) devotes two entire chapters and parts of several others to the missing information problem. Significantly, he writes, "I do not feel this part of the relational model rests on such a solid theoretical foundation as the other parts" while at the same time contending that "…criticisms of SQL's treatment of missing information do not justify abandoning database nulls…"

Perhaps the most important justification Codd invokes for using nulls to record missing information is that the problem must be addressed in such a way that the treatment is uniform and systematic across the entire database. None of the arguments against nulls in SQL databases provide the systematic maintenance for recording missing information that 3VL and nulls do. On the other hand, the uniformity of null manipulation and maintenance is largely missing from SQL, as was detailed in "Summary of Inconsistencies in How SQL Treats Nulls" on page 13-3, and this weakness is the largest chink in its armor. Codd (1986) also readily admits to a significant problem with tautologies[6] in 3VL: they sometimes evaluate to unknown even though, by definition, they must be true.

Codd on Nulls and the Breakdown of Normalization

Critics of nulls sometimes argue that their interpretation presents significant problems for dependency theory. For example, consider a base table T with the attributes X, Y, and Z. In shorthand notation, this can be written T(X,Y,Z). This table satisfies the functional dependence X→Y, for which no assumption is made that X is the primary key of T.

Suppose T has a row (?,y1,z1), where ? represents null. Now suppose that you attempt to insert a row (?,y2,z2) into T, where y1≠y2. The argument is made that either the nulls must be considered to be distinct or the row you are attempting to insert must be rejected because it might violate the stated dependency when its null is replaced by a value.

6 In logic, a tautology is a formula that is necessarily always true and therefore evaluates as true at every line in a truth table.

Codd counters this argument by stating a third option: that the dependency should not be enforced by the database until an attempt is made to replace a null with a value. This position seems to want to suspend the relational model for all such rows until such time as they replace nulls with values. A more positive way of stating this position is to say that the rules of functional and multivalued dependencies do not apply in the presence of nulls. Codd does not give a theoretical justification for this position because none can be provided.

Alternative Proposals for Dealing With Missing Information in Relational Databases

Introduction

Unlike every other feature of the database relational model, the treatment of missing information is not based on a formal foundation. As a result, even though the usual explanations of nulls often appear to make common sense (as Maier, 1983, disapprovingly writes, “It all makes sense if you squint a little and don’t think too hard”), any detailed investigation reveals a number of potentially serious problems with SQL nulls.

This section describes some proposed alternative solutions to the missing value problem.

Systematic Use of Default Values

Date (1992d) has argued for a scheme using 2VL with the systematic application of default values in place of nulls that he believes avoids the problems of 3VL and nulls. There are a number of difficulties with this solution, which even Date calls “quite ugly,” but some find it to be an improvement over the use of nulls with SQL.

The basic concept is to use real values that are outside the defined business rules (see “Business Rules” on page 12-11) for each column in the database. For example, if the valid range for employee numbers in the Employee table is 1 - 99999, then you might specify a default employee number of 0. As a result, any time a row that did not specify a value for employee number was inserted into the Employee table, its value would be 0 by default. Because SQL has a built-in, nonprocedural, method to handle default values in exactly this way (see the default value control phrases “DEFAULT” and “WITH DEFAULT” in the chapter “Default Value Control Phrases” in *Teradata RDBMS SQL Reference, Volume 3* for further information), no special coding is required on the part of users to implement this feature.[7]

Unfortunately, SQL has no built-in functions or operators to handle this default values scheme. Date proposes several extensions to SQL to deal with the issue, but these extensions are not in ANSI SQL, so the burden of ensuring that default marker values do not enter into query results rests with the user. All queries against columns containing default values must be filtered by means of

7 Note that if no defaults are specified for a column and it is not defined as NOT NULL, then SQL automatically inserts nulls to represent any information missing from an INSERT statement.

WHERE clauses that exclude default values from results. For example, if you wanted to know how many employee numbers were currently in use, the query would have to look like this:

```
SELECT COUNT DISTINCT EmpNo
FROM Employee
WHERE EmpNo <> 0;
```

This method of dealing with the issue becomes very cumbersome as the number of projected columns in a query that might contain missing information increases, and a failure to cover all the necessary columns in the WHERE clause predicate ensures that query results are not correct.

It is this liability that forms the core of the opposition to the use of default values. As Codd (1990) writes, "the D[efault]V[alue] approach does not appear to provide any tools for the *handling* of missing information, but merely provides a means for *representing* the fact that something is missing" (emphasis in original). Date (1992) counters with equal effectiveness that the burdens placed on users by nulls are no less problematic.

Missing Information as Metadata

McGoveran (1993, 1994abc) makes a case for dealing with missing information by means of metadata. Using an audit trail analogy, McGoveran argues that each table should have a corresponding dictionary table that records markers for different types of missing information transparently. Because SQL does not provide the facilities necessary to support this scheme, its practicality is currently moot. The interested reader is referred to the cited references for additional information.

Redesigning the Database to Eliminate the Need to Record Nulls

Pascal (2000) bases his argument against using nulls on some fundamental properties of the relational model. In the first order predicate calculus, propositions can have one of two values: they are either true or they are false. Recall that the rows of a relational table correspond to true logical propositions (see "An Illustration of How Databases Are Built From Logical Propositions" on page 12-9), because all false propositions are excluded from the database by means of various integrity constraints.

One can build a set of axioms deriving from such a set of true propositions in addition to a set of well-defined inference rules (referred to as a calculus in formal logic). Additional true propositions, formally known as theorems, can be derived from this calculus. These derived propositions correspond to valid relational queries if and only if the following statements about the original propositions are true:

- The initial set of rows represents only true propositions.
- All operations on a relation that contains tuples corresponding to these true propositions obey the formal inference rules of bivalent logic.

If either of these statements is false, then the correctness (truth of the derived propositions) of any query made against the data cannot be guaranteed. Because SQL nulls represent data values that cannot be asserted, it is not possible to know whether rows that contain nulls represent true propositions or not, hence, the tables that contain them are not true relations in the mathematical sense.

Noting that one of the most frequently provided justifications null supporters give for their use is that maintaining rows with missing values is often a necessary and useful thing to do in the real world. Why, Pascal asks, is it useful to record what he calls "partial nothings" in the database if it is obviously nonsensical to maintain "full nothings"? Pascal then uses the following extreme example to support his case against nulls.

Consider the following classic case study: an employee table that records the salary of each employee currently working for the enterprise. It is not uncommon for salaries to be unknown at a given point, so an employee row might contain a null to hold the place of the true salary of that employee. At the time the row is inserted into the table, the salary value is missing information. At a given instant, this employee table might look like this:

Employee

EmpNum	EmpName	DeptNum	HireDate	Salary
PK				
214	Smith	32	09-12-1989	56150
447	Lau	15	05-30-1993	?
103	Hossein	09	09-13-1984	29775
500	Nakamura	11	06-09-1997	84932
713	Schroeder	24	10-29-2001	?

New hire Schroeder has not yet been assigned a salary and Lau has just been promoted, but her new salary has not yet been determined.

Pascal argues that in reality, this "partial nothing" information is no more useful than the following "partial nothing" table, for which only the primary key values known for each row.

Employee

EmpNum	EmpName	DeptNum	HireDate	Salary
PK				
214	?	?	?	?
447	?	?	?	?
103	?	?	?	?
500	?	?	?	?
713	?	?	?	?

The problem with the "partial nothing" table is that its semantics are perceived as something other than what they are. The intended meaning of the nulls in the Salary column derives from the true proposition that all employees receive a salary. The formal semantics of the table, however, assert only that a salary amount exists for each employee. As a result, the table actually represents a mix of two different categories of assertion:

- Those rows with nulls that apply to all employees ("a salary amount exists for all employees").
- Those rows that describe only those employees whose salaries are known ("all employees receive a salary").

The particular problem with nulls in this table is readily resolved with a minor change in the logical design of the database. Because the attributes in this Employee table are a combination of those that apply to all tuples and those that apply only to some tuples, the relationship is similar to that of a simple entity supertype and subtype, the difference being that the attribute in question, Salary, is not unique: all employees receive a salary.

The following tables resolve the problem of multiple simultaneous semantics for the original Employee table:

Employee

EmpNum	EmpName	DeptNum	Hiredate
PK			
214	Smith	32	09-12-1989
447	Lau	15	05-30-1993
103	Hossein	09	09-13-1984
500	Nakamura	11	06-09-1997
713	Schroeder	24	10-29-2001

EmployeeSalary

EmpNum	Salary
PK	
214	56150
103	29775
500	84932

Using Probability Measures to Capture Attribute Value Uncertainty

One promising approach to dealing with uncertain information, the unknown or missing value type of null, is to devise a consistent method for estimating the value of a missing attribute based on the joint probability of all the attribute values in a row (see Cavallo and Pittarelli, 1987, for an introduction). Recent work in the field has progressed far enough to develop a set of consistent normal forms analogous to 1NF, 2NF, 3NF, BCNF, and 4NF for probabilistic databases (Dey and Sarkar, 2002) and methods for dealing with aggregation (McClean et al., 2001). Though this work is promising, it is still far too preliminary to be considered for commercial database management.

Manipulating Nulls With SQL

Introduction

As described in "Types of Missing Values" on page 13-2, SQL nulls are often mistakenly interpreted in ways that extend significantly beyond their single valid meaning, which is simply "unknown" (see footnote 2 on page 13-2 for a counterexample).

These properties make the use and interpretation of nulls in SQL problematic. The following sections outline the behavior of nulls for various SQL operations to help you to understand how to use them in data manipulation statements and to interpret the results those statements effect.

This section introduces some of the issues involved with manipulating nulls in relational databases. The topics are described in greater detail in following sections of this chapter.

Nulls Are Place Holders for Missing Information, Not Values

Inexperienced SQL programmers often think of nulls as if they represent values, like the symbolic variables x and y in the algebraic equation $y = 2x + 3$. This reasoning is false because nulls are neither values nor are they representations of values as variables are; instead, nulls represent the *absence* of value. A null is nothing more than a place holder indicating that no value is present: the value for the field is unknown.

Logical and Arithmetic Operations on Nulls

Nulls are not valid as predicate conditions other than for the unique example of CASE expressions (see "Nulls and CASE Expressions" on page 13-20).

You can*not* solve for the value of a null because, by definition, it *has* no value. For example, the expression `value = NULL` has no meaning and therefore can never be true (however, see "Null Sorts as the Lowest Value in a Collation" on page 13-22 for a counterexample). A query that specifies the predicate `WHERE value = NULL` is not valid because it can never be true *or* false. The meaning of the comparison it specifies is not only unknown, but unknowable.

If you want to search for fields that do or do not contain nulls, you must use the operators IS NULL or IS NOT NULL (see "Searching for Nulls Using a SELECT Statement" on page 13-22, "Excluding Nulls From Query Results" on page 13-22 and "Searching for Nulls and Non-Nulls In the Same Search Condition" on page 13-23).

The difference is that when you use a mathematical operator like =, you specify a comparison between values or value expressions, whereas when you use the IS NULL or IS NOT NULL operators, you specify an existence condition. Note that even though IS NULL and IS NOT NULL return truth values, their operands are *not* truth values; therefore they are not logical operators and do not count against the number of possible logical operators for a trivalent logic calculated in "How Many Logical Operators Must Be Supported?" on page 13-6.

Null Literals

NULL can be used as literals in some circumstances.

Hashing on Nulls

Even though nulls have no external value, they do have an internal value that can be processed by the Teradata hashing algorithm. If the primary index for a table permits nulls, there is an increased probability of an uneven distribution of the rows for that table across the AMPs of a system because all rows having a null primary index hash to the same AMP (see "Maximum Number of Rows Null" on page 9-25). If the number of rows having a null primary index is sufficiently large, then significant skew occurs, making efficient parallel processing difficult to achieve.

Further Information

See "How SQL Operators and Functions Handle Nulls" on page 13-18, "Miscellaneous Topics on Manipulating Nulls With SQL" on page 13-22, and "NULL Literals" on page 13-24 for details about manipulating nulls with SQL.

How SQL Operators and Functions Handle Nulls

Introduction

This topic presents the behavior of SQL operators and functions with respect to null handling. These properties are not always intuitive, and it is critical that you understand them if you are to interpret the results of query manipulations on data containing nulls correctly.

Nulls and Arithmetic Operators and Functions

If an operand of any arithmetic operator or function is null, then the result of the operation or function is usually null. The following table provides some illustrations and some exceptions:

WHEN the expression is . . .	THEN the result is . . .
5 + NULL	null
LOG(NULL)	null
NULLIFZERO(NULL)	null
ZEROIFNULL(NULL)	0
COALESCE(NULL, 6)	6

Nulls and Comparison Operators

If either operand of a comparison operator is null, then the result is unknown and an error is returned to the requestor. The following examples indicate this behavior.

WHEN the expression is . . .	THEN the result is . . .	AND this error message returns to the requestor ...
5 = NULL	Unknown	3731
5 <> NULL		
NULL = NULL		
NULL <> NULL		
5 = NULL + 5		

Note that if the argument of the NOT operator is unknown, the result is also unknown. This evaluation translates to FALSE as a final boolean result.

Nulls and Aggregate Functions

With the important exception of COUNT(*), aggregate functions ignore nulls in their arguments. This treatment of nulls is very different from the way arithmetic operators and functions treat them, and is one of the major inconsistencies in the way SQL deals with nulls.

This behavior can result in apparent nontransitive anomalies. For example, if there are nulls in either column A or column B (or both), then the following expression is virtually always true.

```
SUM(A) + (SUM B) <> SUM (A+B)
```

In other words, for the case of SUM, the result is never a simple iterated addition if there are nulls in the data being summed.

The only exception to this is the case in which the values for columns A and B are *both* null in the same rows, because in those cases the entire row is not counted in the aggregation. This is a trivial case that does not violate the general rule.

The same is true, the necessary changes being made, for all the aggregate functions except COUNT(*), which does include nulls in its result.

If this property of nulls presents a problem, you can perform either of the following workarounds, each of which produces the desired result of the aggregate computation `SUM(A) + SUM(B) = SUM(A+B)`.

- Define all NUMERIC columns as `NOT NULL DEFAULT 0`.
- Use the ZEROIFNULL function nested within the aggregate function to convert any nulls to zeros for the computation, for example

  ```
  SUM(ZEROIFNULL(x) + ZEROIFNULL(y))
  ```

Nulls and DateTime and Interval Data

The general rule for managing nulls with DateTime and Interval data environments is that, for individual definitions, the rules are identical to those for handling numeric and character string values.

WHEN any component of this type of expression is null . . .	THEN the result is . . .
Value	null.
Conditional	FALSE.
CASE	as it would be for any other CASE expression.

DateTime or Interval values are defined to be either atomically null or atomically non-null. For example, you cannot have an interval YEAR TO MONTH value in which YEAR is null and MONTH is not.

Nulls and CASE Expressions

The ANSI SQL-99 definitions for the CASE expression and its related expressions NULLIF and COALESCE specify that these expressions can return a null. Because of this, their behavior is an exception to the rules for all other predicates and expressions (see "Introduction" on page 13-16).

The rules for null usage in CASE, NULLIF, and COALESCE expressions are as follows.

- If no ELSE clause is specified in a CASE expression and no WHEN clause evaluates to TRUE, then NULL is returned by default.
- Nulls and expressions containing nulls are valid as CASE conditions. The following examples are valid.

```
SELECT CASE NULL
        WHEN 10
        THEN 'TEN'
       END;

SELECT CASE NULL + 1
        WHEN 10
        THEN 'TEN'
       END;
```

- Nulls and expressions containing nulls are not valid as WHEN clause conditions. The following examples are not legal.

```
SELECT CASE column_1
        WHEN NULL
        THEN 'NULL'
       END
FROM table_1;
SELECT CASE column_1
        WHEN NULL + 1
        THEN 'NULL'
       END
FROM table_1;

SELECT CASE
        WHEN column_1 = NULL
        THEN 'NULL'
       END
FROM table_1;

SELECT CASE
        WHEN column_1 = NULL + 1
        THEN 'NULL'
       END
FROM table_1;
```

The following example is valid.

```
SELECT CASE
        WHEN column_1 IS NULL
        THEN 'NULL'
       END
FROM table_1;
```

- Nulls and expressions containing nulls are valid as THEN clause conditions. The following example is valid.

```
SELECT CASE
        WHEN column_1 = 10
        THEN 'NULL'
       END
FROM table_1;
```

NULLIF and COALESCE

The behavior of the CASE shorthand expressions NULLIF and COALESCE is the same as that for CASE with respect to nulls.

See “NULLIF” and “COALESCE” in the chapter “CASE Expressions” in *Teradata RDBMS SQL Reference, Volume 5* for further information.

Miscellaneous Topics on Manipulating Nulls With SQL

Introduction

This topic describes several minor issues you need to understand before you can manipulate data that contains nulls.

Null Sorts as the Lowest Value in a Collation

When rows are sorted using an ORDER BY clause, nulls sort as the lowest value.

If any row has a null in the column being grouped, then all rows having a null are placed into one group. In other words, though it is syntactically incorrect to formulate the predicate NULL = NULL in a DML statement because nulls have no value and therefore cannot be equated, it is also true that when SQL sorts nulls, the implicit result is that NULL = NULL.

Searching for Nulls Using a SELECT Statement

The IS NULL operator tests for the presence of nulls in a specified column. For example, to search for the names of all employees who have a null in the deptno column, you could type the statement:

```
SELECT name
FROM employee
WHERE deptno IS NULL ;
```

This query produces the names of all employees having a null in the deptno field. Because all employees contained in the employee table have been assigned to a department, no rows would be returned for this particular example.

Excluding Nulls From Query Results

The IS NOT NULL operator excludes rows having a null in a specified column from the results of a query.

For example, to search for the names of all employees with non-nulls in the jobtitle column, you could type the following statement:

```
SELECT name
FROM employee
WHERE jobtitle IS NOT NULL ;
```

The result of this query is the names of all employees with a non-null in the jobtitle column. For this particular example, the names of all employees are returned because every employee has, by definition, been assigned a job title.

Searching for Nulls and Non-Nulls In the Same Search Condition

To search for nulls and non-nulls within the same predicate, the search condition for nulls must be specified separately from any other search conditions.

For example, to select the names of all employees with the job title of Vice Pres, Manager, or null, you could type the following SELECT statement:

```
SELECT name, jobtitle
FROM employee
WHERE jobtitle IN ('Manager' OR 'Vice Pres')
OR jobtitle IS NULL ;
```

NULL Literals

Introduction

The keyword NULL is sometimes available as a special construct similar to, but not identical with, a literal. In this case, the keyword NULL represents the SQL null placeholder for a value.

Rules for Using NULL as a Literal

The literal NULL can be used in the following ways:

- As a CAST source operand, for example:
  ```
  CAST (NULL AS value)
  ```
- As a CASE result, for example:
  ```
  CASE expression
  THEN NULL
  ```
  or
  ```
  CASE expression
  ELSE NULL
  END
  ```
- As a referential specification, for example:
  ```
  SET NULL
  ```
- As an item specifying a null is to be placed in a column on INSERT or UPDATE.
- As a default column definition specification, for example:
  ```
  DEFAULT NULL
  ```
- As an explicit SELECT item, for example:
  ```
  SELECT NULL
  ```
  This usage is a Teradata extension to the ANSI SQL-99 standard.
- As an operand of a function, for example:
  ```
  SELECT TYPE(NULL)
  ```
  This usage is a Teradata extension to the ANSI SQL-99 standard.

Data Type of NULL Literals

When you use NULL as an explicit SELECT item or as the operand of a function, its data type is INTEGER. For example, if you perform SELECT TYPE(NULL), then the data type of NULL is returned as INTEGER.

In all other cases NULL has no data type because it has no value.

Nulls and the Outer Join

Introduction

The outer join is a join that retains rows from one or both of the joined tables, depending on whether it is declared to be a left, right, or full outer join, respectively. This means that rows that do not match the rows produced by the inner join are extended in the outer join result by reporting nulls in the nonmatching columns. From the perspective of set theory, these nonmatching column values are not missing values, but the empty set, which is a real value. Because SQL represents the empty set as a null, the semantics of outer joins must be evaluated carefully whenever there are actual missing values in the inner join portion of the operation.

Ambiguities presented by nulls in the outer join also make query optimization more difficult, because certain key methods of query optimization such as transitive closure often cannot be achieved.

Semantics of Nulls in the Outer Join

The motivation behind the outer join is to preserve information that is otherwise lost in an inner join. There can be no doubt that this is an important issue, but the SQL solution to the problem presents some inconsistencies regarding the semantics of the nulls reported by an outer join. The following contrast of simple inner and outer natural joins on the same two tables is based on an example developed by Date (1992c). The respective joins are between the following Supplier and Supplier-Parts tables:

Supplier

SupplNum	SupplName	Status	City
PK			
S2	Jones	10	Paris
S5	Adams	30	Athens

Supplier-Parts

SupplNum	PartNum	Quantity
PK		
S2	P1	300
S2	P2	400

Here is the result of the inner join of the Supplier and Supplier-Parts tables:

SupplNum	SupplName	Status	City	PartNum	Quantity
S2	Jones	10	Paris	P1	300
S2	Jones	10	Paris	P2	400

This join outcome has no information about supplier S5, therefore, it is said to have lost that information.

Here is the result of the natural outer join of the Supplier and Supplier-Parts tables:

SupplNum	SupplName	Status	City	PartNum	Quantity
S2	Jones	10	Paris	P1	300
S2	Jones	10	Paris	P2	400
S5	Adams	30	Athens	?	?

This join outcome retains information about supplier S5, therefore it is said to preserve that information. Note that the result reports nulls for the PartNum and Quantity columns for supplier S5.

Think back to the definition of null semantics that was presented in "Semantics of SQL Nulls" on page 13-2. Recall that the unequivocal definition for an SQL null is that it represents an unknown value. In this case, however, the values for both PartNum and Quantity for supplier S5 *are* known, and those values are both the empty set. Because SQL does not support empty sets directly, it substitutes nulls in their place. This is also true for the result of the length determination of an empty character string (see "Inconsistencies in the SQL Treatment of Nulls" on page 13-3).

The important thing to remember with respect to the semantics of outer join nulls is that nulls represent both the empty set (when used to extend the inner join to preserve information that would otherwise be lost) and semantically correct SQL nulls (when used to represent unknown information in the inner join portion of the result). The determination of which null is which is left to the user.

References

Cavallo, Roger and Michael Pittarelli, "The Theory of Probabilistic Databases," in: Peter M. Stocker, William Kent, Peter Hammersley (eds.): *13th International Conference on Very Large Data Bases*, pp. 71-81, 1987.

Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, 13(6):377-387, 1970.

Codd, E.F., "Extending the Database Relational Model to Capture More Meaning," *ACM Transactions on Database Systems*, 4(4):397-434.

Codd, E.F., "Missing Information (Applicable and Inapplicable) in Relational Databases," *ACM SIGMOD Record*, 15(4):53-78, 1986.

Codd, E.F., *The Relational Model for Database Management, Version 2*, Reading, MA: Addison-Wesley, 1990.

Codd, E.F. and C.J. Date, "Much Ado About Nothing," *Database Programming and Design*, October, 1993.

Date, C.J., "NOT Is Not 'Not'!" in: C.J. Date, *Relational Database: Writings 1985-1989*, Reading, MA: Addison-Wesley, 1990a.

Date, C.J., "EXISTS Is Not 'Exists'!" in: C.J. Date, *Relational Database: Writings 1985-1989*, Reading, MA: Addison-Wesley, 1990b.

Date, C.J., "Three-Valued Logic in the Real World," in: C.J. Date and Hugh Darwen, *Relational Database: Writings 1989-1991*, Reading, MA: Addison-Wesley, 1992a.

Date, C.J., "Oh No Not Nulls Again," in: C.J. Date and Hugh Darwen, *Relational Database: Writings 1989-1991*, Reading, MA: Addison-Wesley, 1992b.

Date, C.J., "Watch Out for Outer Join," in: C.J. Date and Hugh Darwen, *Relational Database: Writings 1989-1991*, Reading, MA: Addison-Wesley, 1992c.

Date, C.J., "The Default Values Approach to Missing Information," in: C.J. Date and Hugh Darwen, *Relational Database: Writings 1989-1991*, Reading, MA: Addison-Wesley, 1992d.

Date, C.J., "A Note on the Logical Operators of SQL," in: C.J. Date, *Relational Database: Writings 1991-1994*, Reading, MA: Addison-Wesley, 1995.

Date, C.J., *An Introduction to Database Systems* (7th ed.), Reading, MA: Addison-Wesley, 2000.

Date, C.J., Hugh Darwen, and David McGoveran, "Nothing to Do With the Case," *Database Programming and Design*, 8(9), 1995.

Date, C.J., Hugh Darwen, and David McGoveran, “Up to a Point, Lord Copper,” *Database Programming and Design*, http://www.dbpd.com/vault/dateresp.htm, 1997.

Dey, Debabrata and Sumit Sarkar, Generalized Normal Forms for Probabilistic Relational Data,” *IEEE Transactions on Knowledge and Data Engineering*, 14:485-497, 2002.

Fesperman, Lee, *Should Nulls Be Considered Harmful?* http://www.firstsql.com/inulls.txt, 1998.

Johnston, Tom, “MVL: Case Open,” *Database Programming and Design*, 8(2), 1995a.

Johnston, Tom, “The Case for MVL,” *Database Programming and Design*, 8(3), 1995b.

Johnston, Tom, "More to the Point," *Database Programming and Design*, 8(11), 1995c.

McClean, Sally, Bryan Scotney, and Mary Shapcott, “Aggregation of Imprecise and Uncertain Information in Databases,” *IEEE Transactions on Knowledge and Data Engineering*, 13:902-912.

McGoveran, David, “Much Ado About Nothing, Part 1: Nothing from Nothing (Or, What’s Logic Got to Do With It?),” *Database Programming and Design*, 6(13), 1993.

McGoveran, David, “Much Ado About Nothing, Part 2: Classical Logic: Nothing Compares 2 U,” *Database Programming and Design*, 7(1), 1994a.

McGoveran, David, “Much Ado About Nothing, Part 3: Can’t Lose What You Never Had,” *Database Programming and Design*, 7(2), 1994b.

McGoveran, David, “Much Ado About Nothing, Part 4: It’s in the Way That You Use It,” *Database Programming and Design*, 7(3), 1994c.

Maier, David, *The Theory of Relational Databases*, Rockville, MD: Computer Science Press, 1983. Note: This book is out of print, but is available on CD-ROM 1 in Volume 4 of the *ACM SIGMOD Anthology*, which is available from the Association for Computing Machinery, Inc.

Pascal, Fabian, *Practical Issues in Database Management: A Reference for the Thinking Practitioner, Chapter 10: What You Don’t Know Can Hurt You: Missing Information*, Reading, MA: Addison-Wesley, 2000.

Wu, Xintao and Daniel Barbará, “Learning Missing Values from Summary Constraints,” *ACM SIGKDD Explorations*, 4(1):21-30, 2002.

Section 5:

Physical Database Design, Part 3: Capacity Planning

Chapter 14:

Database-Level Considerations

Introduction

This chapter describes some of the database-level considerations you must evaluate when planning the storage capacity needs of your data warehouse.

Chapter 15: "System-Level Considerations," describes the more file system- and hardware-oriented considerations undertaken for capacity planning.

Capacity Planning Topics Described In This Chapter

The major capacity planning issues described in this chapter are the following:

- "Base Table Row Format" on page 14-4
- "Table Header Format" on page 14-7
- "Column Sizing Guidelines" on page 14-12
- "System-Derived Columns" on page 14-13
- "Value Compression" on page 14-20
- "Row Size Calculation" on page 14-84
- "Sizing Base Tables, Hash Indexes, and Join Indexes" on page 14-89
- "Sizing a Unique Secondary Index Subtable" on page 14-91
- "Sizing a Nonunique Secondary Index Subtable" on page 14-92
- "Sizing a Reference Index Subtable" on page 14-94
- "Sizing Spool Space" on page 14-96
- "Sizing a Query Capture Database" on page 14-99
- "How to Size Table Space Empirically" on page 14-101

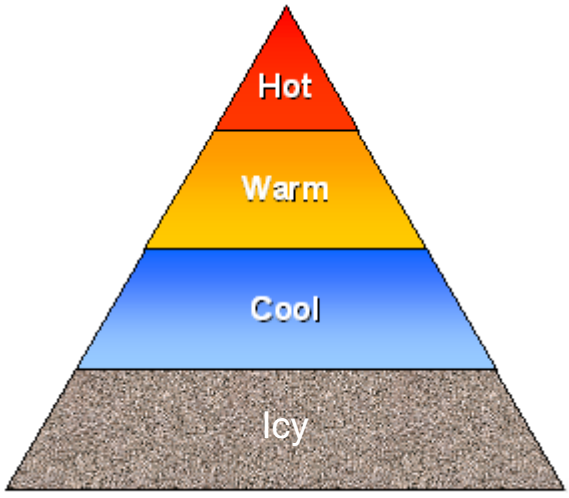
Capacity Planning and Legacy Databases

This chapter emphasizes the importance of capacity planning. When there is no legacy database to build on, capacity planning can be a difficult enterprise to undertake. Fortunately, this is rarely an issue for contemporary users because at least part of their corporate databases are almost always maintained in electronic form. Keep in mind that much of the information presented in this chapter assumes you have a legacy system to draw upon for making your sizing estimations.

Capacity Planning and Data Access Frequency

Capacity planning should begin with the idea of making the most frequently accessed data available at all times. With the relatively low priced, large capacity disk storage units commonly used for data warehousing applications, the nature of the emphasis on this factor has changed from offloading as much historical data as possible to archival storage toward developing the capability of keeping all data forever accessible to the warehouse.

Users often have varied performance requirements for time-based or historical data: recent data might be accessed frequently, while older data is accessed less often. Call these varied access rates hot, warm, cool, and icy ranking from most frequently to least frequently accessed. In a warehouse with massive historical databases, the volume of data is typically inversely proportional to the data usage, as illustrated by the following graphic.



1094A017

In this picture, the temperature of the data reflects its access rate. Typically, there is a vast quantity of cool historical data in the data warehouse that is accessed lightly and a lesser volume of hot and warm data that is accessed frequently. The optimal storage medium for hot, warm, and cool data is online disk that is directly accessible to the data warehouse. Icy, or truly dormant, data is a candidate for alternative storage such as tape or optical disk. The treatment of icy data is not the subject of this chapter. Hot, warm, and cool data are all relevant, particularly with respect to compression (see "Value Compression" on page 14-20).

Note the distinction between cool and icy data. While cool data might be accessed lightly on average, it still has temporal hot spots, such as those that occur when performing comparative analyses of sales data between different time periods. Similarly, if the relational schema is modified by adding new columns or indexes, or if column data types are changed, then the affected data cannot be considered to be dormant. Finally, if data is periodically accessed and somehow recast to make historical data relevant within the current business context, then it is not dormant.

Truly dormant data is rarely, if ever, accessed, and is typically retained for legal rather than operational reasons.

Capacity planning is integral to the optimum performance of diversely accessed data. While numerous critical considerations for capacity planning are developed in this chapter, the principal emphasis is on compression.

Base Table Row Format

Introduction

This topic reviews the structure of Teradata base table rows. The structure is slightly different for rows in a table having a nonpartitioned primary index (NPPI) and rows in a table having a partitioned primary index (PPI).

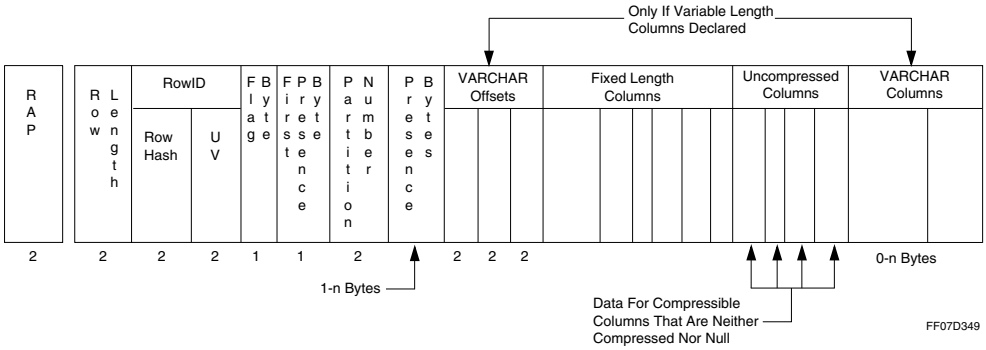
Row format information is used to make fine-grained row size estimates for capacity planning.

Row Structure

The following graphics describe the structure of a Teradata data row. 16 bytes are devoted to overhead. Because the maximum row length is 64 KB (65 536 bytes), the maximum number of data bytes that can be contained in a row is 65 520.

The number of characters that can be represented by this number of bytes varies depending on whether characters are represented by one byte, two bytes, or a combination of single-byte and multibyte representations.

The following graphic illustrates the basic structure of a Teradata row:



where:

This abbreviation …	Has this meaning …
RAP	Reference array pointer
UV	Uniqueness value

Row Length Characteristics

The following table describes the boundary characteristics for a Teradata row. The value for minimum length derives from the mandatory 16 overhead bytes per row.

Characteristic	Description
Maximum length	64 KBytes
Minimum length	16 bytes

Row Components

The following table describes the individual components of a Teradata row:

Component	Number of Bytes	Description
Reference array pointer	2	The reference array pointer is not part of the physical Teradata row. It is a two-byte entry maintained at the bottom of each data block that points to the first byte of each row in the block.
Row length	2	Specifies the exact length of the row in bytes.
RowID	10	Contains three component fields: • Row hash • Uniqueness value • Partition number
• Row hash	4	Contains the hashing algorithm-generated row hash value for the row.
• Uniqueness value	4	Contains the system-generated uniqueness value for the row.
• Partition number	2	Depending on the type of primary index for the table, contains the partitioning expression-generated number of the partition in which the row is stored.

IF the row is from a table with this kind of primary index …	THEN the partition number field …
NPPI	is set to 0.
PPI	contains a partition number for the row.

<table>
<tr><th>Component</th><th>Number of Bytes</th><th>Description</th></tr>
<tr><td>Flag byte</td><td>1</td><td>Defines whether the 2-byte Partition number field is used or not.
<table><tr><th>IF the flag is ...</th><th>THEN the row is ...</th></tr><tr><td>set</td><td>from a PPI table and the row header has an additional 2-byte partition number field following the first presence octet that stores its partition number.</td></tr><tr><td>not set</td><td>not from a PPI table and the row header does not use the partition number field, which is set to 0.</td></tr></table></td></tr>
<tr><td>First presence octet[a]</td><td>1</td><td>Defines column nullability and compressibility.</td></tr>
<tr><td>Partition number</td><td>2</td><td>Contains the partition number for a row in a PPI table.
If the row does not belong to a PPI table, then this field contains a 0.</td></tr>
<tr><td>Additional presence octets</td><td>8 bytes per presence octet</td><td>Provides additional space to define nullability and compressibility for columns if the first presence octet is too small to contain all the information for the table.
For more information, see “Presence Bits” on page 14-37.</td></tr>
<tr><td>Column offsets</td><td>Varies
2 bytes per variable length column</td><td>Only present when variable length columns are defined for a table.
Indicates the intrarow location of the column it references.</td></tr>
<tr><td>Fixed length columns</td><td>Varies</td><td>Contains all noncompressible fixed length columns.</td></tr>
<tr><td>Uncompressed columns</td><td>Varies</td><td>Contains columns defined as COMPRESS, but have neither been compressed nor nullable.</td></tr>
<tr><td>VARCHAR columns</td><td>Varies</td><td>Only present when variable length columns are defined for a table.
Contains all variable length character columns. The column offsets field points to the starting location for each variable length column in the row.</td></tr>
</table>

a. See “Presence Bits” on page 14-37 for more information.

Table Header Format

Introduction

Each Teradata table has an associated subtable called a table header. One copy of each table header is stored on each AMP in the system, having one row per table per AMP. Table headers are composed of a row header, one fixed-length field, and five variable-length fields (see "Table Header Components" on page 14-7).

Teradata uses table headers internally to maintain various information about each table. Table headers are described here primarily because they are used to store compressed column values (see "Value Compression" on page 14-20), and it is difficult to explain how compression works without having at least a superficial understanding of this file structure.

The size limit for a table header row is roughly 64 000 bytes, so it is not possible to create a table with 2 048 columns, 32 secondary indexes with 64 columns each, and so on, without exceeding that byte limit. The maxima stated in Appendix B: "Teradata Database Maxima" are for the individual parameters only, not their cumulative sum.

Table Header Components

The following table describes the individual fields of a Teradata table header.

Component	Description
Row header	See "Base Table Row Format" on page 14-4 for a description of Teradata row headers.
Field 1	• Row 1 header. • Table header row format version. • Database and table names in the format `databasename.tablename`. • Internal ID of the database to which this table belongs. • Internal ID of the database to which space for this table is charged. • Table creation timestamp. • Last table update timestamp. • Last table archive timestamp. • Table structure version. Updated each time the table description is modified. • Table structure version for host utilities. Incremented each time a structural change is made that makes a database dump obsolete. • Number of backup tables associated with this table. • Internal ID of the permanent journal table.

Component	Description
Field 1 (continued)	• Table kind. Describes whether the table is permanent, temporary, volatile, or a join index. For temporary tables, volatile tables, and join indexes, also describes preserve-on-commit and transient journaling characteristics. • Journal type. There are 5 types: – After – Audit – Both – Before – None • Protection type. There are 3 types: – Fallback – Log – None • User journal flag. Describes whether the table is a user-defined journal or not. • Hash flag. Describes whether table is hashed or not. • Dropped flag. Set TRUE from the time the AMP receives a Drop Table step until the table is dropped in the End Transaction step. Otherwise set FALSE. • DDL change flag. Prevents attempts to update the table after its DDL has changed. Set TRUE when table DDL is modified. Host utility table dump sets the flag FALSE. • Byte count of the number of USIs defined on the table. • Host character set at the time the table was created. • Number of parent tables referenced by this table. • Number of child tables referencing this table. • Data block size for the table in bytes. • Percent free space for the table. • Data block size validity. Indicates whether the specified data block size for the table is valid or unspecified. • Percent free space validity. Indicates whether the specified percent free space for the table is valid or unspecified.

Component	Description
Field 1 (continued)	• Message class of primary step. • Message kind of primary step. • Message class of secondary step. • Message kind of secondary step. • Host ID. • Session number. • Request number. • Transaction number. • Table ID of base temporary table. Used only for materialized temporary tables. • Internal ID of replicated error table. • Replicated table initiation flag. Indicates whether replicated table is initiated or not. • Internal ID for primary key index. • Restart flag. Tracks restart and non-restart cases of restore jobs during build phase. • Row format. Specifies whether current environment is 32-bit or 64-bit. • Dummy space. • List of index descriptors for the table in Index ID order, one per primary and secondary index defined on the table. • Row 1 length. Duplicate specification of the length of row 1. Fixed length.
Field 2	Contains the primary index descriptor and all secondary index descriptors. Variable length. Uses Field 1 offset from row header to locate the field.
Field 3	Not used.
Field 4	Contains MultiLoad, FastLoad, and Archive/Recovery information. Variable length. Uses Field 2 offset from row header to locate the field.

Component	Description
Field 5	Contains the following table field descriptors: • Internal ID of the first field in the table. • Number of varying length fields in the table. • Number of presence bits in each row.[a] • Compression flag. Specifies whether the table has compressible fields or not. • Offset in the row to the presence bit array.[b] • Offset in the row to the first byte past the presence bit array. • Number of fields (columns) in the table. • Index into the field descriptor array to the first compressible field. If no fields are compressed, then the value points to the first varying length field in the row. • Offset from the beginning of the row to the first optional (varying or compressible) data. • Index of the field descriptor for the first physical field in the row. • Field 5 type. There are 4 types: – Table descriptor with row hash and unique rowID. – Index descriptor with row hash and unique rowID in the index. – Table descriptor with a PPI rowID. – Index descriptor with a PPI rowID in the index. • Row format. Specifies whether current environment is 32-bit or 64-bit. • Duplicate rows flag. There are 3 types: – Dictionary and non-ANSI tables. – ANSI tables without unique indexes. – ANSI tables with unique indexes. • Offset to system code to build rows and to calculate PPI internal partition numbers. • Field descriptors array. • System code for building rows, including code to calculate partition numbers for the rows of PPI tables.[c] • Compressed values.[d] Variable length. Uses Field 3 offset from row header to locate the field.

Component	Description
Field 6	Contains restartable sort and ALTER TABLE information. Variable length. Usually null. Uses Field 4 offset from row header to locate the field.
Field 7	Contains reference index descriptors. See "Sizing a Reference Index Subtable" on page 14-94 for a description of reference indexes. Variable length.

a. The upper limit for the number of presence and compress bits per row is 89 991.
b. Presence octet locations are determined by dividing the presence bit position by 8.
c. The code is not strictly stored within Field 5, but is stored in the table header row just beyond the Field 5 descriptor and just before the compressed values, if any.
d. Compressed values are not strictly stored within Field 5, but are stored in the table header row just beyond the Field 5 descriptor. The stored values are sorted first in ascending order of their binary values and then in descending order of their field size.

Column Sizing Guidelines

Introduction

This topic summarizes column sizing facts and guidelines. You should carefully read the material on value compression beginning with "Value Compression" on page 14-20 and ending with "Calculating the Efficiency of Compression" on page 14-50 to understand how to implement optimally most of the guidelines described in this topic.

See "System-Derived Columns" on page 14-13 for additional important column sizing information.

Compression Guideline

Take care to compress column values only when compression does not have the net effect of reducing storage capacity. See "Calculating the Efficiency of Compression" on page 14-50 and "Computing and Interpreting the Break-Even Point for Compression" on page 14-44 for information about how to make this determination.

Compression produces shorter rows, which generally produce more performant applications.

Miscellaneous Column Sizing Considerations and Guidelines

The following considerations and guideline are also important for sizing columns:

- Adding a column expands the capacity required for every row in a table if the column is not compressible.
- Dropping a column reduces every row in a table when there was uncompressed data in the dropped column.
- Specify VARCHAR for any character column for which the space savings offsets the two bytes per row overhead and if compression on a fixed length CHARACTER field would save less than 5 percent additional capacity.

System-Derived Columns

Introduction

As the name implies, system-derived columns are columns whose values are not created by users, but instead are generated by Teradata. Some system-derived columns contain values that Teradata creates and maintains for internal use, while others contain values that are critical for user applications.

In some cases, the DBA specifies the name of the column and whether or not it is to be created for a table and in other cases, the existence, name, and contents of the column are all system-controlled.

The following system-derived column types are supported by Teradata:

- Identity
- ROWID
- PARTITION

Identity columns are user-defined, but the system defines the values inserted into them[1].

ROWID and PARTITION columns are always system-defined and their values are always system-generated.

Identity Columns

To create a table with an identity column, you must explicitly specify one column in the table definition as either GENERATED ALWAYS AS IDENTITY or GENERATED BY DEFAULT AS IDENTITY. You can only define one identity column per table.

The data type for an identity column is user-defined, but must be an exact numeric type drawn from the following list:

- BYTEINT
- DECIMAL(n,0)
- INTEGER
- NUMERIC(n,0)
- SMALLINT

1 In the case of GENERATED ALWAYS AS IDENTITY columns, all column values are system-generated.
In the case of GENERATED BY DEFAULT AS IDENTITY columns, inserted column values can be user-generated or system-generated.

In general, you should define the column to be either DECIMAL or NUMERIC, with the value for *n* being the largest number available for your system (either 15 or 18, depending on your setting for the DBSControl record MaxDecimal flag.

Identity columns have many applications, including the automatic generation of UPIs, USIs, and PKs. Values generated for GENERATED ALWAYS AS IDENTITY columns are always unique[2], but those generated for GENERATED BY DEFAULT AS IDENTITY are only unique if you also specify a UNIQUE constraint on the column. See "CREATE TABLE" in the chapter "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4* for more information about identity columns.

Do *not* attempt to update GENERATE ALWAYS identity column values under any circumstances. This operation is not permitted and Teradata returns an error if you attempt to perform it.

ROWID Column

Every Teradata base table has a system-generated column named ROWID. The fields in this column contain the RowID value for their rows. You can specify a ROWID column in a join index definition to facilitate query processing using noncovering join indexes, but you *cannot* use the ROWID keyword for any other purpose.

The data type for ROWID values is BYTE(10). If the primary index for a table is changed from non-partitioned to partitioned, or if the partitioning expression for a partitioned primary index is changed because of an ALTER TABLE statement, then the value of ROWID for any affected rows is likely to change. Similarly, any changes to the primary index columns or partitioning columns for a PPI table by means of an ALTER TABLE statement are likely to change the value of ROWID for many table rows.

Users can only specify the ROWID keyword in a CREATE JOIN INDEX statement to enable noncovering join indexes to join with base table columns to optimize query processing (see "Partially Covering a Query With a Join Index" on page 11-25 and "Create Join Index" *Teradata RDBMS SQL Reference, Volume 4*).

Attempts to use the ROWID keyword in any other context, such as selecting or deleting from, updating, or inserting rows into base tables, views, or derived tables return an error.

2 Note that if you load the same row twice into an identity column SET table, it is not rejected as a duplicate because it is made unique as soon as an identity column value is generated for it. This means that some preprocessing must still be performed on rows to be loaded into identity column tables if real world uniqueness is a concern.

PARTITION Column

A PARTITION column is generated for each Teradata table defined with a partitioned primary index (PPI). The fields in this column contain the number of the partition in which their rows are stored.

The data type for PARTITION values is INTEGER, with the following valid range values.

FOR this type of table …	The valid range for PARTITION is …
PPI	1 - 65 535
NPPI	0

The default title for system-generated partition columns is PARTITION.

The principle use for the values stored in the PARTITION column is to retrieve rows, but they can be specified in any DML statement or referenced in the DROP RANGE WHERE clause of an ALTER TABLE statement.

You can also specify PARTITION as a column name in an SQL SELECT statement in order to determine the partition to which various rows in a table belong. As with any other column, you must fully qualify all ambiguous PARTITION references. Like any other column, PARTITION can be qualified by a database name and table name, and it can be referenced wherever a column of the table can be referenced.

IF this many tables specified in a query have an explicit column named PARTITION …	THEN …
1	an unqualified reference to PARTITION refers to the explicitly defined column.
> 1	all references to PARTITION must be qualified.

If multiple tables are referenced in a query, then all references to the system-derived column PARTITION must be qualified.

PARTITION is not included in the list of columns returned when you specify * in a query; it must be explicitly selected. In particular, the PARTITION column cannot be accessed through a view on the table unless the view definition explicitly specifies the column.

Because PARTITION is not a reserved word, you can also assign the name to user-defined columns. If you do this, however, the system-defined PARTITION column for any such table is not accessible from an SQL query because there is no way to distinguish it from the user-defined column with the same name.

PARTITION is not returned as a column by a HELP TABLE or HELP COLUMN statement, and it does not appear in the data dictionary as a column of the referenced table.

Do *not* attempt to update partition values under any circumstances. This operation is not permitted and Teradata returns an error if you attempt to perform it.

The value of PARTITION takes no space in the table: the internal partition number is extracted from the row and converted to its external partition number. An error is returned to the requestor if there is no external partition number for the internal partition number.

PARTITION is equivalent to a value expression where that expression is identical to the partitioning expression defined for the primary index of the table with column references appropriately qualified as needed (or zero if no partitioning expression for the primary index is defined). This is true if, and only if, the row specifies the correct internal partition. For example, a query against a PPI table that specifies the predicate `WHERE PARTITION <> partitioning_expression` should always return 0 rows. If any rows are returned, then they are not partitioned properly, and the table should be revalidated immediately.

If the partitioning expression for a PPI table is changed, then the values of PARTITION for many rows in the table might change.

Usage Examples for PARTITION

The following table definitions are used for this set of examples:

```
CREATE TABLE Orders
     (o_orderkey INTEGER NOT NULL,
      o_custkey INTEGER,
      o_orderstatus CHAR(1) CASESPECIFIC,
      o_totalprice DECIMAL(13,2) NOT NULL,
      o_orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
      o_orderpriority CHAR(21),
      o_clerk CHAR(16),
      o_shippriority INTEGER,
      o_comment VARCHAR(79))
   PRIMARY INDEX (o_orderkey)
     PARTITION BY RANGE_N(
     o_orderdate BETWEEN DATE '1992-01-01' AND DATE '1998-12-31'
     EACH INTERVAL '1' MONTH)
   UNIQUE INDEX (o_orderkey);
```

```
CREATE TABLE Lineitem
     (l_orderkey INTEGER NOT NULL,
      l_partkey INTEGER NOT NULL,
      l_suppkey INTEGER,
      l_linenumber INTEGER,
      l_quantity INTEGER NOT NULL,
      l_extendedprice DECIMAL(13,2) NOT NULL,
      l_discount DECIMAL(13,2),
      l_tax DECIMAL(13,2),
      l_returnflag CHAR(1),
      l_linestatus CHAR(1),
      l_shipdate DATE FORMAT 'yyyy-mm-dd',
      l_commitdate DATE FORMAT 'yyyy-mm-dd',
      l_receiptdate DATE FORMAT 'yyyy-mm-dd',
      l_shipinstruct VARCHAR(25),
      l_shipmode VARCHAR(10),
      l_comment VARCHAR(44))
   PRIMARY INDEX (l_orderkey)
     PARTITION BY RANGE_N(
     l_shipdate BETWEEN DATE '1992-01-01' AND DATE '1998-12-31'
     EACH INTERVAL '1' MONTH);
```

The following are examples of the usage of PARTITION.

Delete all orders in the Order table from partition 1.

```
DELETE FROM Orders
WHERE Orders.PARTITION = 1;
```

Delete all orders in the Orders table from partitions 30 through 32, inclusive.

```
DELETE FROM Orders
WHERE Orders.PARTITION BETWEEN 30 AND 32;
```

Use an INSERT … SELECT statement to copy orders from partitions 1 and 2 of the Orders table into the Old_Orders table, then delete them from Orders.

```
INSERT INTO Old_Orders
SELECT *
FROM Orders
WHERE Orders.PARTITION IN (1,2)
; DELETE FROM Orders
WHERE Orders.PARTITION IN (1, 2);
```

In the following example, the value of PARTITION is not returned as one of the column values because it was not explicitly specified in the select list for the query.

```
SELECT *
FROM Orders
WHERE Orders.PARTITION = 10
AND Orders.o_totalprice > 19.99;
```

PARTITION does not have to be qualified in this example because its use is unambiguous.

```
SELECT Orders.*, PARTITION
FROM Orders
WHERE Orders.PARTITION = 10
AND Orders.o_totalprice > 100;
```

PARTITION must be qualified in the two following examples to distinguish between PARTITION values in the Orders table and PARTITION values in the Lineitem table.

```
SELECT *
FROM Orders, Lineitem
WHERE Orders.PARTITION = 3
AND Lineitem.PARTITION = 5
AND Orders.o_orderkey = Lineitem.l_orderkey;

SELECT Orders.*, Lineitem.*, Orders.PARTITION
FROM Orders, Lineitem
WHERE Orders.PARTITION = 3
AND Lineitem.PARTITION = 5
AND Orders.o_orderkey = Lineitem.l_orderkey;
```

This example is not valid because PARTITION cannot be referenced in the VALUES clause.

```
INSERT INTO Orders VALUES (PARTITION, 10, 'A', 599.99, DATE
'2001-02-07','HIGH', 'Jack', 3,'Invalid insert');
```

The following two examples provide a list of the active partitions in the Orders table.

```
SELECT DISTINCT PARTITION (FORMAT '999')
FROM Orders
ORDER BY PARTITION;

SELECT DISTINCT CAST (PARTITION AS BYTEINT)
FROM Orders
ORDER BY PARTITION;
```

The following example counts the number of rows in each active partition.

```
SELECT PARTITION, COUNT(*)
FROM Orders
GROUP BY PARTITION
ORDER BY PARTITION;
```

You cannot select PARTITION through this view because it was not specified explicitly in the select list.

```
CREATE VIEW OrdersV AS
SELECT * FROM Orders;
```

Because it is explicitly selected in this view definition, PARTITION values are returned to the requestor if all columns are selected from the view using the * literal.

```
CREATE VIEW OrdersVP AS
SELECT Orders.*, PARTITION
FROM Orders;
```

The following query finds orders for which at least one lineitem was shipped in the same month as the order was recorded.

```
SELECT O.o_orderkey
FROM Orders AS O, Lineitem AS L
WHERE O.PARTITION = L.PARTITION
AND O.o_orderkey = L.l_orderkey
GROUP BY O.o_orderkey;
```

Value Compression

Introduction

The column attributes COMPRESS and NULL (see the chapter "Default Value Control Phrases" in *Teradata RDBMS SQL Reference, Volume 3*) are useful for minimizing table storage space. You can use these attributes to selectively compress as many as 255 distinct, frequently repeated, column values, to compress all nulls in a column, or to compress both.

Compression has two principal applications:

- Reducing storage costs
- Enhancing system performance

Compression reduces storage costs by storing more logical data per unit of physical capacity. Optimal application of compression produces smaller rows, which in turn results in more rows stored per data block and fewer data blocks.

Similarly, compression enhances system performance because there is less physical data to retrieve per row for queries. Additionally, because compressed data remains compressed while in memory, the file system segment (FSG) cache[3] can hold more logical rows, thus reducing disk I/O.

Compression is transparent to applications, ETL utilities, ad hoc queries, and views. Experience with real world customer production databases with very large tables indicates that compression produces performance benefits for a table even when more than 100 of its columns have been compressed. There does not appear to be a downside to properly employing this technique.

Column compression provides the following capacity and performance benefits.

- Enhanced storage capacity
- Improved response time for table scans
- Reduced disk I/O traffic
- Moderate to little CPU savings

3 The FSG cache buffers disk I/O.

How Teradata Compresses Data

Teradata uses a non-adaptive, lossless, corruption-resistant compression algorithm called the Dictionary Index method to compress values independently on a column-by-column basis. Lossless means that although the data is compacted, there is no loss of information as there is, for example, with many audio and video compression algorithms (see "Tradeoffs Between Compression and the Net Capacity Required to Store Compressed Values" on page 14-43 for details).

Unlike many common compression algorithms, Teradata does not replace values to be compressed with encoded representations of those values. Instead, it stores one copy of a compressed value per column in the table header and nothing at all in any row that contains that value. The mechanism of resolving which compressed values belong to which rows is based on using presence bits that index particular values stored in a well known place in the table header.

The granularity of Teradata compression is the individual field of a row, which is the finest level possible. Field compression optimizes concurrency as well as offering superior performance for query and update processing when compared to row-level or block-level compression schemes. Row- and block-level compression methods require additional system resources to uncompress the row or block whenever it might contribute to a result set, whether it eventually does contribute to the result or not. Furthermore, field compression allows compression to be optimized for the data type of each column.

When you designate a set of column values for compression, Teradata adds space to Field 5 of the appropriate table header to store the compressed value set (see "Field 5" on page 14-10).

Because Teradata compression is completely internal to the system, it is transparent to ETL operations, queries using base table access, queries using view access, and all application software.

Default Settings for NULL and COMPRESS

Unless you explicitly specify different options, the following attributes are defined by default for any column in a Teradata table.

- Nulls are allowed.
- Compression is not in effect.

Meaning of The COMPRESS Attribute In Table DDL

The effect of a COMPRESS specification depends on the format of the definition.

IF COMPRESS is defined with …	THEN …
no argument	all nulls for the column are compressed to zero space.
one or more constant arguments	• Each occurrence of a specified constant is compressed to zero space. • All nulls for the column are compressed to zero space.

The following CREATE TABLE fragment specifies that all occurrences of the job titles "cashier," "manager," and "programmer" for the jobtitle column as well as all nulls are to be compressed to zero space.

This definition saves 30 bytes for each row whenever an employee has any of the following job titles:

- null
- cashier
- manager
- programmer

```
CREATE TABLE Employee
  (employee_number INTEGER
     ...
   jobtitle        CHARACTER(30) COMPRESS ('cashier',
                   'manager', 'programmer')
     ...
);
```

Rules and Guidelines for Compressing Columns

The following rules, guidelines, and limitations apply to compression:

- You cannot compress more than 255 distinct values for an individual column.
- You cannot create a table that has more bytes compressed than there is room to store them in the table header.
- You cannot create a table if the expansion of its compression values exceeds the maximum size (8 192 characters) of the compressvaluelist column in the DBC.TVFields system table. As a guideline, note that 255 CHARACTER(29) values barely fit into the 8 192 character limit.

  The DDL text for CREATE TABLE statements is stored in DBC.TVFields for use by SHOW TABLE reports.

- You can compress only the following types of data.
  - Nulls
  - Zeros
  - Blanks
  - Constants having any of the following fixed length data types.
    - BYTE (up to 255 bytes)
    - BYTEINT
    - CHARACTER (up to 255 characters)
    - DATE (expressed as `COMPRESS (DATE 'yyyy-mm-dd')`)
    - DECIMAL/NUMERIC
    - DOUBLE PRECISION
    - FLOAT
    - INTEGER
    - REAL
    - SMALLINT
- You cannot compress the following types of columns:
  - Component of the primary index
  - Identity columns
  - Volatile table columns
  - Derived table columns
  - Spool table columns
  - Referenced primary key columns
  - Referencing foreign key columns
- You can compress columns that are a component of a secondary index with the following qualification: MultiLoad operations on a table with one or more secondary indexes can take longer than they would if the table were not compressed.

  You can avoid this problem by dropping any compressed secondary indexes prior to starting the MultiLoad job and then recreating them afterward.

  Be aware that this performance issue does *not* apply to other load utilities.
- Because spool tables are not compressed, any join operation that includes compressed columns causes the rows that are built for spool to be decompressed.
- The COMPRESS attribute for a column cannot be changed after it has been defined.

  The workaround for this is to perform an INSERT … SELECT into a new table defined with a revised set of compress values.

Compression of Zipfian Data

Introduction

Value compression is optimal for columns having a relatively small range of distinct values. For columns having this characteristic, compression can often reclaim as much as 80 percent of the space required to store the complete set of values for the column when they are not compressed. A distribution like this, for which the frequency of occurrence of a variable is inversely proportional to its rank, is often referred to as Zipfian, after the man who first described it quantitatively.

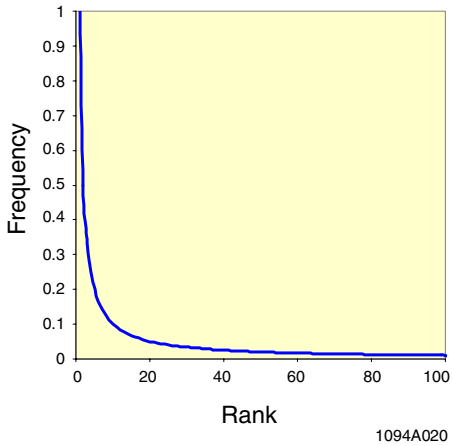
George Kingsley Zipf was a linguist who studied the frequency of occurrence of words in natural language text. Zipf found that the frequency of a particular word in text written in virtually any language was approximately inversely proportional to the rank of its frequency of occurrence. The approximation Zipf derived for this relationship is the following power-law function, where the value for the exponent a is approximately 1:

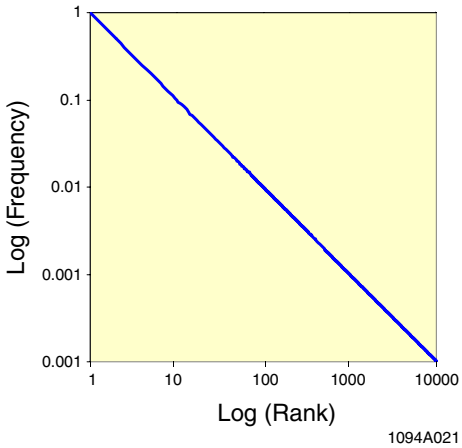
$$P_i \cong \frac{1}{i^a}$$

where:

Equation element ...	Represents ...
P	the event being examined.
i	the rank of event P.
a	an exponent whose value is approximately 1.

The following graphs illustrate typical Zipfian characteristics by plotting frequency of some variable as a function of rank. The first plot is in standard form and the second is in log-log form.





This same relationship had been observed in the late 19th century by the political economist and sociologist Wilfredo Pareto, who named it the Law of the Trivial Many and the Critical Few. The studies that led Pareto to formulate this law concerned the worldwide distribution of wealth, in which he found that roughly 80 percent of all wealth was owned by only 20 percent of the population. Total Quality Management advocate Joseph Juran later renamed this law the 80:20 rule. In this form, it states that roughly 80 percent of the effects of a phenomenon are accounted for by only 20 percent of its causes. Simply put, a few distinct items account for the majority of occurrences of a given type of event.

Compression is maximally effective for this type of data demographic (see "Example 1: Compression Applied to a Column With Zipfian Demographics" on page 14-25, "Example 2: U.S. Male First Names Obey the Zipf Law" on page 14-27, "Example 3: Applying Compression to Zipfian Data from Public Databases" on page 14-28 and "Compressing A World Name Database" on page 14-34 for examples).

Many natural phenomena are found to observe the law, for example:

- City populations
- Corporate revenues
- Internet traffic by site
- Person names in a population

Example 1: Compression Applied to a Column With Zipfian Demographics

Consider the following scenario as an example of how multivalued compression works on a column with Zipfian data demographics.

Assume the following properties for the table in question:

- Cardinality is one billion rows.
- Number of distinct values in the column under observation is 10 000.
- Data type of the column under observation is CHARACTER(8).

The following table ranks the cardinalities of the top 6 values for the column in question as well as the ten thousandth ranking value. The difference between the cardinalities for the top and bottom ranks is four orders of magnitude.

Rank	Frequency of Occurrence	Cardinality Per Value
1	1/1	102 170 030
2	1/2	51 085 015
3	1/3	34 056 677
4	1/4	25 542 507
5	1/5	20 434 006
6	1/6	17 028 338
…	…	…
10 000	1/10 000	10 217
Total:		1 000 000 000

The following table shows the results of the study. The data indicates that for a Zipfian distribution, it is optimal to compress as many values as possible. In this particular example, by compressing the top 255 values, the required storage for the table is cut in half, from 8 billion bytes to just under 4 billion bytes.

Top *n* Values	Cumulative Number of Rows in Top n	Cumulative Percentage of Rows in Top n	Storage Capacity Required for Column (bytes)	Percent Compression	Compression Factor
0	0	0	8 000 000 000	0	0.000
1	102 170 030	10.22	7 307 640 562	9	1.095
3	187 311 721	18.73	6 751 508 230	16	1.185
7	264 912 291	26.49	6 255 706 228	22	1.279
15	339 023 555	33.90	5 787 821 240	28	1.382
31	411 463 761	41.15	5 333 309 828	33	1.500
63	483 087 068	48.31	4 885 343 855	39	1.638
127	554 306 597	55.43	4 440 628 586	44	1.802
255	625 325 405	62.53	3 997 560 037	50	2.001

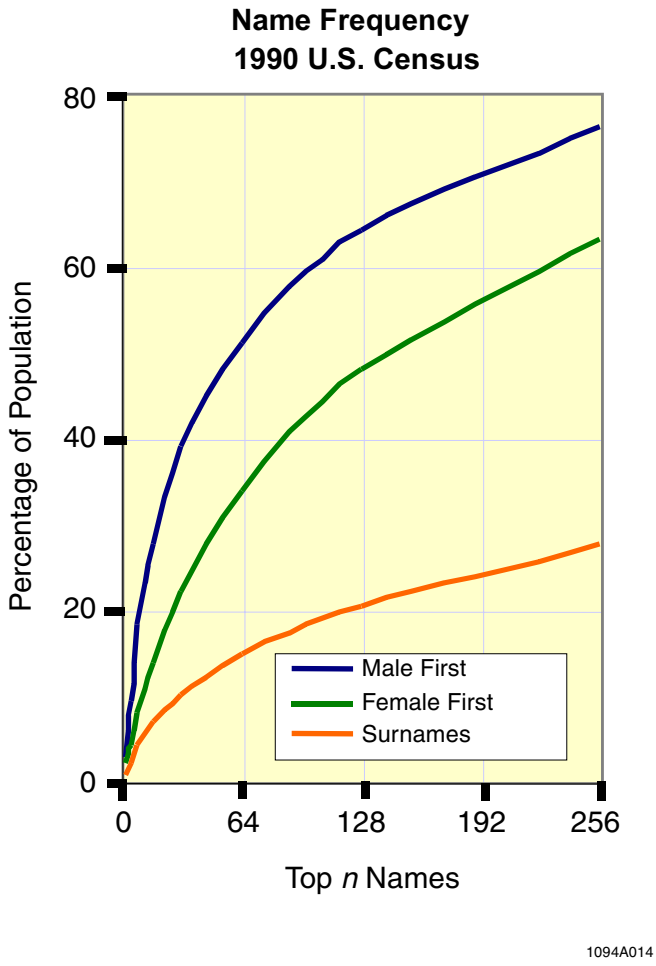
Example 2: U.S. Male First Names Obey the Zipf Law

The 1990 United States census provides a convenient source for publicly available U.S. names data. The following table lists the ten most frequent first names for U.S. males and the percentage of the male population having each name.

Rank	First Name	Percentage of Male Population Having This Name
1	James	3.318
2	John	3.271
3	Robert	3.143
4	Michael	2.629
5	William	2.451
6	David	2.363
7	Richard	1.703
8	Charles	1.523
9	Joseph	1.404
10	Thomas	1.380

These ten first names alone account for the first names of 23.185 percent of all U.S. males according to the 1990 census.

The following graph shows the cumulative frequency distribution of the 255 most common male names, female names, and surnames. It is interesting to note that compression could compress more than 60 percent of all first names for this population. This column is an excellent candidate for compression. However, the surname column compresses only about 25 percent. Is it beneficial to compress surname? This is where the issues of storage efficiency and break-even points come into play (see "Storing Data Efficiently" on page 14-31 and "Computing and Interpreting the Break-Even Point for Compression" on page 14-44).



Example 3: Applying Compression to Zipfian Data from Public Databases

Examples of the compressibility of diverse data are helpful to fully appreciate the power of multivalued compression. The following statistics, sampled from various public databases, show the percentage of data items represented by the top *n* most frequently occurring values, where *n* is 1, 3, 7, 15, 31, 63, 127, 255[4]. Actual compression savings are a function of the number of values compressed (*n*), the native data type storage size, and the frequency count.

4 See "Computing and Interpreting the Break-Even Point for Compression" on page 14-44 for the reason these values are used in this example.

Database	Source	Percentage of Population Represented by Top n Most Frequent Values							
		1	3	7	15	31	63	127	255
U.S. population by city	U.S. census 2000	3.8	7.0	9.9	13.4	18.0	24.1	32.1	40.0
U.S. population by county[a]		3.4	6.9	11.5	18.5	28.2	41.4	56.9	72.8
U.S. male first names	U.S. census 1990	3.3	9.7	18.9	28.0	39.2	51.1	64.3	76.4
U.S. female first names		2.6	4.7	8.4	14.1	22.3	34.1	48.2	63.3
U.S. surnames		1.0	2.5	4.7	7.1	10.7	15.3	20.9	27.7
U.S. population by state		12.0	26.2	44.7	66.0	90.6	100	100	100
Irish automobiles by make	Irish vehicle statistics 1996	16.5	42.2	69.8	89.2	99.7	100	100	100
Australian population by age	AusStats 2000	1.6	4.8	11.1	23.1	46.1	86.1	100	100
World population by country	U.N. POPIN[b]	21.1	42.4	53.5	66.4	80.1	91.4	89.9	100

a. It is important to calculate the demographics of value frequency correctly. Consider county names in a database of U.S. population by county. It is not correct to simply compress the 255 most populous counties. This is because county names are unique within a state, but may be nonunique in the nation. For example, 31 states have a Washington county.

The most frequent 255 county names are not necessarily the same set of values as the most populous 255 counties. Distinctly different Washington counties might appear in the list so that "Washington County" is listed multiple times. Therefore, the top 255 county names must be determined by a frequency analysis of the data in the County column. Census data shows that 61.5 percent of the U.S. population lives in the 255 most populous *counties*. However, 72.8 percent of the population lives in the top 255 county *names*.

b. See “Basic Country Coding Demographics” on page 14-34 for the definition of POPIN.

Determining the Top *n* Values for a Column

The dictionary table DBC.Columns stores all sorts of information about the columns of your tables (see *Teradata RDBMS Data Dictionary Reference* for further information). You can harness this information to your advantage by using simple SQL queries to determine column data value rankings.

You can further tune your queries against DBC.Columns by using the BTEQ .RETLIMIT command to restrict the returned values to the top *n* in rank (see *Teradata BTEQ Reference* for further information).

Storing Data Efficiently

Introduction

Compression is one of several techniques available for storing data efficiently. Some of the other most commonly used methods are listed below:

- Specifying different DECIMAL and INTEGER precisions
- Using standardized encodings for common words
- Specifying VARCHAR instead of compressed CHARACTER data types

Before studying the next compression example, it is useful to understand how these techniques provide superior alternatives to compression in some circumstances.

Efficient Storage of DECIMAL/NUMERIC Data

Storage requirements for decimal data types are a function of the precision required of the values to be stored. The following table indicates the number of bytes used to store decimal numbers of various precisions:

Number of Precision Digits	Number of Bytes Stored
1 - 2	1
3 - 4	2
5 - 9	4
10 - 18	8

You can achieve optimal space savings by using compression together with an efficient decimal size. An example would be to define decimal precision such that a 4-byte representation was used instead of an 8-byte representation: for example, DECIMAL(9) instead of DECIMAL(10). Then, compression could be used on the most frequently occurring values to reduce the storage overhead still further.

Efficient Storage of INTEGER Data

Like the DECIMAL/NUMERIC data type, the INTEGER family of data types offers different levels of precision that you can harness to reduce the number of bytes used to store integer numbers. The following table indicates the number of bytes used to store integer numbers of various precisions:

Integer Data Type	Number of Bytes Stored
BYTEINT	1
SMALLINT	2
INTEGER	4

You can achieve optimal space savings by using compression together with an efficient integer data type. An example would be to define the precision such that a 1-byte representation was used instead of a 4-byte representation: for example, BYTEINT instead of INTEGER. Then, compression could be used on the most frequently occurring values to reduce the storage overhead still further.

Efficient Storage Using Standardized Encodings

There are many standardized encodings of commonly used words. For example, the United States Postal Service (USPS) encodes all U.S. states and possessions using a two-character abbreviation. The lengths of the unabbreviated state and possession strings range from four characters (Guam, Iowa, Ohio, and Utah) to 30 characters (Federated States of Micronesia). While you could define a State column as CHARACTER(30) and compress it, or as VARCHAR(30), the easiest and most compact method of denoting U.S. states and possessions is to use the two-character USPS encodings.

As demonstrated by the ISO 3166 country name encodings examined in "Compressing A World Name Database" on page 14-34, there are other situations where compression is a more efficient storage technique than encodings.

Efficient Storage of Character Data

For character data, an alternative to encodings and compressing fixed-length CHARACTER(*n*) strings is to specify the variable- length VARCHAR data type. The number of bytes used to store each VARCHAR field is the length of the data item plus 2 bytes. Contrast this to the fixed-length CHARACTER data type which uses *n* bytes per row, regardless of the actual number of characters in each individual field.

The demographics of the data determine whether VARCHAR or CHARACTER plus compression is more efficient. The most important factors are these:

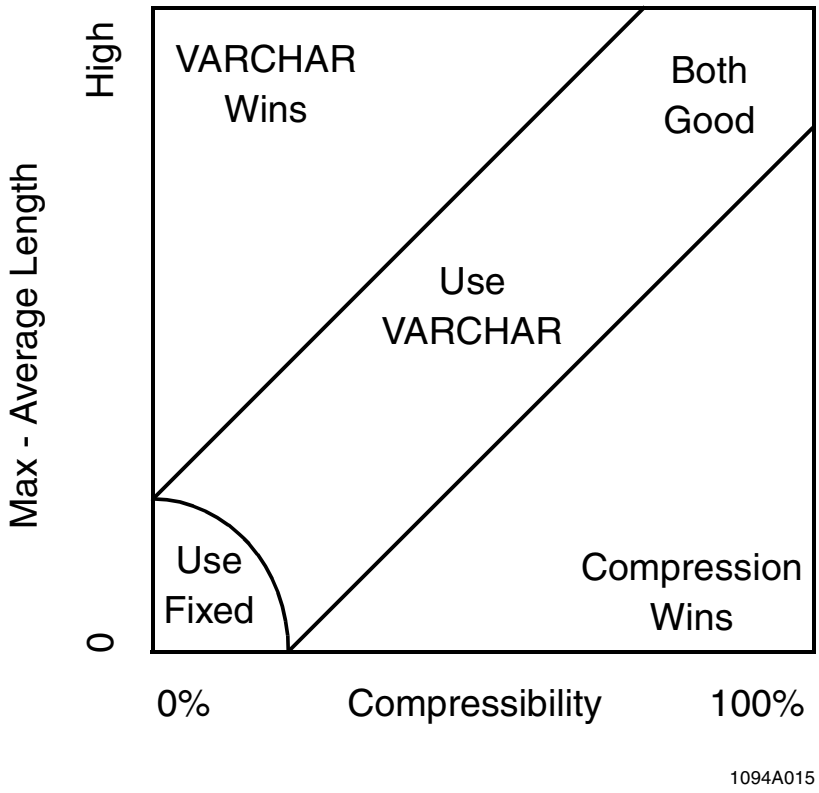
- Maximum field length
- Average field length
- Compressibility of the data

Evaluate the following factors when determining which approach to storing the data is the more efficient:

- VARCHAR is more efficient when the difference of maximum and average field length is high and compressibility is low.
- Compression with CHARACTER is more efficient when the difference of maximum and average field length is low and compressibility is high.

When neither CHARACTER nor VARCHAR is clearly a superior choice, use VARCHAR because VARCHAR data requires slightly less CPU resource to manipulate than CHARACTER data.

The following chart indicates the tradeoffs between using VARCHAR or compressed CHARACTER data:



The same considerations apply to deciding between VARBYTE or BYTE types and between VARCHAR(n) CHARACTER SET GRAPHIC or CHARACTER(n) CHARACTER SET GRAPHIC types.

Compressing A World Name Database

Introduction

Suppose you want to create a table that records the name and address for all humanity. Each row stores a name and address. A component of each address is a country. This topic analyzes whether compression is an effective method of reducing the storage overhead for the Country column.

Basic Country Coding Demographics

United Nations Population Information Network (POPIN) statistics indicate that in the year 2000, there were slightly more than 6 billion people on the planet and that those people resided in 187 distinct countries. Teradata can compress up to 255 values per column, so it is possible to compress the Country field in 100 percent of the rows in this table.

Data Encoding and Typing Considerations

The next questions to evaluate are how to type the Country field and whether it should be encoded or not.

The POPIN data shows that the longest country name has 37 characters. There are several existing standards for encoding country name, in particular the ISO standard geopolitical entity data model. This standard specifies both a short and long ISO country name, typed as VARCHAR(50) and VARCHAR(75), respectively.

Other published encoding standards use type specifications such as VARCHAR(30) and VARCHAR(40). The ISO 3166 standard provides 2 and 3 letter abbreviations for 237 different countries, with the longest "short" country name being 44 characters.

Another possibility to consider is the Internet Assigned Numbers Authority encoding of countries into two character fields.

For a fully normalized database, encoded data is typically stored in a lookup table, with the code acting as the primary key and a full description of each code being the principal, if not only, attribute.

The downside of encoding the country name and creating a separate lookup table is that a join is needed to process queries that require a full description of coded values in the answer set. Given the 6 billion row cardinality of the table, frequent joins between it and a lookup table might be a more expensive operation than you care to undertake.

Comparing the Effects of Different Data Typings on Storage Efficiency

The following table shows the storage capacity required to store the Country column as a function of the data type chosen. Though encoding of the Country field might or might not be likely because of the lookup table join considerations mentioned in "Data Encoding and Typing Considerations" on page 14-34, that option is presented in the capacity calculations for completeness.

CHARACTER(n) n =	Compress Values	Disk Capacity (gigabytes)
75	none	454
50	none	303
37	none	224
30	none	182
37	1	178
37	3	131
37	7	107
37	15	78
VARCHAR		63
37	31	48
37	63	24
3	ISO code	18
2	ISO code	12
37	127	8
37	187	6

Prior to the availability of multivalued compression, the most likely choice would have been to specify the column as VARCHAR (see "Storing Data Efficiently" on page 14-31). This would have required about 63 GB of storage capacity for just this column. With multivalued compression, a 10 to 1 reduction in capacity can be realized, and the field can be stored in just 6 GB, an order of magnitude savings. You can also see that there are some inefficient choices that could be made.

Typical Uses For The Capacity Reclaimed By Compression

Before continuing with an in-depth description of compression and how to calculate various analytic measures for helping you determine the usefulness of compressing specific data in specific columns, this topic examines several possible uses for the reclaimed storage capacity realized by compression.

- On an existing system, the introduction of compression is not likely to free a significant quantity of CPU resources. For those systems having residual CPU capacity, the space savings from compression might permit more applications to be introduced prior to the next system upgrade. If excess CPU capacity is already exhausted, then alternative ways to use reclaimed capacity must be explored.
- Perhaps the most obvious way to recycle newly liberated capacity is to keep more infrequently accessed historical data online (see “Capacity Planning and Data Access Frequency” on page 14-2).
- Another way to use the reclaimed capacity is to improve system availability by declaring fallback for some or all tables.
- Alternatively, the excess capacity can be harnessed to improve performance by adding new indexes to various SQL workloads (see the chapter "Database Foundations for the Teradata Index Wizard" in *Teradata RDBMS SQL Reference, Volume 2* and *Teradata Index Wizard User Guide* for information about automatic evaluation of workloads to develop more performant secondary indexes).

How the Row Header Indicates Column Nullability and Compressibility

Introduction

This topic explains how the settings of the presence bits for a column within the row header indicate its nullability and compressibility.

Presence Bits

Presence bits indicate the status of each column with respect to its nullability and compressibility. Compression presence bits are added to the row header of each row to specify how compression is used for that row. Each row has at least one octet[5] of presence bits and can have more, depending on the degree of the table and the cumulative number of values compressed. All rows for a given table have the same presence bits defined because nullability and compressibility are table attributes.

The first bit of the first presence bits octet is always set to 1, the result of which is that the first octet defines the nullability and compressibility for no more than seven columns. When necessary, additional presence bit octets are added to the row header. Although only 7 of the bits in the first octet are available for use as presence bits, all 8 bits of successive octets are available.

Each compression-enabled column defined for a table has as many as 8 presence bits set[6] (see "Computing and Interpreting the Break-Even Point for Compression" on page 14-44) depending on how many values are compressed.

5 The difference between bytes and octets is conceptual. The 8 bits of a byte define an atomic unit, while each individual bit of an octet is an atomic flag. For the purpose of storage capacity analysis, both are treated as bytes.

6 8 bits because 255 compressed values plus null require 2^8, or 256 bit combinations, to be represented.

The meaning of the number of presence bits set per column for compression of a single value is provided in the following table.

This type of presence bit field...	Has this many presence bits ...	For this type of column ...
Nullability	0	Non-nullable.
	1	Nullable.
Compressibility	0	Not compressed or compressed on nulls only.
	1	Compressed on a non-null value.

For multivalued compression, the following equation calculates the number of compression presence bits required to define compression for a column in its table header:

$$\text{compression\_presence\_bits} = \text{ceiling}(\log_2(1 + \text{number\_of\_distinct\_values}))$$

where:

This term ...	Specifies ...
ceiling(x)	a function that returns the smallest integer greater than or equal to x. This function is common in the mathematics library of most programming languages. The ceiling function rounds the result of the expression up to the nearest integer, which is the exact number of presence bits required to account for both single-valued and multivalued compression information for the given table.
number_of_distinct_values	the number of distinct values to be compressed for the column.

The total number of presence bits for a given row is the sum of the nullability presence bits and the compressibility presence bits.

$$\text{presence\_bits} = \sum_{\text{col}=1}^{\text{col}=N} \text{nullability\_presence\_bits} + \text{compressability\_presence\_bits}$$

The following table expresses the same information in a slightly different way.

Compressible		Nullable	
Bit Value	**Meaning**	**Bit Value**	**Meaning**
0	The field is compressed.	0	The field is null.
1	A non-compressed field value present.	1	The field is not null.
• One to eight bits for each compressible column (see footnote 6 and "Computing and Interpreting the Break-Even Point for Compression" on page 14-44). • Valid for fixed length columns only.		One bit for each nullable column.	

The following table provides a comprehensive mapping of the presence bits and their various combinations for the single value compression case:

WHEN the presence bits for a column have these values …		THEN the column …	AND …
NULL	**COMPRESS**		
no bit	no bit	is not compressible	is not nullable.
no bit	0	is compressed	is not nullable.
no bit	1	contains non-compressed column value	is not nullable.
0	no bit	See the next table on this page for a description of these conditions.	
1	no bit		
0	0	is compressed	is null.
1	1	This combination is not valid because a column cannot simultaneously be null and populated by a value.	
0	1	is compressed	is not null.
1	0	contains a non-compressed, non-null value.	

Mappings of COMPRESS bit values for multivalued compression generalize from this specific case as illustrated by the following table:

IF the presence bit is …	THEN the data is …
1	not compressed. The corresponding compress bits are all 0.
0	compressed.

IF the corresponding compress bits are …	THEN the compress value is …
all 0	null.
not all 0	an index to the compress value list array in the table header.

Number of Presence Bits Required to Represent Compressed Values

Because compression is signaled by populating octets of presence bit fields in the row header, there is a tradeoff between the capacity required to add presence octets and the savings realized from compression at the base table row level: the size of the bit field required to express compression increases nonlinearly as a function of the number of values to be compressed. With respect to the savings gained by compression, this tradeoff is almost always worth doing.

The following table indicates the relationship between the number of values specified for compression per column and the number of compress bits required to indicate that number of values.

Nullable?	Number of Column Values Compressed	Number of Compress Bits in the Row Header Bit Field
Yes	Null	1[a]
No	1	1
No	2 - 3	2
No	4 - 7	3
No	8 - 15	4
No	16 - 31	5
No	32 - 63	6
No	64 - 127	7
No	128 - 255	8

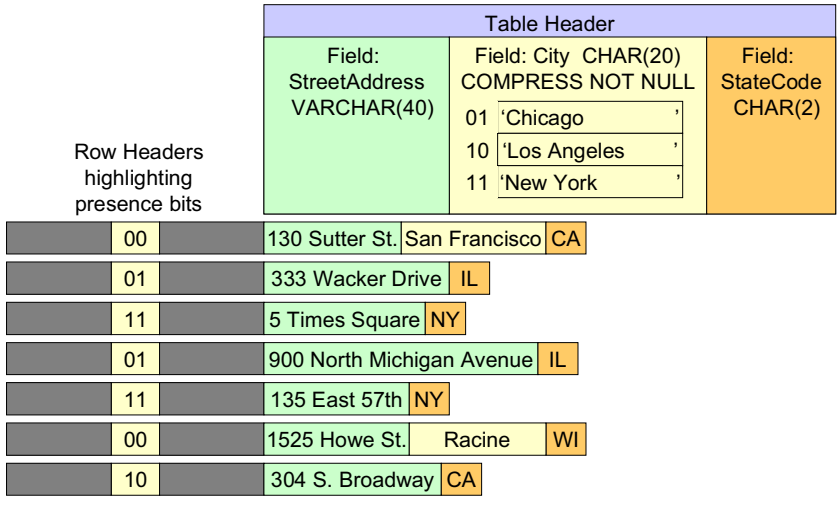
Nullable?	Number of Column Values Compressed	Number of Compress Bits in the Row Header Bit Field
Yes	1 + Null	1
Yes	2 - 3 + Null	2
Yes	4 - 7 + Null	3
Yes	8 - 15 + Null	4
Yes	16 - 31 + Null	5
Yes	32 - 63 + Null	6
Yes	64 - 127 + Null	7
Yes	128 - 255 + Null	8

a. Shared with the presence bit.

The compress bits value represents an index into the compress value list array for that column in the table header. This is illustrated by the following graphic showing that the presence bit pattern 10 points to the compressed character string "Los Angeles" in the table header. Note that because only 3 values are compressed for this column, 2 compression bits are required to uniquely identify them.

The table definition DDL for this illustration is as follows:

```
CREATE TABLE MVCompress (
  StreetAddress VARCHAR(40),
  City          CHARACTER(20) COMPRESS ('New York',
                'Los Angeles', 'Chicago') NOT NULL,
  StateCode     CHARACTER(2));
```



As you can see, all the possible combinations of bit patterns are used to identify and locate the values. For this particular example, the bit patterns and their corresponding values are those indicated in the following table:

FOR this pattern of presence bits …	The value for City is …	AND stored in …
00	not compressed.	the row.
01	'Chicago' and is compressed	the table header.
10	'Los Angeles' and is compressed	the table header.
11	'New York' and is compressed	the table header.

From the perspective of the row header, the best policy is to compress the highest number of values in each range, because doing so does not add any more space in the row header. Because of this, the best practice is to optimize multivalued compression for 1, 3, 7, 15, 31, 63, 127, or 255 values. You should evaluate all these possibilities to determine which yields the best compression for a particular column.

You must also take into account the number of bytes compression transfers to the table header, because those bytes count against the maximum row length of 64K bytes.

The compressed values are stored adjacent to one another in ascending value order and descending order of field size following the end of Field5 in the table header. See "Table Header Components" on page 14-7.

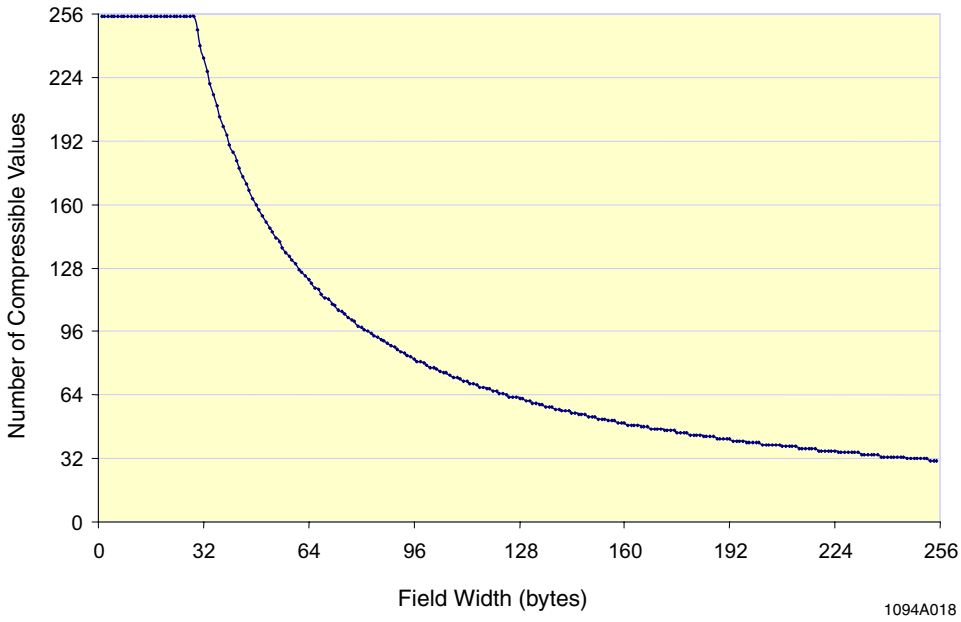
Tradeoffs Between Compression and the Net Capacity Required to Store Compressed Values

While compression removes specified values from row storage, those values do not disappear: they must be stored somewhere.

The presence bits in the row header index into field 5 of the table header, where the compressed values are stored, once per column per AMP. This is why the number of AMPs figures into the calculation of compression efficiency (see "Equation 6: Net Capacity Usage" on page 14-52).

Table header rows are limited in length to 64K just like other Teradata rows; therefore, there is a limit to how many bytes can be compressed for a given column. If the number of bytes compressed exceeds the maximum row length, then the CREATE or ALTER TABLE statement used to create the new table is not valid and the DDL statement aborts. This is true even if the number of values specified for compression does not exceed the upper limit of 255.

The following graph plots the number of compressible values that can be specified for a column as a function of field width:



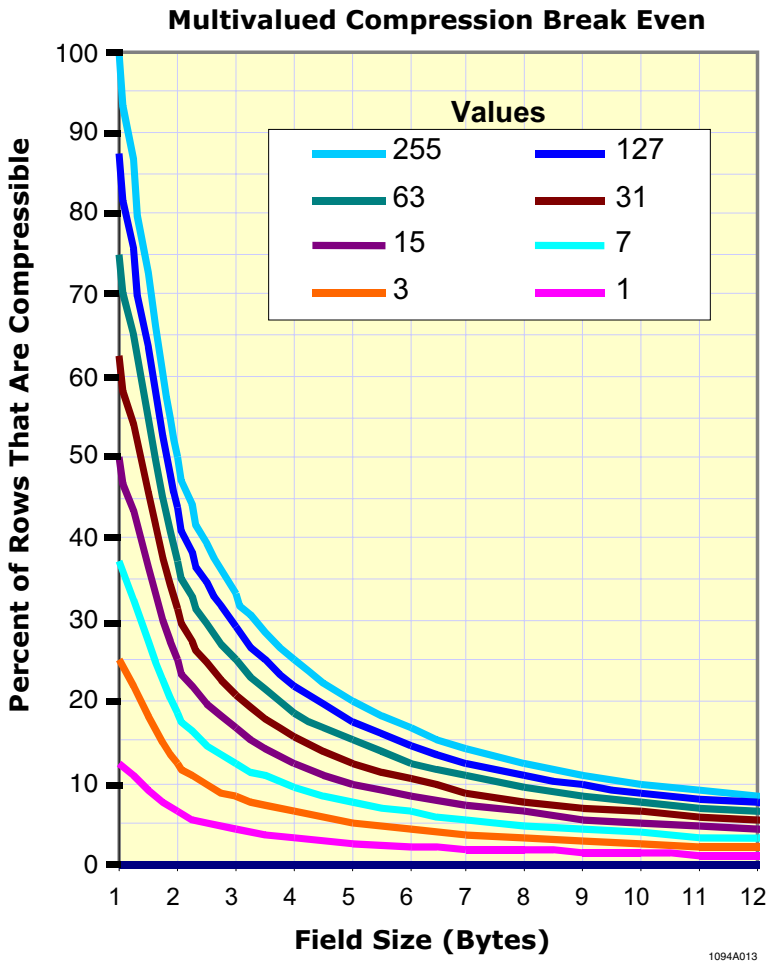
Not surprisingly, the plot clearly indicates that the wider the field, the fewer the number of values that can be compressed for the column. Particularly for wider columns, this means that in order to optimize compression, you must carefully analyze your tables to determine which values occur the most frequently and then limit compression to the top *n* values from that list.

Computing and Interpreting the Break-Even Point for Compression

Introduction

The break-even point for compression is the point at which there is zero net savings as computed by "Equation 7: Net Capacity Usage With Fallback Not Enabled" or "Equation 8: Net Capacity Usage Compression With Fallback Enabled."At this point, the savings in field storage are exactly balanced by the overhead for the compression bits field (see "Computing and Interpreting the Break-Even Point for Compression" on page 14-44). In other words, compressing the specified value for a column neither adds to nor subtracts from the net storage required to compress the value.

The following graphic indicates the break-even point for a column for various numbers of distinct values by representing the percentage of compressible rows as a function of the field size width in bytes.



The break-even point for the efficiency of compression on a column varies as a function of the size of the compressed field. This is obvious. The more data (larger field size) you can compress, the more storage space you save.

A break-even point represents a boundary condition. If compression of a value or null for a column results in a higher percentage of rows being compressed (expressed as a negative value for net capacity usage) than the break-even point, then it is worth doing; otherwise, it provides no benefit.

Computing the Break-Even Percentage

Another way to look at breaking even is to compute the percentage of rows in the table that must be compressed in order to realize no net savings. When you can compress a higher percentage of rows than this through compression (expressed as a higher value for the break-even percentage), then you should compress the value.

Use the following equation to compute the break-even percentage for value compression on a column.

Equation 1: Break-Even Percentage

$$\text{Break-even percentage} = \left(\frac{\text{table\_header\_space} + (\text{fallback\_factor} \times \text{row\_header\_space})}{\text{fallback\_factor} \times \text{cardinality} \times \text{field\_size}}\right) \times 10$$

where:

Variable name ...	Specifies ...
table_header_space	• the number of table header bytes used by the compressed column for 64-bit systems. • the number of table header bytes used by the compressed column + 2 for 32-bit systems. $\text{table\_header\_space} = \text{number\_of\_AMPs} \bullet \sum_{\text{col}=1}^{\text{col}=N} \text{field\_size}_{\text{col}}$ This value indicates the number of additional bytes required in the table header to compress the specified value, nulls, or both.
fallback_factor	whether fallback is enabled for the table. • 1 = fallback not enabled • 2 = fallback enabled
row_header_space	the number of presence octets (see "Equation 9: Additional Presence Octets" on page 14-54) in the table. $\text{row\_header\_space} = \text{table\_cardinality} \bullet (\text{presence\_octets} - 1)$ This value indicates the number of additional bytes required in the row header to compress the specified value, nulls, or both.

Variable name ...	Specifies ...
cardinality	the number of rows in the table.
field_size	the width of the compressed column in bytes.
$field\_size_{col}$	the number of table header bytes used by the compressed column + 1.

As you can see in "Equation 1: Break-Even Percentage" on page 14-45, field size, fallback, table header space, row header space, and cardinality are all factors in this determination, so the break-even point is different depending on the demographics of the case being evaluated.

Determining How Much Compression Can Be Realized

Introduction

You can determine how much compression can be achieved for a set of parameters by calculating its percent compression. Percent compression is the savings when considering the compressed size in comparison to the uncompressed size. The following tables show the percent compression for a billion row table along several dimensions:

- Column field widths of 2, 4, 8, and 16 bytes.
- Number of compressed distinct values per column ranging through 1, 3, 7, 15, 31, 63, 127, and 255 (corresponding to the range maxima expressible by 1 to 8 compress bits in the row header).
- Number of rows with compressible fields equal to 20%, 50%, 80%, and 100% of the cardinality for the table.

The row header overhead for compression can be larger than the savings realized, thus producing a net increase, or expansion, in capacity. In other words, more storage capacity is required to store the presence bits in each row and the compressed data in the table header than is returned to the available storage pool by compression. This condition is indicated by the MINUS character in the tables.

The compression savings expressed are for individual columns. Overall system savings are dependent on the number of columns that are compressed and the individual savings realized from each column.

Calculating Percent Compression

Percent compression is a normalized measure of the reduction in total table capacity achieved by compression. "Equation 2: Percent Compression" is the equation for computing percent compression.

Equation 2: Percent Compression

$$\text{Percent compression} = \left(1 - \frac{\text{table\_capacity}_{\text{compressed}}}{\text{table\_capacity}_{\text{uncompressed}}}\right) \times 100$$

where:

This term ...	Specifies ...
$\text{table\_capacity}_{\text{compressed}}$	the value calculated by "Equation 4: Table Capacity for Compression" on page 14-51.
$\text{table\_capacity}_{\text{uncompressed}}$	the value calculated by "Equation 3: Uncompressed Table Capacity" on page 14-50.

Percent Savings Realizable With 20 Percent Compression

The following table indicates the percentage of savings that can be realized when 20 percent of the subject rows can be compressed. A MINUS sign indicates net capacity expansion rather than savings.

Number of Values	Field Width (bytes)			
	2	4	8	16
1	14	17	18	19
3	7	14	17	18
7	1	11	15	18
15	-	7	14	17
31	-	4	12	16
63	-	1	11	15
127	-	-	9	15
255	-	-	7	14

Percent Savings Realizable With 50 Percent Compression

The following table indicates the percentage of savings that can be realized when 50 percent of the subject rows can be compressed. A MINUS sign indicates net capacity expansion rather than savings.

Number of Values	Field Width (bytes)			
	2	4	8	16
1	44	47	48	49
3	37	44	47	48
7	31	41	45	48
15	25	37	44	47
31	19	34	42	46
63	12	31	41	45
127	6	28	39	45
255	-	25	37	44

Percent Savings Realizable With 80 Percent Compression

The following table indicates the percentage of savings that can be realized when 80 percent of the subject rows can be compressed.

Number of Values	Field Width (bytes)			
	2	4	8	16
1	74	77	78	79
3	67	74	77	78
7	61	71	75	78
15	55	67	74	77
31	49	64	72	76
63	42	61	71	75
127	36	58	69	75
255	30	55	67	74

Percent Savings Realizable With 100 Percent Compression

The following table indicates the percentage of savings that can be realized when 100 percent of the subject rows can be compressed.

Number of Values	Field Width (bytes)			
	2	4	8	16
1	94	97	98	99
3	87	94	97	98
7	81	91	95	98
15	75	87	94	97
31	69	84	92	96
63	62	81	91	95
127	56	78	89	95
255	50	75	87	94

Calculating the Efficiency of Compression

Introduction

This topic defines several measures of the efficiency of compression: percent compression, compression ratio, and the net capacity usage statistic.

Several factors are involved in determining whether it is useful to compress a column or not. The net capacity usage statistic consolidates these factors and provides an objective measure of how efficient it would be to compress a particular value for a column.

Table Capacity

Table capacity is a measure of the storage space occupied by a table. Table capacity varies as a function of compression. The more compression, the smaller the computed value for table capacity.

"Equation 3: Uncompressed Table Capacity" defines raw table capacity in the absence of compression.

Equation 3: Uncompressed Table Capacity

$$\text{Table\_Capacity}_{\text{uncompressed}} = (\text{number\_of\_AMPs})(\text{base\_table\_header\_size})$$

$$+ (\text{cardinality})(14 + \text{additional\_nullability\_presence\_bytes})$$

$$+ (\text{cardinality})\left(\sum_{\text{col}=1}^{\text{col}=N} \text{field\_size}_{\text{col}}\right)$$

where:

This term ...	Specifies the number of ...
number_of_AMPs	AMPs in the system.
base_table_header_size	bytes in the table header.
cardinality	rows in the table.
additional_nullability_presence_octets	extra presence octets required to account for the compressed values.
$\text{field\_size}_{\text{col}}$	number of bytes in a compressed column.

"Equation 4: Table Capacity for Compression" defines table capacity with multivalued compression enabled for one or more columns.

Equation 4: Table Capacity for Compression

$$\begin{aligned}\text{Table\_capacity}_{\text{compression}} = {} & \text{number\_of\_AMPs}(\text{base\_table\_header\_size}) \\ & + \text{number\_of\_AMPs}\left(\sum_{\text{col}=1}^{\text{col}=M}(\text{field\_size}_{\text{col}} \times \text{number\_distinct\_values}_{\text{col}})\right) \\ & + (\text{number\_of\_AMPs} \times \text{number\_of\_value\_list\_compression\_columns} \times 1 \text{ byte}) \\ & + \text{cardinality} \times (\text{additional\_presence\_octets} + 14) \\ & + \text{cardinality} \times \sum_{\text{col}=1}^{\text{col}=N}(\text{FieldSize}_{\text{col}}) \\ & - \sum_{\text{col}=1}^{\text{col}=N}(\text{rows\_with\_compressed\_values}_{\text{col}} \times \text{FieldSize}_{\text{col}})\end{aligned}$$

where:

This term ...	Specifies the number of ...
number_of_AMPs	AMPs in the system.
base_table_header_size	bytes in the table header.
$\text{field\_size}_{\text{col}}$*	the number of table header bytes used by the compressed column + 1.
cardinality	rows in the table.
additional_nullability_presence_octets	extra presence octets required to account for the compressed values.
$\text{field\_size}_{\text{col}}$	bytes in a compressed column.

Compression Ratio

The compression ratio is the ratio of uncompressed table capacity to compressed table capacity. The ratio is a normalized measure of the effectiveness of the compression. The larger the value for the compression ratio, the more effective the compression.

Equation 5: Compression Ratio

$$\text{Compression ratio} = \frac{\text{table\_capacity}_{\text{uncompressed}}}{\text{table\_capacity}_{\text{compressed}}}$$

Relationship Between Percent Compression and Compression Ratio

The values for percent compression and compression ratio for a given compression calculation are related as indicated in the following two equations. Let PC represent percent compression and CF represent compression factor.

$$\text{Percent compression} = 100 - \frac{100}{\text{CF}}$$

$$\text{Compression ratio} = \frac{100}{100 - \text{PC}}$$

Net Capacity Usage

Net capacity usage is a measure of the net savings realized from compressing a column.

IF the computed value for net capacity usage is …	THEN …
positive	the column is a poor candidate for compression because it would require more net space to compress the specified value than would be saved in row space.
negative	compression will produce a net space savings for the column because it reduces the net space used to store the compressed value and recovered row space savings.

Equation 6: Net Capacity Usage

$$\begin{aligned}\text{Net capacity usage} = \text{number\_of\_AMPs}&\left(\sum_{\text{col}=1}^{\text{col}=M}(\text{number\_of\_distinct\_values}_{\text{col}} \times \text{FieldSize}_{\text{col}})\right)\\ &+ (\text{number\_of\_AMPs} \times \text{number\_of\_compressed\_columns})\\ &+ (\text{cardinality} \times \text{number\_additional\_compression\_presence\_bytes})\\ &- \sum_{\text{col}=1}^{\text{col}=N}(\text{rows\_with\_compressed\_values}_{\text{col}} \times \text{FieldSize}_{\text{col}})\end{aligned}$$

Calculate the value for net capacity usage with "Equation 7: Net Capacity Usage With Fallback Not Enabled" on page 14-53 and "Equation 8: Net Capacity Usage Compression With Fallback Enabled" on page 14-53.

Equation 7: Net Capacity Usage With Fallback Not Enabled

$$\text{Net capacity usage} = \text{table\_header\_space} + \text{row\_header\_space} - \text{row\_vacated\_space}$$

where:

This term ...	Specifies the number of ...
table_header_space	table header bytes used by the compressed column set + 2. $\text{table\_header\_space} = \text{number\_of\_AMPs} \bullet \sum_{col=1}^{col=N} \text{field\_size}_{col}$ This value indicates the number of additional bytes required in the table header to compress the specified values, nulls, or both.
row_header_space	presence octets (see "Equation 9: Additional Presence Octets" on page 14-54) in the table. $\text{row\_header\_space} = \text{table\_cardinality} \bullet (\text{presence\_octets} - 1)$ This value indicates the number of additional bytes required in the row header to compress the specified values, nulls, or both.
row_vacated_space	compressed bytes in the compressed column. $\text{row\_vacated\_space} = \sum_{col=1}^{col=N} (\text{number\_compressed\_rows}_{col} \bullet \text{field\_size}_{col})$ This value indicates the number of bytes in the table saved by compression.

Equation 8: Net Capacity Usage Compression With Fallback Enabled

If the table has fallback enabled, some of the terms in the equation must be doubled.

$$\text{Net capacity usage} = \text{table\_header\_space} + 2(\text{row\_header\_space}) - 2(\text{row\_vacated\_space})$$

Presence Octets

The number of additional presence octets for a table is a measure of how many octets (see "Presence Bits" on page 14-37) are required to define the compressibility of all the values for all the columns for a given table beyond those needed to express nullability. "Equation 9: Additional Presence Octets" presents the formula for calculating the number of additional presence octets in a table.

Equation 9: Additional Presence Octets

$$\text{Number of additional presence octets} = \text{floor}\left(\frac{\sum_{col=1}^{col=N} \text{presence\_bits}_{col}}{\frac{8\ \text{bits}}{\text{octet}}}\right)$$

where:

This term ...	Specifies ...
floor(x)	a function that returns the greatest integer less than or equal to x. This function is common in the mathematics library of most programming languages. The floor function rounds the result of the expression up to the nearest integer, which is the exact number of presence octets required to store the compression information for the given table.
Σ $\text{presence\_bits}_{col}$	the sum of all the presence bits used for all columns (from column 1 through column N) in the table.
8 bits/octet	8 bits define an octet. This term produces the number of presence octets required to contain the presence bits used by all compressed columns in the table. Because the value can be a non-integer, it must be rounded up to the next highest integer number.

You can calculate the individual contributions of additional nullability presence octets and additional compressibility presence octets to the total additional presence octets required to compress column values and nulls using "Equation 10: Additional Presence Octets for Nullability" on page 14-55 and "Equation 11: Additional Presence Octets for Compressibility" on page 14-55.

Note that available presence octet bits are always used exhaustively before adding additional presence octets to the row header.

Equation 10: Additional Presence Octets for Nullability

$$\text{Number of additional nullability presence octets} = \text{floor}\left(\frac{\sum_{col=1}^{col=N} \text{nullability\_presence\_bits}_{col}}{\frac{8\ \text{bits}}{\text{octet}}}\right)$$

Equation 11: Additional Presence Octets for Compressibility

$$\text{Number additional compressibility presence octets} = \text{num\_additional\_presence\_octets} - \text{num\_additional\_nullability\_presence\_octets}$$

The following table provides examples of representative figures.

Number of Presence Bits for Nullability	Additional Presence Octets for Nullability	Residual Bits in Presence Octets After Accounting for Nullability	Number of Presence Bits for Compressibility	Additional Presence Octets for Compressibility
3	0	4	2	0
8	1	7	7	0
8	1	7	8	1
12	1	3	39	5
12	1	3	43	5

Detailed Calculation of Net Capacity Usage

"Equation 7: Net Capacity Usage With Fallback Not Enabled" on page 14-53 and "Equation 8: Net Capacity Usage Compression With Fallback Enabled" on page 14-53 describe net capacity usage at a high-level. "Equation 12: Net Capacity Usage" presents the same formula in greater detail.

Equation 12: Net Capacity Usage

$$\text{Net capacity usage} = \left((\text{AMPs})\left(\sum_{col=1}^{col=N} \text{field\_size}_{col}^{*}\right)\right) + (\text{cardinality})(\text{additional\_presence\_octets}) - \sum_{col=1}^{col=N} (\text{rows\_with\_compressed\_values}_{col})(\text{field\_size}_{col})$$

where:

This term ...	Specifies the number of ...
N	compressed columns in the table.
AMPs	AMPs in the system.
$field\_size_{col}$*	the number of table header bytes used by the compressed column + 1.
cardinality	rows in the table.
additional_presence_octets	presence octets in the table determined by "Equation 9: Additional Presence Octets" on page 14-54.
rows_with_compressed_values	rows in the table having compressed values.
$field\_size_{col}$	bytes in the compressed column.

Example Calculation

With the information provided by equations 1, 2, 3, and 4, you are now able to calculate the net capacity usage for a table. This example is for single-valued compression. Consider the following table demographics.

Characteristic	Description
Cardinality	1×10^9 rows
Candidate column	• Base field size is 8 bytes (`CHARACTER(8)`) • Calculated field size is 10 bytes (see "field_sizecol" on page 14-56). • Field is nullable • 10% of rows have a common value for the column • 5% of rows are null for the column
Number of AMPs in system	2 000
Additional presence octets	0 Only one column is compressed.
Number of rows having compressed field	$\text{Compressible rows} = ((0.10)(1 \times 10^9)) + ((0.05)(1 \times 10^9)) = 1.5 \times 10^8$

The value for net capacity usage is calculated as follows. Note that because there is only 1 presence octet, the row_header_space term evaluates to 0, so it has no effect in the computation.

$$\text{Net capacity usage} = ((2000 \text{ AMPs})(10 \text{ bytes})) - ((1.5 \times 10^8)(8 \text{ bytes})) = -1\,199\,980\,000 \text{ bytes}$$

The highly negative result indicates that the column is an excellent candidate for compression.

Expanded Examples

The following set of examples studies the relative benefits of compression for different table and system sizes. The calculations evaluate compression as if presence bits were added individually. You can also use this methodology to evaluate the outcome of compressing multiple columns within a table.

Example 1

This example examines a particular case for a 10 AMP system with 1 compressible value. Fallback is enabled for this table. The specifications for the study are as follows.

Variable	Condition
Number of AMPs in system	10
Column nullability	Nullable
Null compression	Compressed
Cardinality	1.0×10^9
Number of rows having compressible values	1.0×10^8
Capacity utilization if column is not compressed (bytes)	20 250 000 000
Number of non-null values to compress	1
Field size (bytes)	10
Fallback	Yes
1 in *n* rows is compressible, where *n* is this value	10.0
Percentage of rows compressible in table	10
Presence bits for value compression	1
Presence bits for null compression	1

The following table indicates the net space savings recovered through compression.

	Bytes Recovered by Compression Expressed in Different Units of Magnitude			
	Bytes	**Megabytes**	**Gigabytes**	**Terabytes**
Additional Table Header Space	120	0	0.0	0.00
Additional Row Header Space	250 000 000	250	0.3	0.00
Savings from Row Compression	2 000 000 000	2 000	2.0	0.00
	==========	==========	==========	==========
Net Savings from Compression	1 749 999 880	1 750	1.7	0.00

Capacity utilization if column not compressed	20 250 000 000
Capacity utilization if column is compressed	18 500 000 120
Percent compression	9.00
Compression ratio	1.09
Break-Even Percentage of Compressible Rows	1.25
To Break Even, 1 in *n* Rows Must Be Compressible, Where *n* Is This Value	80.00

Example 2

This example examines a case that is identical to the one presented in "Example 1" on page 14-57 except that it compresses 100 values. Fallback is enabled for this table. The specifications for the study are as follows:

Variable	Condition
Number of AMPs in system	10
Column nullability	Nullable
Null compression	Compressed
Cardinality	1.0×10^9
Number of rows having compressible values	1.0×10^8
Capacity utilization if column is not compressed (bytes)	20 250 000 000
Number of non-null values to compress	100
Field size (bytes)	10
Fallback	Yes
1 in *n* rows is compressible, where *n* is this value	10.0
Percentage of rows compressible in table	10
Presence bits for value compression	7
Presence bits for null compression	1

The following table indicates the net space savings recovered through compression. Notice the relative net savings *loss* for this case when 100 values are compressed rather than 1 value.

	Bytes Recovered by Compression Expressed in Different Units of Magnitude			
	Bytes	**Megabytes**	**Gigabytes**	**Terabytes**
Additional Table Header Space	10 010	0	0.00	0.00
Additional Row Header Space	1 750 000 000	1 750	1.8	0.00
Savings from Row Compression	2 000 000 000	2 000	2.0	0.00
	==========	==========	==========	==========
Net Savings from Compression	249 989 990	250	0.2	0.00

Capacity utilization if column not compressed	20 250 000 000
Capacity utilization if column is compressed	20 000 010 010
Percent compression	1
Compression ratio	1.01
Break-Even Percentage of Compressible Rows	8.75
To Break Even, 1 in *n* Rows Must Be Compressible, Where *n* Is This Value	11.4

Example 3

This example examines a case that is identical to the one presented in "Example 2" on page 14-59 except that it compresses 200 values rather than 100 values. Fallback is enabled for this table. The specifications for the study are as follows.

Variable	Condition
Number of AMPs in system	10
Column nullability	Nullable
Null compression	Compressed
Number of non-null values to compress	200
Cardinality	1.0×10^9
Number of rows having compressible values	1.0×10^8
Capacity utilization if column is not compressed (bytes)	20 250 000 000
Number of non-null values to compress	200
Field size (bytes)	10
Fallback	Yes
1 in *n* rows is compressible, where *n* is this value	10.0
Percentage of rows compressible in table	10
Presence bits for value compression	8
Presence bits for null compression	1

The following table indicates the net space savings recovered through compression. Notice the absolute net savings *loss* for this case when 200 values are compressed rather than 100 values. Compression of this column actually increases the space consumed rather than decreasing it.

	Bytes Recovered by Compression Expressed in Different Units of Magnitude			
	Bytes	**Megabytes**	**Gigabytes**	**Terabytes**
Additional Table Header Space	20 010	0	0.0	0.00
Additional Row Header Space	2 000 000 000	175	0.2	0.00
Savings from Row Compression	2 000 000 000	2 000	2.0	0.00
	==========	==========	==========	==========
Net Savings from Compression	-20 010	-2 000	-2.0	0.00

Capacity utilization if column not compressed	20 250 000 000
Capacity utilization if column is compressed	20 250 020 010
Percent compression	0
Compression ratio	1.00
Break-Even Percentage of Compressible Rows	10.00
To Break Even, 1 in *n* Rows Must Be Compressible, Where *n* Is This Value	10.00

Example 4

This example examines a case that is identical to the one presented in "Example 3" on page 14-61 except that fallback is not enabled for this table. The specifications for the study are as follows.

Variable	Condition
Number of AMPs in system	10
Column nullability	Nullable
Null compression	Compressed
Number of non-null values to compress	200
Cardinality	1.0×10^9
Number of rows having compressible values	1.0×10^8
Capacity utilization if column is not compressed (bytes)	10 125 000 000
Number of non-null values to compress	1
Field size (bytes)	10
Fallback	No
1 in *n* rows is compressible, where *n* is this value	10.0
Percentage of rows compressible in table	10
Presence bits for value compression	8
Presence bits for null compression	1

The following table indicates the net space savings recovered through compression. Notice the absolute net savings *loss* for this case when 200 values are compressed rather than 100 values. Compression of this column actually increases the space consumed rather than decreasing it.

	Bytes Recovered by Compression Expressed in Different Units of Magnitude			
	Bytes	**Megabytes**	**Gigabytes**	**Terabytes**
Additional Table Header Space	20 010	0	0.0	0.00
Additional Row Header Space	1 000 000 000	175	0.2	0.00
Savings from Row Compression	1 000 000 000	2 000	2.0	0.00
	==========	==========	==========	==========
Net Savings from Compression	-20 010	-2 000	-2.0	0.00

Capacity utilization if column not compressed	10 125 000 000
Capacity utilization if column is compressed	10 120 020 010
Percent compression	0
Compression ratio	1.00
Break-Even Percentage of Compressible Rows	10.00
To Break Even, 1 in *n* Rows Must Be Compressible, Where *n* Is This Value	10.00

Example 5

This example examines a case that is identical to the one presented in "Example 3" on page 14-61 except that the field width of the compressed column is 40 rather than 10 bytes. Fallback is enabled for this table. The specifications for the study are as follows.

Variable	Condition
Number of AMPs in system	10
Column nullability	Nullable
Null compression	Compressed
Cardinality	1.0×10^9
Number of rows having compressible values	1.0×10^8
Capacity utilization if column is not compressed (bytes)	80 250 000 000
Number of non-null values to compress	200
Field size (bytes)	40
Fallback	Yes
1 in *n* rows is compressible, where *n* is this value	10.0
Percentage of rows compressible in table	10
Presence bits for value compression	8
Presence bits for null compression	1

The following table indicates the net space savings recovered through compression. Notice the net savings *gain* for this case when the number of bytes in the compressed column is increased to 40 from 10. This not only demonstrates the obvious fact that greater space savings can be extracted by compressing wider column values, but that there is a crossover point between where the column width is such that a net savings cannot be realized by compression and where the column width is such that a net savings *can* be realized by compression.

	Bytes Recovered by Compression Expressed in Different Units of Magnitude			
	Bytes	**Megabytes**	**Gigabytes**	**Terabytes**
Additional Table Header Space	80 010	0	0.0	0.00
Additional Row Header Space	2 000 000 000	2 000	2.0	0.00
Savings from Row Compression	8 000 000 000	8 000	8.0	0.01
	==========	==========	==========	==========
Net Savings from Compression	5 999 919 990	6 000	6.0	0.01

Capacity utilization if column not compressed	80 250 000 000
Capacity utilization if column is compressed	74 250 080 010
Percent compression	7
Compression ratio	1.08
Break-Even Percentage of Compressible Rows	2.50
To Break Even, 1 in *n* Rows Must Be Compressible, Where *n* Is This Value	40.00

Example 6

This example examines a case that is identical to the one presented in "Example 3" on page 14-61 except that the percentage of compressible rows for the compressed column is 30 rather than 10. Fallback is enabled for this table. The specifications for the study are as follows.

Variable	Condition
Number of AMPs in system	10
Column nullability	Nullable
Null compression	Compressed
Cardinality	1.0×10^9
Number of rows having compressible values	1.0×10^8
Capacity utilization if column is not compressed (bytes)	20 250 000 000
Number of non-null values to compress	200
Field size (bytes)	10
Fallback	Yes
1 in *n* rows is compressible, where *n* is this value	10.0
Percentage of rows compressible in table	30
Presence bits for value compression	8
Presence bits for null compression	1

The following table indicates the net space savings recovered through compression. Notice the net savings *gain* for this case when the percentage of compressible rows in the compressed column is increased to 30 from 10.

	Bytes Recovered by Compression Expressed in Different Units of Magnitude			
	Bytes	**Megabytes**	**Gigabytes**	**Terabytes**
Additional Table Header Space	20 010	0	0.0	0.00
Additional Row Header Space	2 000 000 000	2 000	2.0	0.00
Savings from Row Compression	6 000 000 000	6 000	6.0	0.01
	==========	==========	==========	==========
Net Savings from Compression	3 999 979 990	4 000	4.0	0.00

Capacity utilization if column not compressed	20 250 000 000
Capacity utilization if column is compressed	16 250 020 010
Percent compression	20
Compression ratio	1.25
Break-Even Percentage of Compressible Rows	10.00
To Break Even, 1 in *n* Rows Must Be Compressible, Where *n* Is This Value	10.00

Example 7

This example examines a particular case for an 120 AMP system and 200 compressed values. Fallback is enabled for this table. The specifications for the study are as follows.

Variable	Condition
Number of AMPs in system	120
Column nullability	Nullable
Null compression	Compressed
Cardinality	1.0×10^9
Number of rows having compressible values	2.0×10^8
Capacity utilization if column is not compressed (bytes)	100 250 000 000
Number of non-null values to compress	200
Field size (bytes)	50
Fallback	Yes
1 in *n* rows is compressible, where *n* is this value	5.0
Percentage of rows compressible in table	20
Presence bits for value compression	8
Presence bits for null compression	1

The following table indicates the net space savings recovered through compression.

	Bytes Recovered by Compression Expressed in Different Units of Magnitude			
	Bytes	**Megabytes**	**Gigabytes**	**Terabytes**
Additional Table Header Space	942 120	1	0.0	0.00
Additional Row Header Space	2 000 000 000	2 000	2.0	0.00
Savings from Row Compression	20 000 000 000	20 000	20.0	0.02
	==========	==========	==========	==========
Net Savings from Compression	17 999 057 880	17 999	18.0	0.02

Capacity utilization if column not compressed	100 250 000 000
Capacity utilization if column is compressed	82 250 942 120
Percent compression	18
Compression ratio	1.22
Break-Even Percentage of Compressible Rows	2.00
To Break Even, 1 in *n* Rows Must Be Compressible, Where *n* Is This Value	50.00

Example 8

This example examines a particular case for an 120 AMP system. Fallback is enabled for this table. The specifications for the study are as follows.

Variable	Condition
Number of AMPs in system	120
Column nullability	Nullable
Null compression	Compressed
Cardinality	1.0×10^9
Number of rows having compressible values	2.0×10^8
Capacity utilization if column is not compressed (bytes)	100 250 000 000
Number of non-null values to compress	200
Field size (bytes)	50
Fallback	Yes
1 in *n* rows is compressible, where *n* is this value	50.0
Percentage of rows compressible in table	20
Presence bits for value compression	8
Presence bits for null compression	1

The following table indicates the net space savings recovered through compression.

	Bytes Recovered by Compression Expressed in Different Units of Magnitude			
	Bytes	**Megabytes**	**Gigabytes**	**Terabytes**
Additional Table Header Space	1 200 120	1	0.0	0.00
Additional Row Header Space	2 000 000 000	2 000	2.0	0.00
Savings from Row Compression	20 000 000 000	20 000	20.0	0.02
	==========	==========	==========	==========
Net Savings from Compression	17 998 799 880	17 999	18.0	0.02

Capacity utilization if column not compressed	100 250 000 000
Capacity utilization if column is compressed	82 251 200 120
Percent compression	18
Compression ratio	1.22
Break-Even Percentage of Compressible Rows	2.00
To Break Even, 1 in *n* Rows Must Be Compressible, Where *n* Is This Value	50.00

Example 9

This example examines a particular case for an 160 AMP system. Fallback is not enabled for this table. The specifications for the study are as follows.

Variable	Condition
Number of AMPs in system	160
Column nullability	Nullable
Null compression	Compressed
Cardinality	1.0×10^9
Number of rows having compressible values	3.0×10^7
Capacity utilization if column is not compressed (bytes)	4 125 000 000
Number of non-null values to compress	200
Field size (bytes)	4
Fallback	No
1 in *n* rows is compressible, where *n* is this value	33.3
Percentage of rows compressible in table	3
Presence bits for value compression	8
Presence bits for null compression	1

The following table indicates that no net space savings can be recovered through compression. The net "savings" from compression is actually a loss of 880 128 160 bytes.

	Bytes Recovered by Compression Expressed in Different Units of Magnitude			
	Bytes	**Megabytes**	**Gigabytes**	**Terabytes**
Additional Table Header Space	128 160	0	0.0	0.00
Additional Row Header Space	1 000 000 000	1 000	1.0	0.00
Savings from Row Compression	120 000 000	120	0.1	0.00
	==========	==========	==========	==========
Net Savings from Compression	- 880 128 160	- 880	- 0.9	- 0.00

Capacity utilization if column not compressed	4 125 000 000
Capacity utilization if column is compressed	5 005 128 160
Percent compression	None
Compression ratio	0.82
Percent capacity reduction for this column	None
Break-Even Percentage of Compressible Rows	25.00
To Break Even, 1 in *n* Rows Must Be Compressible, Where *n* Is This Value	4.00

Data Type Considerations

Introduction

Different column data types occupy different amounts of disk space. This topic examines the various Teradata data types and indicates their absolute sizes.

The data types for the primary index column set also have an important effect on how rows hash. For example, a primary index value typed DECIMAL generally hashes to a different AMP than the same primary index value typed INTEGER (see "Hashing and Data Types" on page 8-50 for additional information).

The information presented here is intended to be used for sizing purposes only. For usage information about the various data types, see *Teradata RDBMS SQL Reference, Volume 3*.

Integer Types

BYTEINT: 1 Byte

Bit 0	Bits 1- 7
Sign	-128 — +127

SMALLINT: 2 Bytes

Bit 0	Bits 1 - 15
Sign	-32 768 — +32 767

INTEGER: 4 Bytes

Bit 0	Bits 1 - 31
Sign	-2 147 483 648 — +2 147 483 647

Other Numeric Types

DECIMAL/NUMERIC [(n[,m])]: 1, 2, 4, or 8 Bytes

1 - 2 digits	Not used						
3 - 4 digits		Not used					
5 - 9 digits				Not used			
10 - 18 digits							
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8

FLOAT/REAL/DOUBLE PRECISION: 8 Bytes

Bit 0	Bits 1 - 10	Bits 11 - 63
	Exponent	Mantissa
Sign	$2 * 10^{-307}$ — $2 * 10^{308}$	

Byte Types

BYTE(n): 1 - 64 000 Bytes Fixed

Byte 1	...	Byte 64 000
1 — 64 000		

VARBYTE(n): 1 - 64 000 Bytes Variable

Byte 1	...	Byte 64 000
1 — 64 000		

DateTime Types

DATE: 4 Bytes

Year	Month	Day
AD January 1, 1 — AD December 31, 9999		

TIME: 6 Bytes		
Hour	**Minute**	**Seconds**
hh:mm:ss[.ssssss]		

TIME WITH TIME ZONE: 8 Bytes				
Hour	**Minute**	**Second**	**Timezone_Hour**	**Timezone_Minute**
hh:mm:ss.ssssss±hh:mm				

TIMESTAMP WITH TIME ZONE: 12 Bytes							
Year	**Month**	**Day**	**Hour**	**Minute**	**Second**	**Timezone_Hour**	**Timezone_Minute**
yyyy-mm-dd hh:mm:ss±hh:mm							

Interval Types

INTERVAL YEAR: 2 Bytes			
Year Precision			
1	**2**	**3**	**4**
Range			
-9 — 9			
-99 — 99 (default)			
-999 — 999			
-9999 — 9999			

INTERVAL YEAR TO MONTH: 4 Bytes

Year Precision				Month
1	2	3	4	
Range				
-9 — 9				
-99 — 99 (default)				
-999 — 999				
-9999 — 9999				00 — 11

INTERVAL MONTH: 2 Bytes

Month Precision			
1	2	3	4
Range			
-9 — 9			
-99 — 99 (default)			
-999 — 999			
-9999 — 9999			

INTERVAL DAY: 2 Bytes

Day Precision			
1	2	3	4
Range			
-9 — 9			
-99 — 99 (default)			
-999 — 999			
-9999 — 9999			

INTERVAL DAY TO HOUR: 4 Bytes

Day Precision				Hour
1	2	3	4	
Range				
-9 — 9				
-99 — 99 (default)				
-999 — 999				
-9999 — 9999				00 — 23

INTERVAL DAY TO MINUTE: 8 Bytes (including 2 pad bytes)

Day Precision				Minute
1	2	3	4	
Range				
-9 — 9				00 — 59
-99 — 99 (default)				
-999 — 999				
-9999 — 9999				

INTERVAL DAY TO SECOND: 10 Bytes

Day Precision				Seconds Precision					
1	2	3	4	1	2	3	4	5	6
Range									
-9 — 9				00.0 — 59.9					
-99 — 99 (default)				00.00 — 59.99					
-999 — 999				00.000 — 59.999					
-9999 — 9999				00.0000 — 59.9999					
				00.00000 — 59.99999					
				00.000000 — 59.999999 (default)					

INTERVAL HOUR: 2 Bytes

Hour Precision			
1	**2**	**3**	**4**
Range			
-9 — 9			
-99 — 99 (default)			
-999 — 999			
-9999 — 9999			

INTERVAL HOUR TO MINUTE: 4 Bytes

Hour Precision				Minute
1	**2**	**3**	**4**	
Range				
-9 — 9				
-99 — 99 (default)				
-999 — 999				
-9999 — 9999				00 — 59

INTERVAL HOUR TO SECOND: 8 Bytes

Hour Precision				Seconds Precision					
1	**2**	**3**	**4**	**1**	**2**	**3**	**4**	**5**	**6**
Range									
-9 — 9				00.0 — 59.9					
-99 — 99 (default)				00.00 — 59.99					
-999 — 999				00.000 — 59.999					
-9999 — 9999				00.0000 — 59.9999					
				00.00000 — 59.99999					
				00.000000 — 59.999999 (default)					

INTERVAL MINUTE: 2 Bytes

Minutes Precision			
1	**2**	**3**	**4**
Range			
-9 — 9			
-99 — 99 (default)			
-999 — 999			
-9999 — 9999			

INTERVAL MINUTE TO SECOND: 6 Bytes

Minute Precision				Seconds Precision					
1	**2**	**3**	**4**	**1**	**2**	**3**	**4**	**5**	**6**
Range									
-9 — 9				0.0 — 59.9					
-99 — 99 (default)				0.00 — 59.99					
-999 — 999				0.000 — 59.999					
-9999 — 9999				0.0000 — 59.9999					
				0.00000 — 59.99999					
				0.000000 — 59.999999 (default)					

INTERVAL SECOND: 6 Bytes

Seconds Precision				Fractional Seconds Precision					
1	**2**	**3**	**4**	**1**	**2**	**3**	**4**	**5**	**6**
Range									
-9 — 9				0.0 — 0.9					
-99 — 99 (default)				0.00 — 0.99					
-999 — 999				0.000 — 0.999					

INTERVAL SECOND: 6 Bytes

Seconds Precision				Fractional Seconds Precision					
1	2	3	4	1	2	3	4	5	6
Range									
-9999 — 9999				0.0000 — 0.9999					
				0.00000 — 0.99999					
				0.000000 — 0.999999 (default)					

Character Types

CHARACTER(n): 1 - 64 000 Bytes Fixed

Byte 1	...	Byte 64 000
1 — 64 000		

VARCHAR(n): 1 - 64 000 Bytes Variable

Byte 1	Byte 2	...	Byte 64 000
Column Offset		1 — 64 000	

LONGVARCHAR(n): 64 000 Bytes Fixed (equivalent to VARCHAR(64000))

Byte 1	Byte 2	...	Byte 64 000
Column Offset		1 — 64 000	

Graphic Types

GRAPHIC(n): 1 - 32 000 Bytes Fixed

Byte 1	...	Byte 32 000
1 — 32 000		

VARGRAPHIC(n): 1 - 32 000 Bytes Variable

Byte 1	Byte 2	...	Byte 32 000
Column Offset		1 — 32 000	

LONGVARGRAPHIC(n): 32 000 Bytes Fixed (equivalent to VARGRAPHIC(32 000))

Byte 1	Byte 2	...	Byte 32 000
Column Offset		1 — 32 000	

Row Size Calculation

Introduction

This topic describes a procedure for calculating the physical size of a row for a typical table.

The process uses the row size form found on page 14-87.

Row Definition Used for This Exercise

The following Employee table row definition is used for this exercise.

Employee

Employee Number	Supervisor Employee Number	Department Number	Job Code	Last Name	First Name	Hire Date	Birth Date	Salary Amount
PK, SA	FK	FK	FK	NN	NN	NN	NN	NN
INTEGER	INTEGER	INTEGER	SMALLINT	CHAR(20)	VARCHAR(30)	DATE	DATE	DECIMAL(10,2)

Procedure

Follow this procedure to determine the physical size of a typical row for any given table.

Step	Action
1	List all the varying length columns in the four columns labeled Variable Data Detail.
2	For each variable length column, estimate the average number of bytes expected.
3	Add the total number of bytes in varying length columns and record the figure in the field labeled SUM(a). In the example table, there is only one varying length column, First Name, which is typed VARCHAR(30). Its average number of bytes is estimated to be 14, so the value for SUM(a) is 14.

Step	Action
4	Determine how many columns in the table there are for each fixed byte data type. The example table has the following number of each fixed byte data type. • 1 SMALLINT • 3 INTEGER • 2 DATE • 1 DECIMAL
5	Enter these figures in the Number of Columns field next to each relevant data type.
6	Multiply the counts by the sizing factor provided for the type and enter the results under the Total field.
7	Enter the total byte counts for the fixed length character types in the SUM(n) field. • CHARACTER • BYTE • GRAPHIC For the example, only one CHARACTER column with a byte count of 20 is found, so enter 20 as the result for SUM(n).
8	Add the values for SUM(n) and SUM(a) and record them in the field labeled Logical Size. For the example, the logical size is 64 bytes.
9	The overhead for any row is 14 bytes. That number is prerecorded for you in the field labeled Overhead.
10	Multiply the number of variable length columns by 2 to account for the 2-byte variable column offset pointers determined in Step 1. Write the value in the field labeled Variable Column Offsets. For the example, there is only one variable column, so write the number 2 here (1 x 2 = 2).
11	Record the number of columns compressed on a non-null value.
12	Record the number of nullable columns.

Step	Action
13	Divide the sum of steps 11 and 12 by 8 and record the quotient of the operation. The purpose of this step is to account for any required additional presence bits that might be required. For the example, the calculation is 3/8, so the quotient is 0.
14	Add the integers recorded in the Total column and record the sum in the field labeled Physical Size. If the sum is an uneven number, round it up to the next even number. For the example, the sum is 82, so no rounding is necessary.

Row Size Form

Row Size Calculation Form

Table Name:

Variable Data Detail

Column Name	Type	Max	Avg
		SUM(a)	

Data Type	Number of Columns	Sizing Factor	Total
Byteint	*	1 =	
Smallint	*	2 =	
Integer	*	4 =	
Date	*	4 =	
Time(ANSI)	*	6 =	
Time With Time Zone	*	8 =	
Timestamp	*	10 =	
Timestamp With Time Zone	*	12 =	
Interval Year	*	2 =	
Interval Year to Month	*	4 =	
Interval Month	*	2 =	
Interval Month to Day	*	2 =	
Interval Day	*	2 =	
Interval Day to Hour	*	4 =	
Interval Day to Minute	*	8 =	
Interval Day to Second	*	10 =	
Interval Hour	*	2 =	
Interval Hour to Minute	*	4 =	
Interval Hour to Second	*	8 =	
Interval Minute	*	2 =	
Interval Minute to Second	*	6 =	
Interval Second	*	6 =	
Dec 1-2	*	1 =	
3-4	*	2 =	
5-9	*	4 =	
10-18	*	8 =	
Float	*	8 =	
Fixed		SUM(n) =	
Variable		SUM(a) =	
		Logical Size =	
		Overhead =	14
Variable Column Offsets		=	
______ Compress Columns			
______ Nullable Columns			
______ /8 (Quotient only)		=	
		Physical Size** =	

SUM(a)=SUM of the AVERAGE number of bytes expected for the variable column.
SUM(n)=SUM of the CHAR and GRAPHIC column bytes.
**Round up to an even number of bytes

FF07D347

Sizing Base Tables and Index Subtables

Teradata data block sizes are variable. This permits a designer the flexibility of designing tables without having to spend time balancing data types, row sizes, and block sizes in order to optimize their storage.

The critical variables for sizing Teradata tables and indexes are the following.

- Row size
- Projected cardinality
- Presence of fallback

Table headers are a fixed variable, but must be considered when you are evaluating the projected sizes of your tables. Ensure that you take compression into account when sizing table headers (see "Value Compression" on page 14-20).

Sizing Base Tables, Hash Indexes, and Join Indexes

Introduction

Estimate the size of your base tables and hash and join indexes using the equations provided in "Table Sizing Equations" on page 14-89.

These equations assume a constant block size. Because block sizes can vary widely within a table, you can obtain more finely tuned estimates by computing values over a range of block sizes.

Calculating a Typical Block Size

Depending on how you define your tables and the boundary conditions on their rows, the maximum size in bytes for a typical block can be estimated by the following equation.

$$\text{Typical block size} = \frac{\text{MAX}}{2} + 512 \mid 1024$$

where:

This variable ...	Specifies ...
MAX	the maximum block size in bytes. The upper limit on this value is 255 sectors, or 130 560 Kbytes.
512 | 1024	the number of bytes per sector for the table. Depending on your configuration, this number is either 512 or 1024 bytes, as signified by 512 | 1024, where the symbol | indicates the logical OR.

Table Sizing Equations

The following parameter definitions are used with this equation set.

Parameter	Definition
Block Overhead	Block Header + Block Trailer = 16 bytes.
Minimum Header Size	512 bytes.
Typical Block Size	Number of bytes per block (see "Calculating a Typical Block Size" on page 14-89).
NumAmps	Number of AMPs in the configuration.

Parameter	Definition
RowCount	Projected cardinality for the table.
Average RowSize	Physical row size for the table (see "Row Size Calculation" on page 14-84).

$$\text{Rows per block (rounded down)} = \frac{\text{Typical block size} - 16}{\text{Row size}}$$

$$\text{Number of blocks (rounded up)} = \frac{\text{Row count}}{\text{Rows per block}}$$

$$\text{Number of header bytes} = (\text{Number of AMPs})(512)$$

$$\text{Number of base table bytes (without fallback)} = (\text{Number of blocks} \times \text{Typical block size}) + \text{Number of header bytes}$$

$$\text{Number of base table bytes (with fallback)} = 2(\text{Number of blocks})(\text{Typical block size}) - \text{Number of header bytes}$$

Example

You have collected the following information about a table, hash, or join index.

Parameter	Value
Block size	5 120 bytes **Note:** Typical block size used.
Number of AMPs	20
Row count	1 000 000
Row size	100 bytes

Calculate the size of this table, hash, or join index when it is defined with fallback.

$$\text{Number of rows per block} = \frac{5120 - 16}{100} = 51$$

$$\text{Number of blocks} = \frac{1\,000\,000}{51} = 19\,608$$

$$\text{Number of header bytes} = 20 \times 512 = 10\,240$$

$$\text{Number of bytes in base table with fallback} = ((19\,608 \times 5120) \times 2) + 10\,240$$

Note that values are rounded in the following manner.

Parameter	Rounding Direction
Rows per block	Down
Number of blocks	Up

Sizing a Unique Secondary Index Subtable

Introduction

Estimate the size of your unique secondary indexes using the equation provided in "USI Sizing Equation."

Because the index is unique, there is one USI subtable row for each row in the base table, so the row counts are identical.

USI Sizing Equation

The following parameter definitions are used with this equation.

Parameter	Definition
Index Value Size	Size of the column set on which the USI is defined.
Block Overhead	Sum of the following factors. • Block headers and trailers • Row headers and trailers • USI rowID • Spare byte • Presence octets • Base table rowID = 25 bytes

$$\text{USI Subtable Size} = ((\text{Row Count}) \times (\text{Index Value Size} + 25))$$

If fallback is defined for the base table, then double the calculated result.

Teradata implicitly creates a unique secondary index on any column set specified as PRIMARY KEY or UNIQUE, so you must take these indexes into consideration for your capacity planning as well.

Sizing a Nonunique Secondary Index Subtable

Introduction

Estimate the size of your nonunique secondary indexes using the equation provided in "NUSI Sizing Equation."

Special Considerations for NUSI Size Estimates

The number of AMPs in the configuration are an important factor in estimating the total size of any NUSIs defined on a base table, as summarized in the following table.

IF the number of AMPS is ...	THEN at least ...	AND the result is that ...
less than the number of rows per value	one row from each NUSI value is probably distributed to each AMP.	• Every AMP has every value. • Every AMP has a subtable row for every value.
greater than the number of rows per value	some AMPs are missing some NUSI values.	• Not every AMP has every value. • Not every AMP has a subtable row for every value.

NUSI Sizing Equation

The following parameter definitions are used with this equation.

Parameter	Definition
Cardinality * 8	Each base table rowID is stored in a NUSI subtable. Each rowID is 8 bytes long.
NumDistinct	The value is an estimate of the number of distinct NUSI subtable values and is based on each NUSI subtable having at least one index row per AMP for each distinct index value of a base table row stored on that AMP.
IndexValueSize	The number of index data bytes.
NUSI Block Overhead	Sum of the following factors. • Block headers and trailers • Row headers and trailers • NUSI row rowID • Spare byte • Presence octets = 17 bytes

Parameter	Definition
MIN (NumAmps | Rows per value)	The lesser of the two parameters.

$$\text{NUSI subtable size} = 8(\text{Cardinality}) + ((\text{NumDistinct}) \times (\text{IndexValueSize} + 17) \times \text{MIN}(\text{NumAMPs} \mid \text{Rows per value}))$$

If fallback is defined for the base table, then double the calculated result.

NUSI Space Considerations

The PERM space required during the creation of a NUSI might temporarily be much greater than the space occupied by the finished index as described by the following table.

AMP cluster size, which determines the relative size of the backup subtables, is a an important factor. For information about AMP clusters, see *Introduction to Teradata RDBMS*. A typical AMP cluster size is 4 AMPs, but the valid range varies from 2 to 16 AMPs per cluster.

NUSIs are built one subtable at a time. Spool space is not used when you create a NUSI.

<table>
<tr><th>IF you create this sort of table ...</th><th>THEN the following peak temporary PERM space usage factors apply ...</th></tr>
<tr><td>non-fallback or fallback with small AMP cluster size</td><td>Estimate the temporary PERM space required when building a NUSI using the following worst case estimation equation.
$\text{TemporarySpaceNF} = \text{Cardinality} \times (\text{LengthOfKey} + 30)$
where:
<table><tr><th>This variable ...</th><th>Specifies the ...</th></tr><tr><td>TemporarySpaceNF</td><td>size of the temporary PERM space required without fallback.</td></tr><tr><td>Cardinality</td><td>number of rows in the base table.</td></tr><tr><td>LengthOfKey</td><td>combined byte length of the key.</td></tr></table></td></tr>
<tr><td>fallback</td><td>Estimate the temporary PERM space required, assuming typical AMP cluster size, using the following equation.
$\text{TemporarySpaceFB} = \text{NUSISubtableSize} + \text{TemporarySpaceNF}$
This is a very conservative space estimate. For typical AMP cluster sizes, peak usage exceeds the prediction made by the model.</td></tr>
</table>

Sizing a Reference Index Subtable

Introduction

A reference index is an internal structure that the system creates whenever a referential integrity constraint is defined between tables using a PRIMARY KEY or UNIQUE constraint on the parent table in the relationship and a REFERENCES constraint on a foreign key in the child table.

The index row contains a count of the number of references in the child, or foreign key, table to the PRIMARY KEY or UNIQUE constraint in the parent table.

Estimate the size of your reference indexes using the equation provided in "Reference Index Sizing Equation."

Reference Index Sizing Equation

The following parameter definitions are used with this equation.

<table>
<tr><th>Parameter</th><th>Definition</th></tr>
<tr><td>Row Count * 4</td><td>A count of the number of foreign key row references is stored in an RI subtable. Each foreign key row count is 4 bytes long.</td></tr>
<tr><td>FKLength</td><td>The length of a fixed length foreign key value in bytes.
Use one of these parameters depending on the RI in question.
<table><tr><th>IF the FK field value is ...</th><th>THEN use this parameter ...</th></tr><tr><td>fixed</td><td>length of the value</td></tr><tr><td>variable</td><td>average length of the variable length FK values</td></tr></table></td></tr>
<tr><td>NumDistinct</td><td>The value is an estimate of the number of distinct foreign key subtable values.
Exclude null foreign keys from this estimate.</td></tr>
<tr><td>Presence Overhead</td><td>Use the following equation to calculate this parameter.
$$\text{Overhead} = \frac{1 + \text{Number nullable FK fields}}{8}$$
If there are no presence bits, then the value for this parameter is 0.</td></tr>
</table>

Parameter	Definition
VarLength Overhead	Use the following equation to calculate this parameter. Overhead = 2 × (Number variable length FK fields + 1) If there are no variable length foreign keys, then the value for this parameter is 0.
RI Block Overhead	Sum of the following factors. • Row length • RI row rowID • Spare byte • Presence octets • Offsets • Valid flag • Foreign key count • Reference array = 25 bytes

RI Subtable Size = (NumDistinct) × (FKLength + PresenceOverhead + VarLengthOverhead+ 25)

If fallback is defined for the child table in the relationship, then double the calculated result.

Sizing Spool Space

Introduction

Spool, or temporary space, is frequently overlooked in capacity planning, yet it is critical to the operations of the database. Spool space needs vary from table to table, application to application, and with frequency of use. Very large systems use even more spool than smaller systems.

Spool falls into two categories: intermediate and output.

Intermediate Spool Space

Intermediate spool results are retained until no longer needed. You can determine when intermediate spool is flushed by examining the output of an EXPLAIN. The first step performed after intermediate spool has been flushed is designated “Last Use.”

Output Spool Space

Output spool results are the final information returned for a query or the rows updated within, inserted into, or deleted from a base table. The length of time output spool is retained depends on the subsystem and various system conditions, as described in the following table.

Subsystem/Condition	When Output Spool Is Released
BTEQ	Last spool response.
Embedded SQL	The open cursor is closed.
CLIv2	• ERQ received • Function terminated
Session terminates asynchronously due to any number of conditions, including the following. • Job abort • Timeout • Logoff	At the time the termination occurs.
System restart	At the time the restart occurs.

Source of Spool Space

Spool space is taken only from disk cylinders that are not being used for data. Data blocks and spool blocks cannot coexist on the same cylinder.

When spool is released, the file system returns the cylinders it was using to the free cylinder list.

Spool Limits

The amount of spool space allocated to each user and database is assigned at CREATE USER or CREATE DATABASE time.

If you find that queries do not run because they run out of spool space, then increase the spool assignment for the user or database having the problem using the MODIFY USER or MODIFY DATABASE statements, respectively (see *Teradata RDBMS SQL Reference, Volume 4* for the syntax and usage notes for these statements).

Guidelines for Allocating Spool Space

Unless you reserve a pool of spool space, available space tends to disappear quickly. When applications consume all the spool space allocated for a system, processing halts.

The following method is used as a guideline for allocating spool space by a large Teradata customer in the retail business.

Step	Action
1	Create a special database to act as a spool space reservoir. Allocate 2 percent of the total user space in the system for this database.
2	Assign roughly 0.25 percent of the total space to each user as an upper limit, ensuring that each receives at least as much space as the size of the largest table they access concurrently.

Consider the following factors to perform finer tuning of user spool allotment.

- Query size

  The smaller the query, the less spool space required. If a particular user only performs small queries, then allocate less spool to that user.

  If a user performs many large queries, then allocate more spool to that user. With the exception of runaway queries, allocating more spool to a user is never harmful as long as system resources are not wasted.
- Database size

  The more AMPs in the configuration, the more thinly spread the data, so the more spool required per AMP.
- Average spool use per user
- Query types

  Decision support queries generally require more spool than OLTP queries.
- Number of concurrent users
- Number of concurrent queries permitted for any one user

  Spool space is cumulative per user.

Sizing a Query Capture Database

Introduction

Query Capture Databases are used to analyze SQL queries for ways in which they can be better optimized. A QCD can occupy significant storage space, so it is important to understand how much capacity is required to perform query optimization analyses.

Capacity Planning for a Query Capture Database

The following physical limits constrain the QCD functionality within certain boundaries.

- Because QCD tables are ordinary Teradata relational tables, they are limited to a row length of 64 Kbytes.
- QCD draws its disk space from PERM space just like any other persistent Teradata relational tables.

You can estimate the magnitude of the raw data (without indexes) generated by a single INSERT EXPLAIN or DUMP EXPLAIN statement from the following equation:

$$\text{Raw data size} \cong (2n + m + (t \cdot d))$$

where:

Equation element ...	Specifies the ...
n	size in bytes of the EXPLAIN statement output for the same query. The approximation assumes that all captured data demographics can be accommodated within $2n$ bytes.
m	size in bytes of the SQL query text.
t	average size in bytes of the DDL text for the object.
d	number of tables and views in the query.

The approximation is slightly more complex when statistics are captured using INSERT EXPLAIN WITH STATISTICS:[7]

$$\text{Raw data size} \cong (2n + m + (t \cdot d)) + 5000c$$

where:

Equation element …	Specifies the …
n	size in bytes of the EXPLAIN statement output for the same query. The approximation assumes that all captured data demographics can be accommodated within 2*n* bytes.
m	size in bytes of the SQL query text.
t	average size in bytes of the DDL text for the object.
d	number of tables and views in the query.
c	number of columns involved in range and explicit equality conditions in the query WHERE clause. If you do not specify WITH STATISTICS when you perform INSERT EXPLAIN, the value for *c* is 0.

The coefficient 5000 is used as the average size of statistics based on the following information:

- The 101[8] interval histograms used to store the statistics occupy 4 900 bytes per column.
- 100 bytes are added to account for an upward bound on miscellaneous bytes per column.

$$\text{Storage overhead in bytes per column due to statistics} \cong 4900 + 100 \cong 5000$$

For More Information

See the following chapters in *Teradata RDBMS SQL Reference, Volume 2* for more information about query capture databases and query optimization analysis:

- "Query Capture Facility"
- "Database Foundations for the Teradata Index Wizard"
- "Target Level Emulation"

7 This equation applies only to INSERT EXPLAIN WITH STATISTICS. DUMP EXPLAIN does not capture statistics.

8 This count includes interval 0 in addition to intervals 1 through 100. Interval 0 contains column- or index-global statistics, while intervals 1 through 100 contain interval-specific column statistics. See the chapter "Query Optimization" in *Teradata RDBMS SQL Reference, Volume 2*, for further information.

How to Size Table Space Empirically

Introduction

The most accurate method for sizing table space for a new configuration is to create a table with a definition as similar as possible to the legacy table you want to port to Teradata. You cannot just extrapolate the space information from your legacy system and expect the result to accurately estimate your Teradata table space needs because commercial database vendors all store their data in different ways.

General Procedure

Use one of the load utilities to load the empty table with a representative sample of rows that constitutes a known percentage of the entire cardinality of the legacy table.

For the first iteration of testing, do not define any secondary indexes on the table. Analyze the space characteristics for this test table, add a secondary index, and repeat the task. Continue this process until you are reasonably certain that you have captured the characteristics of the base table and all the secondary indexes you initially want to define on it.

Extrapolate the total spool space required from an examination of the smaller temporary table. The scaled up data should provide a reliable picture of the initial space requirements for the table and its indexes.

Explicit Procedure

The following table provides a more explicit procedure.

Step	Action
1	Create a new base table on the Teradata server with a definition that matches the definition for the legacy table it is replacing as closely as possible. Do not define any secondary indexes on this base table at this stage of the process.
2	Use one of the Teradata load utilities to load the newly defined base table with a representative sample of rows from the legacy table. The loaded sample should not only reflect the demographics of the entire population of the legacy table, but also be a fairly precise percentage of the cardinality for that table.
3	Query the system view DBC.TableSize for the space occupied by the new base table. "Querying DBC.TableSize" on page 14-102 provides a representative SQL query for you to use for this purpose.

Step	Action
4	Record the number of bytes returned by the query.
5	Add a secondary index to the table.
6	Query the system view DBC.TableSize again for the space occupied by the new base table and the new secondary index.
7	Record the number of bytes returned by the query.
8	The arithmetic difference between the numbers you recorded in steps 4 and 7 is the size of the newly defined secondary index.
9	Iterate steps 5 and 6 until you have finished adding all the secondary indexes you anticipate defining for this table.
10	Repeat the procedure with another legacy table until you have estimated the initial size of your entire Teradata database.
11	You should also evaluate the sizes of any hash or join indexes (and any secondary indexes defined on join indexes) you anticipate using with the new database. Because hash and join index tables are structurally virtually identical to base tables, you can use the same procedure and the same queries documented by this procedure to determine their size and the size of their indexes.

Querying DBC.TableSize

Use the following query to evaluate the sizing of your legacy tables. Assume the database in which the table was created is named Employee_LargeTable and the name of the table is Employee.

```
SELECT SUM(CurrentPerm)
FROM DBC.TableSize
WHERE DatabaseName = 'Employee_LargeTable'
AND TableName = 'Employee';
```

The information returned might look something like this.

```
Sum(CurrentPerm)
      2,527,232
```

Table Sizing Summary

The following list reviews the principal points to remember about estimating the table space requirements for your new database.

- Accurate estimates require accurate base data. The minimum base data required for these estimates are the following.
  - Cardinalities (or anticipated cardinalities) of your tables
  - Row size estimates
- Estimate sizes for all the following database objects.
  - Base tables
  - Base table fallback tables (when defined)
  - Secondary indexes
  - Secondary index fallback tables (when defined)
  - Hash indexes
  - Hash index fallback tables (when defined)
  - Join indexes (including any secondary indexes defined on them)
  - Join index fallback tables (when defined)
  - Spool space
  - Stored procedures

Chapter 15:

System-Level Considerations

This chapter examines capacity planning for system and table space.

The majority of the material focuses on system space planning including disk space. Database sizing issues such as allocating permanent space and estimating database size requirements are also described.

Database Size Considerations

The size of relational databases stored by Teradata has a range of up to 2048 CPUs and 2048 GB of memory to support a 128-TB or larger database. The BYNET interconnect supports up to 512 nodes.

Database sizing considerations need to take the following issues into account:

- Contents of system disks
- Contents of data disks
- Data disk space allocation
- Determining usable data space

Contents of System Disks

Introduction

At least two system disks, Disk 0 and Disk 1, are cabled to a controller not associated with the disk arrays.

Disk 0 is the boot disk. The slices on Disk 0 contain file systems under the control of the operating systems root directory.

Disk 1 provides additional space for dumps and memory swapping.

Boot Disk Contents

The following table lists the contents of the boot disk, Disk 0.

Content	Use
AMP identifiers file	Stores identifiers of all AMPs
Configuration maps	Defines vprocs and ranks and the assignment of vdisks to AMPs
PDE and operating system	Stores software under which Teradata is running
Executable Teradata software	Includes any optional NCR CLIent software packages
Copy of Teradata GDOs	Includes DBSControl Record (DBSCONTROLGDO)
Values	Initialize hash buckets
Bad disk sectors file	Lists bad disk sectors
space	• Diagnostic reports • Memory swap space • Operating system and PDE dumps

Contents of Data Disks

Introduction

The virtual data disks controlled by the AMPs include reserved space and permanent space.

Reserved Space

Reserved space is used for the following:

- A file containing the vprocID of each AMP
- Optional memory swapping

For memory swapping information, see the *Teradata RDBMS Performance Optimization*.

Permanent Space

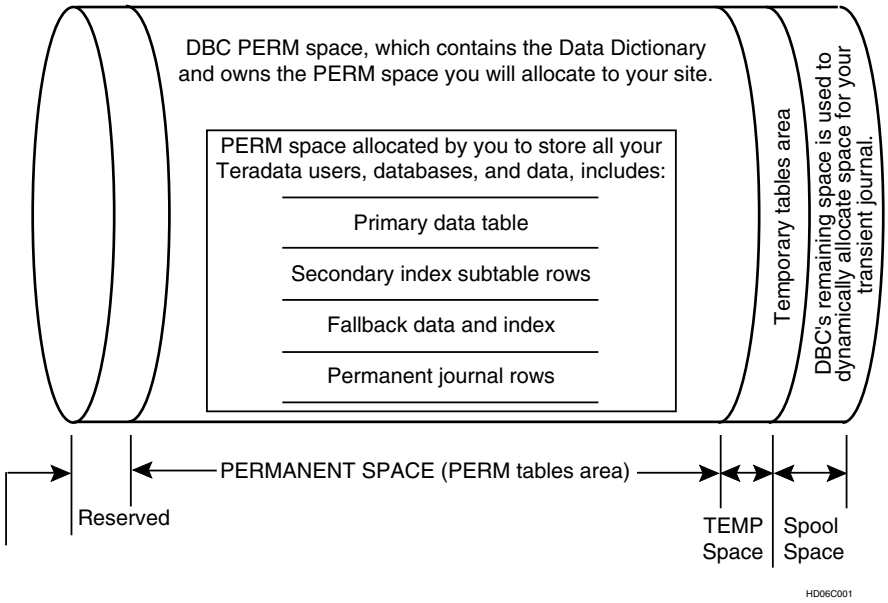
Permanent space, owned by a system user named DBC, is used for the following:

Content	Use
CRASHDUMPS database	Temporary storage of diagnostic information and crash dumps generated by the Teradata database and PDE.
SYSTEMFE user	Special macros and objects used by NCR field support personnel.
SYSADMIN user	Special views and macros used by NCR support personnel.
Data dictionary	Maintains Teradata user, database, table, index, trigger, permanent journal, view, stored procedure, and macro definitions.
Transient journal	Stores before-image copy of every row affected by all current sessions.
Spool files	Stores intermediate and response results of SQL queries.
Top-level hierarchical ownership of PERM and TEMP spaces	• Teradata databases • Base data tables • Join index tables • Hash index tables • Fallback data and join index tables • Secondary index subtables • Permanent journal tables • Stored procedures

Data Disk Space Allocation

The allocation of data disk space is reserved for system use, PERM space, or TEMP space owned by system user DBC, as illustrated in the graphic.

The size considerations for each type of space are described in "Permanent Space Allocations" on page 15-6.



Permanent Space Allocations

Introduction

Permanent data disk space includes database and user fixed PERM allocation. The Teradata database initialization routine creates three system users as follows:

- DBC (at the top of the hierarchy)
- SYSTEMFE (owned by DBC)
- SYSADMIN (owned by DBC)

All permanent space not consumed by SYSTEMFE and SYSADMIN is allocated to user DBC. The ramifications of the space allocation for each of the three system users are discussed in the subsections that follow.

System User DBC

The PERM space allocation of system user DBC is used for the following:

This DBC space…	Performs this service …	For this function …
PERM	provides ownership	Teradata production database space, which must accommodate the maximum PERM space allocated to each user and database by its creator. The allocated space is used to store the rows in the data tables created by each user, including: • Primary data table rows • Join index table rows • Hash index table rows • Secondary index subtable rows • Optional permanent journal image rows • Optional fallback rows, which replicate the rows of primary data tables, index subtables, hash and join index tables, and permanent journals
	creates and maintains the transient journal	storing a before-image copy of every row affected by all current sessions.
	stores system table, index, macro, trigger, and view definitions	setting up the Teradata data dictionary.
	provides permanent space	CRASHDUMPS database.

This DBC space…	Performs this service …	For this function …
TEMP	provides temporary space	global temporary tables.
SPOOL	provides maximum defined SPOOL space per creator-defined allocation to each user and database	• dynamic allocation and releasing during query processing. • volatile temporary tables.

System User SYSTEMFE

System user SYSTEMFE is owned by user DBC. SYSTEMFE contains special macros and objects. The quantity of PERM space allocated to SYSEMFE is minimal.

Only NCR field support personnel normally log on under this user name.

System User SYSADMIN

System user SYSADMIN is owned by DBC. SYSADMIN contains special views and macros. The quantity of PERM space allocated to SYSADMIN is minimal.

Only NCR field support personnel normally log on under this user name.

Estimating Database Size Requirements

Allocating PERM, TEMP, and Spool Space

Use the SQL CREATE USER DDL statement to allocate all the PERM space required by the applications that support your enterprise to user SYSADMIN. To determine these space requirements, use the equations provided in "Calculating Total PERM Space Requirements" on page 15-15.

Note: If the Data Dictionary and the CRASHDUMPS database have not yet been created, run the DIP utility and then check the remaining PERM allocation before completing space calculations.

The quantity of space specified is allocated from the current PERM space of user DBC. DBC becomes the owner of the SYSADMIN user and of all users and databases subsequently created. Be sure to leave enough space in DBC to accommodate the growth of system tables and logs and the transient journal.

Some guidelines for estimating temporary spool and transient journal space requirements are explained in the following subsections.

Estimating Administrative Spool Space Requirements

As with PERM space, spool space is allocated from the available space of the owning user. The SPOOL specification in the CREATE statement is only a maximum limit.

During query processing, spool space is dynamically allocated from any free space, including the free PERM space of the user who submitted the query.

In general, the following guidelines apply.

- Reserve 25 to 35 percent of total space for spool space and spool growth buffer.

  When you create user SYSADMIN, you can leave the SPOOL parameter unspecified, so it defaults to the maximum allocation of the owning user, user DBC.
- Allow an extra 5 percent of PERM space in user DBC.
- Each time a new user is created, you can specify the maximum amount of spool space that a query submitted by that user can consume.

Details and formulas for estimating user spool space requirements are provided in "Tables Area: User Spool Space Requirements" on page 15-12, "Rules for Using the Spool Space Equations" on page 15-12, "Field Mode Sizing Equation" on page 15-13, and "Record and Indicator Mode Sizing Equation" on page 15-14.

Estimating Administrative TEMP Space Requirements

Allocate the TEMP space required by the applications that support your enterprise to user SYSADMIN.

Like PERM space, TEMP space is allocated from the available space of the owning user. Make an estimate of temporary space requirements.

The TEMPORARY = n clause in the CREATE USER statement permits you to define how much many bytes are to be allocated to a user for creating global temporary tables. Note that temporary space is reserved prior to spool space for any user defined to have this attribute.

Disk usage for materialized global temporary tables is charged to the temporary space allocation of the user who referenced the table.

If no temporary space is defined for a user, then the space allocated for any global temporary tables referenced by that user is set to the maximum temporary space allocated for the immediate owner.

Estimating Transient Journal Space Requirements

Teradata creates and manages a transient journal to store the before-change image of every data row involved in every SQL transaction.

The transient journal is used to recover data tables when transactions are aborted. As transactions are processed, the transient journal grows and shrinks. While transactions are in progress, the transient journal grows according to the total number of data rows being updated or deleted.

The applicable journal rows are purged at intervals (but not before the transaction is completed or, if it was aborted, not before the affected rows are recovered).

Space for the transient journal is acquired from the current PERM space of user DBC. Therefore, it is important that you leave enough PERM space in DBC to accommodate the growth of the largest foreseeable transient journal.

To estimate the maximum size of the transient journal, follow the procedure documented by this table.

Step	Action
1	Determine the length of the longest row in your production database. Use this figure as your maximum row length.
2	Multiply the maximum row length by the total number of rows in your application programs, batch jobs, and ad-hoc queries that users are likely to update and delete in *concurrent* transactions.

Transient Journal Data Block Size

When you know the maximum row length, double the value to calculate the size of your multirow data block definition in the JournalDBSize field of the DBS Control Record. For example, the JournalDBSize should be greater or equal to two times the maximum row length.

The file system allocates any row that exceeds the current size of a multirow data block to a transient journal data block of its own.

For details on the JournalDBSize field, see the DBS Control utility chapter in *Teradata RDBMS Utilities*.

Determining Available User Table Data Space

Introduction

To determine available user table data space, you must take several *non*user table data space allotments into account.

First you must determine how much space these allotments account for and then you must subtract them from the total available table space.

The nonuser table space allotments that must be accounted for are those in the following list.

- Overhead space.
- Tables area, which includes the following.
  - Data Dictionary (for the transient journal and system tables)
  - CRASHDUMPS database
  - User temporary space
  - User spool space

The following topics describe specific space requirements and provide equations with which to determine variable space requirements.

Overhead

Allow space for overhead as follows.

Number of Cylinders	Purpose
3	Spares
1.5 (minimum)	Cylinder indexes
7 percent of available cylinders	Fragmentation

Tables Area: Dictionary and Spool Space Requirements

After you determine how much space remains when overhead is accounted for, determine how much space is required for the data dictionary and spool files.

You should reserve approximately 80 megabytes for growth of system tables and the transient journal. However, optimum space for the transient journal should be based on how many rows are dropped or updated by concurrently running transactions during your peak workload.

Tables Area: User TEMP Space Requirements

The TEMPORARY = n clause in the CREATE DATABASE statement permits you to allocate default space for creating global temporary tables by users within this database. Note that temporary space is reserved *prior* to spool space for any user defined with this attribute.

Disk usage for a materialized global temporary table is charged against the temporary space allocation of the user who referenced the table.

If no default temporary space is defined for a database, then the space allocated for any global temporary tables created in that database is set to the maximum temporary space allocated for its immediate owner.

Subtract the TEMP amount allocated to this database.

Tables Area: CRASHDUMPS Database Space Requirements

Subtract the PERM amount allocated to this database. The DIP utility creates this database with a default allocation of 1 GB.

If you modified the default amount when you set up the CRASHDUMPS database, subtract the actual current amount.

Tables Area: User Spool Space Requirements

Whenever you create a new user, reserve a minimum of 20 percent of user permanent space for spool space. It is better to determine the optimum amount of space according to your application environment. This takes into account such factors as average table size, protection choices, number of rows involved in commonly used transactions, and so forth.

The optimum allocation depends on your application environment. If your requirement is calculating according to the formulas provided in this section and the calculated amount exceeds 20 percent of permanent space, use the actual figure plus 5 percent. To determine an allowance based on your applications, use the formulas in this chapter.

Rules for Using the Spool Space Equations

When using the spool space equations, keep the following rules in mind.

- The equations are based on the following variables.
  - Number of rows in the spool file
  - Amount of data in each row
  - Mode of select operation (field or record)
- The equations only determine the space needed to return rows to the user.
- Response rows are limited to approximately 64 000 bytes.
  If the row is longer, a row length error is reported.

- Even if an ORDER BY clause is not included, the system creates a minimum length sort key of 8 bytes for both record mode and field mode select operations.
- All descriptions of CHARACTER fields also apply to BYTE fields.
- Be sure the NF, CF, SCF, and RDS values represent the sums of all selected or sorted fields. For example, a FORMAT phrase can cause blanks to be added, which must be figured into the total field length.

Field Mode Sizing Equation

$$\text{Usable Data Space} = a\,(RO + RP + n(PH + NF + CF)) + b(SNF + SCF)$$

The following parameter definitions are used with this equation.

Parameter	Definition
a	Number of rows being selected.
RO	Row overhead (22 bytes per row).
RP	Row parcel indicators; 8 bytes per row (rec start, rec end).
n	Number of columns being selected.
PH	Parcel headers (4 bytes per field).
NF	Formatted size of each numeric field (spaces, dollar signs, and commas should be included).
CF	Formatted size of each character field. For example, if a selected column is defined as VARCHAR(200), it will take up 200 characters even if there are only 3 non-blank characters. The value for CF can be less than the CREATE TABLE definition of the string only if there is a FORMAT phrase.
b	Number of numeric columns in the ORDER BY clause (sort key).
SNF	Sorted numeric fields (8 bytes each).
SCF	Formatted size of each character field in the ORDER BY clause. If an order operation is done on a column defined as VARCHAR(200), 200 characters will be allowed, even if there are only 3 non-blank characters. The value for CF can be less than the CREATE TABLE definition of the string only if there is a FORMAT phrase. The formatted length of the character string should be rounded up to the next even value, then 2 more bytes should be added to the number.

Record and Indicator Mode Sizing Equation

$$\text{Usable Data Space} = a(RO + RP + RDS + (b(SNF + SCF)) + \frac{n}{(IF + 1)})$$

The following parameter definitions are used with this equation.

Parameter	Definition
a	Number of rows being selected.
RO	Row overhead (12 bytes per row).
RP	Row parcel indicators; 8 bytes per row (rec start, rec end).
RDS	Raw data size of each field returned to the user.
b	Number of numeric columns in the ORDER BY clause (sort key).
SNF	Sorted numeric fields (8 bytes each).
SCF	Formatted size of each character field in the ORDER BY clause. If an order operation is done on a column defined as VARCHAR(200), 200 characters will be allowed, even if there are only 3 non-blank characters. The value for CF can be less than the CREATE TABLE definition of the string only if there is a FORMAT phrase. The formatted length of the character string should be rounded up to the next even value, then 2 more bytes should be added to the number.
n	Number of columns being selected.
IF	Record mode selects use indicator variables to represent null values. IF is the constant 8, meaning that for each 8 columns selected, 1 byte is needed to represent a null field.

Calculating Total PERM Space Requirements

Use the following procedure to determine how much PERM space to allocate to user SYSADMIN.

<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>1</td><td>Estimate the size of each database, as follows.
<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>1</td><td>Estimate the size of the primary data table, including fallback (see “Sizing Base Tables and Index Subtables” on page 14-88 for instructions for how to do this).</td></tr>
<tr><td>2</td><td>Estimate the size of unique and nonunique secondary index subtables.</td></tr>
<tr><td>3</td><td>Add the primary data table estimate and secondary index subtable estimates (including fallback) to obtain the estimated total table size.</td></tr>
</table></td></tr>
<tr><td>2</td><td>Estimate the total table storage by adding together the space estimates of all table sizes (Substeps 1, 2, and 3 on page 15-15). Use this sum for step 3.</td></tr>
<tr><td>3</td><td>Estimate the space requirement for the application.

Note: If the calculated sum is greater than the remainder calculated in step 7 on page 15-16, contact your NCR sales representative to discuss a system expansion.
<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>1</td><td>Add 7 percent for fragmentation.</td></tr>
<tr><td>2</td><td>Add extra space for spooling, TEMP space, and for variability in hashing. Although this space requirement varies according to the applications, adding 20 to 30 percent is usually adequate if you do not use permanent journaling.</td></tr>
<tr><td>3</td><td>Estimate the size of each journal table per database.</td></tr>
<tr><td>4</td><td>Note that if journaling is used, the extra space requirement depends on two factors:
• Size of the data tables that write to that journal
• Number of changes applied to those tables before the journal table is dropped</td></tr>
<tr><td>5</td><td>Add the sums calculated for steps 1 through 7 on page page 15-15 and use their sum for the calculation in step 6.</td></tr>
<tr><td>6</td><td>For each disk, add 3 cylinders per disk as spaces and allow 2 percent of the formatted space for cylinder indexes.</td></tr>
</table></td></tr>
</table>

Step	Action
7	Subtract the amount of space needed to accommodate the DBC database contents and your maximum transient journal from user DBC PERM space. Specify about 80 megabytes for growth of system tables and the transient journal, plus another 5 percent of the current PERM space for spool growth. Use the remainder for step 8 on page 15-16.
8	Subtract the sum of step 3 from the result of step 4, if possible.
9	Deduct the number of cylinders that comprise the default free space specified in the DBS Control Record for operations that use the Free Space Percent (FSP). For example, if your FSP is set to 15 percent, deduct a additional 15 percent. If you plan to create certain large tables with a specified FSP that is greater than the GDO default, then use the larger percentage. For example, if you define your largest tables with an FSP of 30 percent, then deduct an additional 30 percent. The remainder is the amount of PERM space you should allocate to the SYSADMIN user.

Note: Table sizing is normally the responsibility of NCR field support personnel.

Designing for Backups

Introduction

There was a time not so very long ago when backups were a minor issue in database management. You set up a batch job to perform the backup overnight, while the system was not being used by anybody else, and that was that. Several issues have complicated that once common scenario.

First is the tremendous increase in the size of databases that can be supported by a relational database management system like Teradata. Where at one time a "very large database" might be on the order of several gigabytes, it is increasingly common to see multiterabyte databases supporting large data warehouses and support for petabyte and even yottabyte databases is on the near horizon. Magnetic tape is a serial medium. Even if you archive to multiple tape drives simultaneously, there is still a considerable time issue involved with backing up multiterabyte databases.

Second is the end of the era of the batch window. Enterprise data warehouses now typically support worldwide operations, and that means that the system must be available to users 24 hours per day, seven days a week. Without the old batch window, how do you back up your business-critical data without having a significant negative effect on the capability of your data warehouse to support its worldwide user base?

Many approaches to solving these problems fall outside the scope of database design, but at least one method for minimizing the impact of backing up very large databases can be designed into your database. That method is the subject of this topic.

Terminology

For purposes of this topic, "large" and "small" tables and databases are defined as follows.

Term	Definition
Small table or database	Less than 100 gigabytes
Large table or database	Greater than 100 gigabytes

Product Issues

The issue described in this topic and its suggested solution apply to the large system environment, primarily when using the ASF2 product to backup to and recovery from different machines. The REEL product makes it possible to restore to or recover from *the same machine* using its fast path option, but even that option does not relieve the problem of backing up to a second machine.

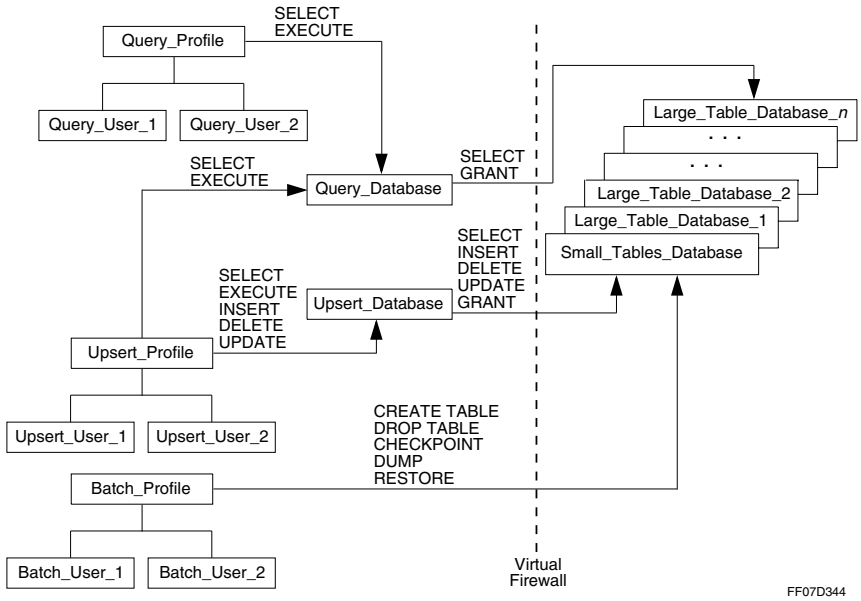
The Scenario

Database and table dumps and restores are a time consuming process. To minimize this problem, you can isolate large tables in their own databases in order to isolate their backup and restoration. Otherwise, when tables of this magnitude are mixed with much smaller tables, you have two choices:

- Back up the large table in one pass and then backup the remaining tables in a second pass.
- Back up the entire database at once, which takes a very long time to complete (and even longer to restore).

A Solution

Design your database in such a way that large tables are isolated within their own, separate databases, then build views for your users to use to access the data. Carefully constructed views can mask this separation and permit users to access data across multiple databases transparently within a single query. The following graphic illustrates one possible design that uses this method.



Section 6:

Designing for Tactical Query Support

Chapter 16:

Design Issues for Tactical Queries

The majority of the topics covered by this manual apply to designing for strategic, or decision support, queries. Because you can move your operational data store into Teradata, this chapter highlights some of the design issues you must consider when implementing a physical design to support mixed workloads of both strategic and tactical queries.

This chapter describes the following topics related to designing for tactical queries:

- "What Are Tactical Queries?" on page 16-4
- "Scalability Considerations for Tactical Queries" on page 16-5
- "Localizing the Work" on page 16-8
- "Application Opportunities" on page 16-20
- "Join Indexes and Tactical Queries" on page 16-25

The chapter does not make suggestions about scheduling job priorities to achieve an optimal mix of strategic and tactical workloads for your system. See *Teradata RDBMS Utilities, Volume 2* for a description of the Priority Scheduler utility.

Table Definitions

The following table definitions are used for the examples in this chapter:

Supplier

```
CREATE TABLE supplier (
s_suppkey INTEGER NOT NULL,
s_name CHARACTER(25) CASESPECIFIC NOT NULL,
s_address VARCHAR(40) CASESPECIFIC NOT NULL,
s_nationkey INTEGER NOT NULL,
s_phone CHARACTER(15) CASESPECIFIC NOT NULL,
s_acctbal DECIMAL(15,2) NOT NULL,
s_comment VARCHAR(101) CASESPECIFIC NOT NULL)
UNIQUE PRIMARY INDEX (s_suppkey);
```

Supplier2

```
CREATE TABLE supplier2 (
s_suppkey INTEGER NOT NULL,
s_name CHARACTER(25) CASESPECIFIC NOT NULL,
s_address VARCHAR(40) CASESPECIFIC NOT NULL,
s_nationkey INTEGER NOT NULL,
s_phone CHARACTER(15) CASESPECIFIC NOT NULL,
s_acctbal DECIMAL(15,2) NOT NULL,
s_comment VARCHAR(101) CASESPECIFIC NOT NULL)
UNIQUE PRIMARY INDEX (s_suppkey);
```

Customer

```
CREATE TABLE customer (
c_custkey INTEGER NOT NULL,
c_name VARCHAR(25) CASESPECIFIC NOT NULL,
c_address VARCHAR(40) CASESPECIFIC NOT NULL,
c_nationkey INTEGER NOT NULL,
c_phone CHARACTER(15) CASESPECIFIC NOT NULL,
c_acctbal DECIMAL(15,2) NOT NULL,
c_mktsegment CHARACTER(10) CASESPECIFIC NOT NULL,
c_comment VARCHAR(117) CASESPECIFIC NOT NULL)
UNIQUE PRIMARY INDEX (c_custkey);
```

Orders

```
CREATE TABLE orders (
o_orderkey DECIMAL(15,0) NOT NULL,
o_custkey INTEGER NOT NULL,
o_orderstatus CHARACTER(1) CASESPECIFIC NOT NULL,
o_totalprice DECIMAL(15,2) NOT NULL,
o_orderdate DATE,
o_orderpriority CHARACTER(15) CASESPECIFIC NOT NULL,
o_clerk CHARACTER(15) CASESPECIFIC NOT NULL,
o_shippriority INTEGER,
o_comment VARCHAR(79) CASESPECIFIC NOT NULL)
UNIQUE PRIMARY INDEX (o_orderkey);
```

Lineitem

```
CREATE TABLE lineitem (
l_orderkey DECIMAL(15,0) NOT NULL,
l_partkey INTEGER NOT NULL,
l_suppkey INTEGER NOT NULL,
l_linenumber INTEGER,
l_quantity DECIMAL(15,0),
l_extendedprice DECIMAL(15,2) NOT NULL,
l_discount DECIMAL(15,2) NOT NULL,
l_tax DECIMAL(15,2) NOT NULL,
l_returnflag CHARACTER(1) CASESPECIFIC NOT NULL,
l_linestatus CHARACTER(1) CASESPECIFIC NOT NULL,
l_shipdate DATE,
l_commitdate DATE,
l_receitdate DATE,
l_shipinstruct CHARACTER(25) CASESPECIFIC NOT NULL,
l_shipmode CHARACTER(10) CASESPECIFIC NOT NULL,
l_comment VARCHAR(44) CASESPECIFIC NOT NULL)
UNIQUE PRIMARY INDEX (l_orderkey, l_linenumber);
```

Nation

```
CREATE TABLE nation (
n_nationkey INTEGER NOT NULL,
n_name CHARACTER(25) CASESPECIFIC NOT NULL,
n_regionkey INTEGER NOT NULL,
n_comment VARCHAR(152) CASESPECIFIC NOT NULL)
UNIQUE PRIMARY INDEX (n_nationkey);
```

What Are Tactical Queries?

Introduction

Traditionally, data warehouse applications have been based on drawing strategic advantage from the data. Strategic queries are often complex, sometimes long-running, usually broad in scope. The parallel architecture of Teradata supports these types of queries by spreading the work across all of the parallel units and nodes in the configuration.

As data warehouse users require more versatility out of their data, they are supplementing their strategic focus with a tactical component. Today, increasing numbers of Teradata users are introducing the quick turnaround tactical queries alongside of strategic queries in their Teradata system.

Tactical Queries

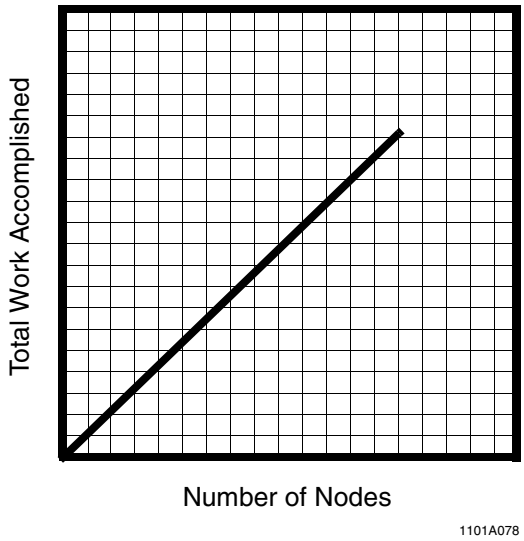
The name tactical query is given to a certain category of work that is short, highly tuned, and focused on more operational decision-making. This class of queries is sometimes categorized under the initialism OLCP, which stands for Online Complex Processing. Tactical queries are similar to classical online transaction processing (OLTP), in that both are composed of short, often direct-access queries, coming with defined response time requirements.

OLTP, however, is more focused on bookkeeping activities and recording operational events, while tactical queries are more focused on decision-making, recording selections or choices made. OLTP generally comes with a higher ratio or writes compared to reads, while tactical queries tend to be more read intensive. OLTP usually has a departmental view and a very narrow context, while tactical queries are often more enterprise-wide. With OLTP queries, short latency is always critical. Tactical queries, on the other hand, are designed for response time requirements that can vary from subsecond up to 10 or 20 seconds or more.

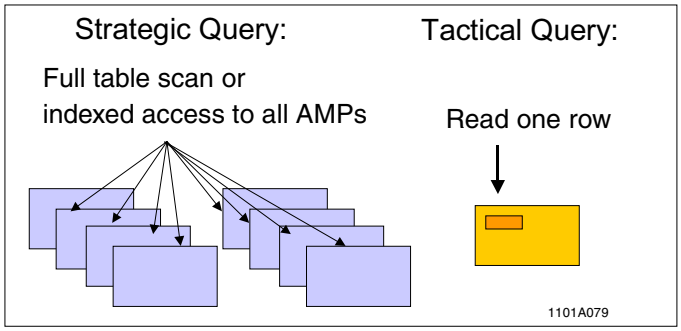
Scalability Considerations for Tactical Queries

Scalability Is a Relative Concept

In traditional decision support applications, setting the stage for scalable performance with Teradata involves distributing the work equally across all the AMPs in the system. As more nodes are added to a system, the number of AMPs increases proportionally, and the effort to process a complex query is spread across more parallel units, reducing response times. The following plot indicates linear scalability, graphing the total workload accomplished as a function of the number of nodes in the system. Strategic queries run proportionally faster as more nodes are added to a system.



Spreading the data evenly across all AMPs remains the goal of designing the database to support tactical queries. But to achieve the type of scalability that best supports tactical queries, you need to localize the work in such a way that it accesses only a few rows using the least resources possible. This means accessing the smallest number of AMPs possible to perform an operation; optimally only one. The following figure indicates that when tactical queries access only a few AMPs, more of them can run concurrently.



Effect of Data Volume Growth on Tactical Query Response Times

Query times for complex decision support are directly impacted by increases in the data volume. Because a large portion, if not all, of a table might need to be read, if that table doubles in size, then the effort required to process the work can also double. On the other hand, a query that accesses one or a few rows in the database is not impacted by changes to the data volume. In most cases, a tactical query accesses one or few rows whether the table those rows are in is 1 GB or 10 TB in size.

Effect of Growth in Concurrent Users on Tactical Query Response Times

Raising the level of concurrent users doing the same work in traditional decision support tends to slow response times for the current users because more demands are now being made on the same resources, and all are spread out across all nodes and AMPs in the system. In contrast, increasing the number of users performing tactical queries that are very localized and limited in their resource use boosts overall throughput up to the point of system saturation.

Effect of Configuration Expansion on Tactical Query Response Times

For a complex query, adding nodes translates to a proportional decrease in response time because the work is distributed across a larger number of AMPs. At the same time, the response time for a query that accesses a single AMP, as many tactical queries do, is not affected by the number of AMPs in the configuration.

There is a one benefit gained by single-AMP tactical queries when nodes are added to a configuration: more of them can be performed at the same time and still deliver short turnaround times.

Consider the following contrived example: Assume you have a table with a cardinality of 1 million rows. The read capacity of each node in your configuration is 100 rows per second. If you are doing a complex strategic query that involves a full table scan of this table, then the response time for the query diminishes proportionally with the increase in nodes, as illustrated by the data in the following table:

Number of Nodes	Number of Rows per Node	Response Time (seconds)
1	1 000 000	10 000
10	100 000	1 000
100	10 000	100
200	5 000	50

This performance enhancement occurs because each node has fewer rows as more nodes are added, so each node can perform its portion of the scan faster.

On the other hand, if your application is performing single- or few-AMP tactical queries, adding nodes does not shorten the response time. However, it does increase the number of tactical queries that can be performed in a given interval of time, as indicated by the data in the following table:

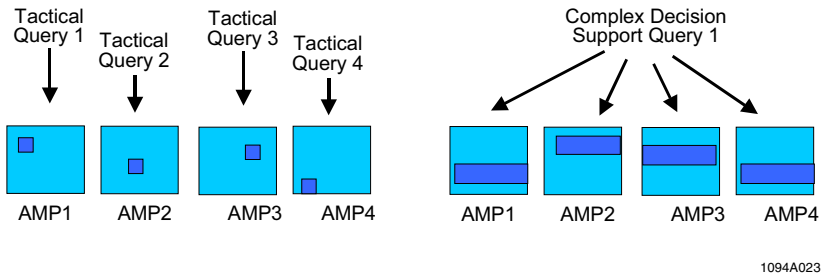
Number of Nodes	Response Time (seconds)	Throughput (requests/second)
1	0.01	100
10		1 000
100		10 000
200		20 000

Localizing the Work

Introduction

After an SQL statement is optimized, the Dispatcher sends the individual steps that make up that query, one at a time, to some number of AMPs in the system. Steps are usually dispatched, and processes started, on all AMPs in the system. Under some conditions, the Optimizer might decide that this query work can be accomplished by accessing only a single AMP in the configuration. This is referred to as a single-AMP operation, or a single-AMP step.

Teradata uses single-AMP steps to localize work to the fewest resources necessary when it is beneficial to do so. This is frequently the case for tactical queries. Single-AMP work frees the other AMPs in the system to do other work, thereby increasing the potential for overall system throughput. This is diagrammed in the following figure.

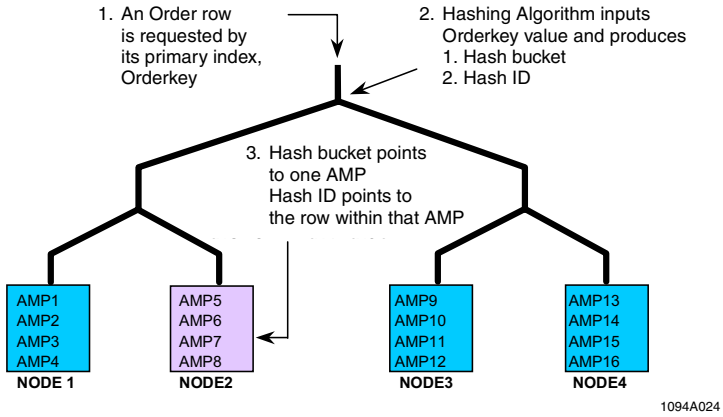


Tactical queries are most efficient and most scalable when designed around single- or few-AMP operations. This is the single most important way to increase throughput and preserve response times for very short tactical queries.

Single- or few-AMP queries scale well because they engage the same small level of database resources even when the system doubles or triples in size (see “Effect of Configuration Expansion on Tactical Query Response Times” on page 16-6).

Single-AMP Operations

Single-AMP operations use a primary index value to locate a row.



Single-AMP operations can be achieved by any of the following data manipulation operations:

- Simple single-row inserts
- Simple selects, updates and deletes qualified with a primary index value
- Joins between two tables that share one of the following characteristics:
  - the same primary index domain
  - a primary index making up the join constraint
  - a single primary index value is specified for the join

The following EXPLAIN report is for a query that accesses the supplier table using the single primary index value s_suppkey = 583. Only a single AMP is engaged, as demonstrated by the single-AMP RETRIEVE step reported in step 1 of the EXPLAIN text:

```
EXPLAIN
SELECT s_name, s_acctbal
FROM supplier
WHERE s_suppkey = 583;
```

```
Explanation
---------------------------------------------------------------------------
  1) First, we do a single-AMP RETRIEVE step from TPCD50G.supplier by
     way of the unique primary index "TPCD50G.supplier.S_SUPPKEY = 583"
    with no residual conditions.  The estimated time for this step is 0.03 seconds.
```

Notice that there are no references to locking in the EXPLAIN report for this query. That is because the Optimizer has folded the locking activity (in this case a single row hash READ lock) into the same step that retrieves the row. This sort of lock folding is done only with row hash locks.

This special shortcut for handling row hash locks eliminates the need for the Dispatcher to dispatch a separate locking step when only one AMP and one row are involved. This reduces the PE-to-AMP communication effort.

The following example performs the same query against a table that differs only in its primary index definition: supplier2 has s_name as its primary index instead of s_suppkey. Because a primary index value for supplier2 table is not specified in the WHERE clause, the Optimizer is forced to specify an all-AMPs full table scan.

```
EXPLAIN
SELECT s_name, s_acctbal
FROM supplier2
WHERE s_suppkey = 583;
```

```
Explanation
------------------------------------------------------------------------
  1) First, we lock a distinct TPCD50G."pseudo table" for read on a
     RowHash to prevent global deadlock for TPCD50G.supplier2.
  2) Next, we lock TPCD50G.supplier2 for read.
  3) We do an all-AMPs RETRIEVE step from TPCD50G.supplier2 by way of
     an all-rows scan with a condition of (
     "TPCD50G.supplier2.S_SUPPKEY = 583") into Spool 1, which is built
     locally on the AMPs.  The size of Spool 1 is estimated with high
     confidence to be 1 row.  The estimated time for this step is 7.61 seconds.
  4) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

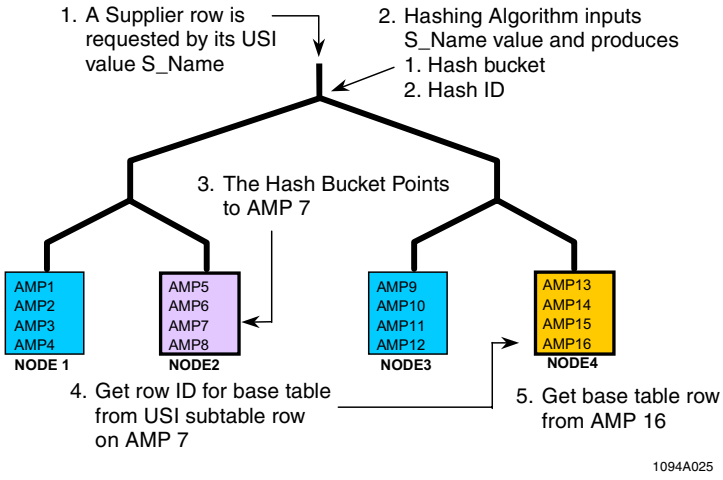
Step two in this query plan reports that a table-level READ lock is applied to supplier2. Table-level locks are always applied in a separate step that precedes database access. The END TRANSACTION step is also an all-AMPs operation that is required to remove the table-level lock from each AMP.

Few-AMP Operations

Few-AMP operations are also desirable in many tactical query situations. Several types of operations, such as USI access, can ensure that only a few AMPs are engaged.

USI Access and Tactical Queries

When an application specifies a value that can be used to access a table using its USI, a 2-AMP[1] operation results.



1 USI access *can* be a single-AMP operation if the USI value for a row happens to hash to a subtable on the same AMP as the primary index for the same row, but is *never* more than a 2-AMP operation.

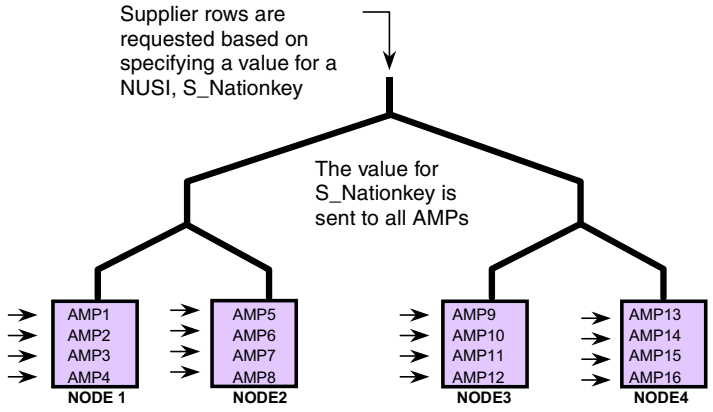
In the following query, the column s_name is defined as a USI on the original supplier table having s_suppkey as its UPI.

```
EXPLAIN
SELECT s_suppkey, s_acctbal
FROM supplier
WHERE s_name = 'Supplier#000038729';
```

```
Explanation
------------------------------------------------------------------------
  1) First, we do a two-AMP RETRIEVE step from CAB.supplier by way
     of unique index # 8 "CAB.supplier.S_NAME = 'Supplier#000038729'" with no
     residual conditions.  The estimated time for this step is 0.07 seconds.
```

NUSI Access and Tactical Queries

Compare the previous example of USI access to access using a NUSI. With NUSIs, each AMP contains an index structure for the base table rows that it owns. Even if only a single row contains the specified NUSI value, the Optimizer cannot know that a priori, so the NUSI subtables for all AMPs are always searched, making all NUSI accesses all-AMPs operations.



The following example assumes the s_name column is defined as a NUSI on supplier. Notice the difference between this EXPLAIN report and the previous one for the USI (see "USI Access and Tactical Queries" on page 16-11):

```
EXPLAIN
SELECT s_suppkey, s_acctbal
FROM supplier
WHERE s_name = 'SUPPLIER#000000647';
```

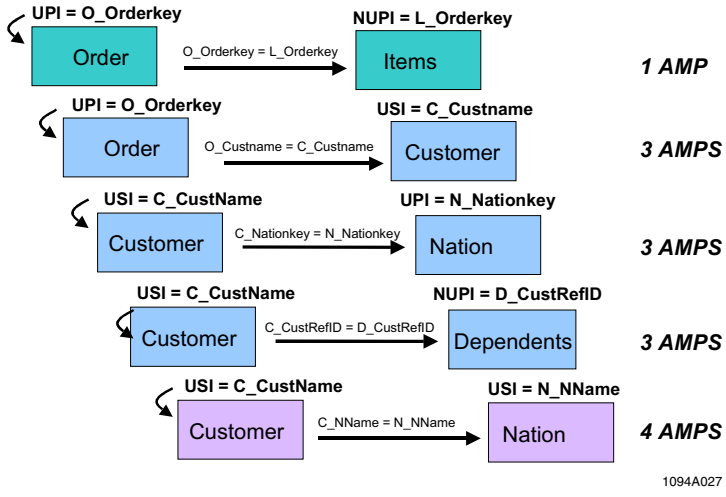
```
Explanation
-----------------------------------------------------------------------
  1) First, we lock a distinct TPCD50G."pseudo table" for read on a
     RowHash to prevent global deadlock for TPCD50G.supplier.
  2) Next, we lock TPCD50G.supplier for read.
  3) We do an all-AMPs RETRIEVE step from TPCD50G.supplier by way of
     index # 8 "TPCD50G.supplier.S_NAME = 'SUPPLIER#000000647'" with no
     residual conditions into Spool 1, which is built locally on the
     AMPs.  The size of Spool 1 is estimated with high confidence to be
     1 row.  The estimated time for this step is 0.20 seconds.
  4) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

With NUSI access, table-level locks are applied and an all-AMPs operation is performed. Value-ordered indexes and covering indexes are variants of a NUSI. Because of that, they also require all-AMPs operations that are similar to NUSI access.

Few-AMP Joins and Tactical Queries

Even when joins must be made to process a tactical query, it is possible to make the join with few-AMP operations. The following picture illustrates the combinations of accessing and joining that can be performed engaging one or few AMPs. In all of these few-AMP examples, the first table is accessed using either a UPI or a USI

The first join in the following example is made using a merge join. Because the primary index columns of both tables share the same domain, their associated rows reside on the same AMP, with the second table being joined based on a NUPI. The remaining joins use combinations of primary index and USI access and perform few-AMP nested joins between the tables.



One idiosyncrasy of EXPLAIN text is that it sometimes says "we do a two-AMP join step" when a USI is defined on one half of the nested join. Three AMPs are engaged for most such joins, not two. For example, the third join in the preceding graphic produces this EXPLAIN, but it engages 3 AMPs rather than the 2 reported.

```
   EXPLAIN SELECT *
   FROM customer, nation
   WHERE c_nationkey = n_nationkey
   AND c_name = 'Customer#75532';

Explanation
------------------------------------------------------------------------
  1) First, we do a two-AMP JOIN step from TPCD50G.customer by way of
     unique index # 8 "TPCD50G.customer.C_NAME = 'me'" with no residual
     conditions, which is joined to TPCD50G.nation by way of the
     primary index "TPCD50G.nation.N_NATIONKEY =
     TPCD50G.customer.C_NATIONKEY".  TPCD50G.customer and
     TPCD50G.nation are joined using a nested join, with a join
     condition of ("(1=1)").  The input table TPCD50G.customer will not
     be cached in memory, but it is eligible for synchronized scanning.
     The result goes into Spool 1, which is built locally on the AMPs.
```

The following EXPLAIN report demonstrates a single-AMP join of the orders and lineitem tables on their respective primary indexes o_orderkey and l_orderkey. This is similar to the first join in the graphic. Only one step is required, and only one AMP is engaged.

```
   EXPLAIN
   SELECT l_quantity, l_partkey, o_orderdate
   FROM orders, lineitem
   WHERE o_orderkey = l_orderkey
   AND o_orderkey = 58389;

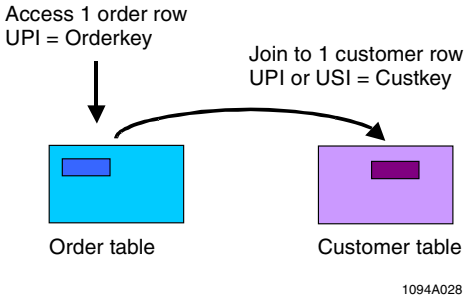
Explanation
------------------------------------------------------------------------
1)First, we do a single-AMP JOIN step from CAB.ordersorders by way of the unique
primary index "CAB.orders.O_ORDERKEY = 58389" with no residual conditions,
which is joined to CAB.lineitem by way of the primary index
"CAB.lineitem.L_ORDERKEY = 58389".  CAB.orders and CAB.lineitem are joined
using a merge join, with a join condition of ("CAB.orders.O_ORDERKEY =
CAB.lineitem.L_ORDERKEY").  The input tables CAB.orders and CAB.lineitem will
not be cached in memory, but CAB.orders is eligible for synchronized scanning. The
result goes into Spool 1, which is built locally on that AMP. The size of Spool 1 is
estimated with high confidence to be 4 rows.
```

Tactical Queries Benefit From Nested Joins

All AMPs are engaged for product and hash joins, and also for most merge joins. For this reason these join methods are less desirable for more localized queries. Tactical queries benefit from the few-AMP joins described in “Few-AMP Joins and Tactical Queries” on page 16-13.

Of these join methods, the most beneficial for tactical queries is the nested join, which under specific conditions involves only a few AMPs.

A few-AMP nested join is possible when the first table can be accessed by a UPI or USI specified by the application. Following that, a foreign key within that first table must be available to drive a nested join into a second table, which contains its associated primary key. For this nested join to involve only one or a few AMPs, the second table must have either a primary index or a USI defined on the column set mapped to by the foreign key in the first table, as seen in the following figure.



The following EXPLAIN reports the plan for a query that joins the orders and customer tables based on the value of the UPI of the first table (orders) being available and then joins to the UPI (c_custkey) in the second table (customer). The EXPLAIN reports a single AMP, but in fact two AMPs are engaged, all within a single step: one for primary index access to a single orders table row and one for access to a single customer table row.

```
EXPLAIN
SELECT o_orderdate, o_totalprice, c_name
FROM orders, customer
WHERE o_custkey = c_custkey
AND o_orderkey = 378453;
```

```
Explanation
------------------------------------------------------------------------
  1) First, we do a single-AMP JOIN step from CAB.orders by way
     of the unique primary index "CAB.orders.O_ORDERKEY = 378453"
     with no residual conditions, which is joined to CAB.customer
     by way of the unique primary index "CAB.customer.C_CUSTKEY =
     CAB.orders.O_CUSTKEY".  CAB.orders and
     CAB.customer are joined using a nested join, with a join
     condition of ("(1=1)").  The input tables CAB.orders and
     CAB.customer will not be cached in memory, but
     CAB.orders is eligible for synchronized scanning.  The
     result goes into Spool 1, which is built locally on that AMP.  The
     size of Spool 1 is estimated with high confidence to be 1 row.
     The estimated time for this step is 0.18 seconds.
```

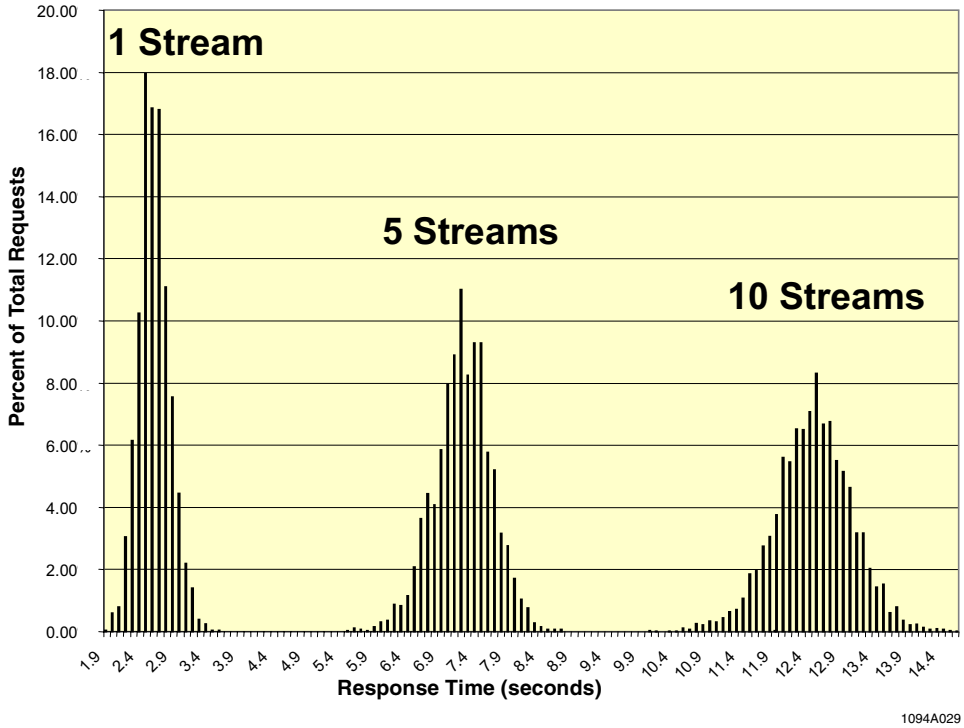
All-AMP Queries

Some tactical queries require all-AMPs steps. All-AMPs queries are likely to have a more relaxed response time expectation. Some suggestions for coding tactical queries that involve all-AMPs operations follow:

- Review the queries for unnecessary database access. If found, eliminate those accesses from the query.
- Rely on NUSI access, where feasible, to avoid scanning a large table.
- Define a join index where it makes sense to do so. Scanning a join index that contains only a subset of the columns from the base table is faster than scanning the base table.
- Look for possibilities to break complex, long running tactical queries into several smaller statements and encapsulate them within a stored procedure.
- Specify explicit ACCESS locks wherever possible (see the chapter "Transaction Processing" in *Teradata RDBMS SQL Reference, Volume 2* for further information).

All-AMP tactical queries behave differently than few-AMP queries when concurrency increases. The following histogram plots percent of total requests as a function of response time for the same repetitively executed all-AMPs query with one, 5, and 10 active streams. Each bar in the histogram represents the percentage of total query executions that ran at that particular response time. For example, when one all-AMP query stream was active, 18 percent of the queries completed in 2.7 seconds and 17 percent completed in 2.9 seconds.

The following histogram plots the percentage of total tactical query requests as a function of response time.



You can readily see that with one stream of moderate all-AMP queries active, the variability in response time is less than when 5 or 10 streams are active. The histogram shows conclusively that all-AMPs tactical queries take longer to perform as more users become active.

The following table reports the response time distributions for all-AMPs tactical queries as the concurrency load on the system increases.

Note that mean query times increase notably as the concurrency load on the system grows.

Response Time Interval (seconds)	Percentage of Queries in the Interval		
	1 Stream	5 Streams	10 Streams
1 - 2	0.65		
2 - 3	95.00		
3 - 4	4.35		
4 - 5			
5 - 6		0.76	
6 - 7		31.33	
7 - 8		65.38	
8 - 9		2.48	
9 - 10			
10 - 11			1.12
11 - 12			21.88
12 - 13			62.62
13 - 14			13.36
14 - 15			0.70
Mean response time (seconds)	2.7	7.3	12.5
Maximum response time (seconds)	3.6	8.7	14.6

This is in contrast to single-AMP queries that access one row (see the following table), where response times remain overwhelmingly consistent, even as the concurrency load on the system for similar work increases. A substantial majority of single-AMP queries retain subsecond response times even as the number of users on the system increases.

Response Time Interval (seconds)	Percentage of Queries in the Interval		
	1 Stream	5 Streams	10 Streams
0.00 - 0.10	99.998	99.956	99.406
0.10 - 0.20		0.021	0.547
0.20 - 0.30	0.003	0.015	0.104
0.30 - 0.40		0.008	0.023
0.40 - 0.50		0.001	0.001
0.50 - 1.00			0.007
1.00 - 2.00			0.003
Mean response time (seconds)	0.006	0.007	0.009
Maximum response time (seconds)	0.340	0.400	1.900

The All-AMPs tactical query in these tests involves NUSI access of customer table rows using c_nationkey followed by an all-AMPs nested join to orders using the NUSI o_custkey. Here is the SQL code for the tests:

```
USING key1 (CHAR(12)), key2 (CHAR(15)), key3 (CHAR(6))
SELECT COUNT(*), AVG(c_acctbal), SUM(o_totalprice)
FROM customer c, orders o
WHERE o_custkey = c_custkey
AND  c_nationkey = :key1
AND  c_mktsegment = :key2
AND  substr(c_phone,1,6) = :key3;
```

Database Design Techniques to Support Localized Work

Several approaches to physical database design can make one-AMP or few-AMP query plans more likely. Most of these choices need to be made carefully, because they also influence how other work in the system is optimized. All applications running in the database need to be considered, particularly when selecting or changing primary index columns:

- Same Primary Index Definitions

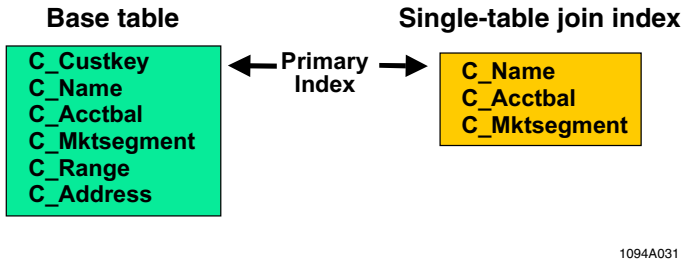
  When you expect frequent joins between two associated tables, consider using identical column sets for the primary index definitions of both tables. In other words, define the primary indexes for both tables identically on the join column set. This technique can enhance both decision support and tactical queries, depending on the frequency of the join and the demographics of the data.

- Increase the Likelihood of Few-AMP Nested Joins

  Consider placing USIs on the join constraint of one of the tables. From the perspective of logical database design, you are placing the USI on the primary key of the primary table in the primary key-foreign key relationship.

- Consider a Join Index

  You can create a single-table join index or hash index with a different primary index than the base table. For example, you could define a primary index for the join or hash index composed of the column that corresponds to values frequently specified by the application. The example in the following diagram indicates how a query that only has a value for customer name could use the join index for single-AMP access.



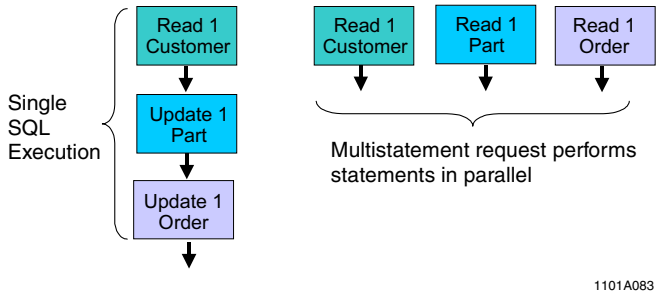
Application Opportunities

Introduction

In addition to physical database design choices, there are several application opportunities that benefit tactical query performance. The main areas of interest are multistatement requests and macros.

Multistatement Requests

Multistatement requests are a Teradata feature in which multiple SQL statements are bundled together and treated as a single parsing and recovery unit, as illustrated by the following graphic:



Because they are run in parallel, these single-AMP multistatement requests are processed with great efficiency. Multistatement requests are application-independent and can improve performance in a variety of ways. They are particularly useful in improving response times for the combined statements.

The benefits of multistatement requests include the following:

- Communication overhead is reduced
- One parser-optimization process is performed instead of several
- Greater inter-request parallelism is possible

Coding Multistatement Requests

Multistatement requests are coded by placing the terminating SEMICOLON character of each statement in a column other than the last position, so that some amount of text for the next SQL statement follows the termination character from the previous statement. For example, Teradata optimizes the following SQL statements as a single request and attempts to perform the three statements in parallel:

```
EXPLAIN
SELECT c_acctbal FROM customer WHERE c_custkey =38521
; SELECT p_retailprice FROM parttbl WHERE p_partkey = 9811
; SELECT o_totalprice FROM orders WHERE o_orderkey = 53355;
```

```
Explanation
------------------------------------------------------------------------
  1) First, we execute the following steps in parallel.
       1) We do a single-AMP RETRIEVE step from CAB.customer by way
          of the unique primary index "CAB.customer.C_CUSTKEY =
          38521" with no residual conditions.  The estimated time for
          this step is 0.03 seconds.
       2) We do a single-AMP RETRIEVE step from CAB.parttbl by way
          of the unique primary index "CAB.parttbl.P_PARTKEY = 9811"
          with no residual conditions.  The estimated time for this
          step is 0.03 seconds.
       3) We do a single-AMP RETRIEVE step from CAB.orders by way
          of the unique primary index "CAB.orders.O_ORDERKEY =
          53355" with no residual conditions.  The estimated time for
          this step is 0.03 seconds.
  2) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> The row is sent directly back to the user as the result of statement 1.
     The row is sent directly back to the user as the result of statement 2.
     The row is sent directly back to the user as the result of statement 3.
     The total estimated time is 0.03 seconds.
```

As the EXPLAIN report indicates, the three statements are performed as parallel steps, each accessing a single row (usually from a different AMP) at roughly the same time. The END TRANSACTION step 2 involves only the three AMPs required to access the three rows. In this context, END TRANSACTION is not an all-AMPs operation because only three AMPs participate in the processing.

Multistatement requests are treated as a single implicit transaction. For example, if several UPDATE statements are within one multistatement request, and one of them fails, then all are rolled back. The fewer UPDATE statements in the multistatement request, the lower the impact of the rollback. However, the more statements included in the request, the higher the degree of parallelism among them.

ACCESS Locking and Multistatement Requests

If you want each statement in a multistatement request to use row hash-level ACCESS locking, specify the LOCKING ROW FOR ACCESS modifier preceding each individual SELECT statement in the transaction:

```
EXPLAIN
LOCKING ROW FOR ACCESS
SELECT c_acctbal FROM customer WHERE c_custkey = 53856
 ; LOCKING ROW FOR ACCESS
SELECT p_retailprice FROM parttbl WHERE p_partkey = 53856
 ; LOCKING ROW FOR ACCESS
SELECT o_totalprice FROM orders WHERE o_orderkey = 53856
```

```
Explanation
-----------------------------------------------------------------------
  1) First, we execute the following steps in parallel.
       1) We do a single-AMP RETRIEVE step from CAB.customer by way
          of the unique primary index "CAB.customer.C_CUSTKEY =
          53856" with no residual conditions locking row for access.
          The estimated time for this step is 0.03 seconds.
       2) We do a single-AMP RETRIEVE step from CAB.parttbl by way
          of the unique primary index "CAB.parttbl.P_PARTKEY =
          53856" with no residual conditions locking row for access.
          The estimated time for this step is 0.03 seconds.
       3) We do a single-AMP RETRIEVE step from CAB.orders by way
          of the unique primary index "CAB.orders.O_ORDERKEY =
          53856" with no residual conditions locking row for access.
          The estimated time for this step is 0.03 seconds.
  2) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
```

Macros and Tactical Queries

You automatically get a multistatement request without coding it if you write multiple SQL statements within a macro because macros are always performed as a single request. The macro code can also specify parameters that are replaced by data each time the macro is performed. The most common way of substituting for the parameters is to specify a USING clause (see “USING Row Descriptor” in *Teradata RDBMS SQL Reference, Volume 6*).

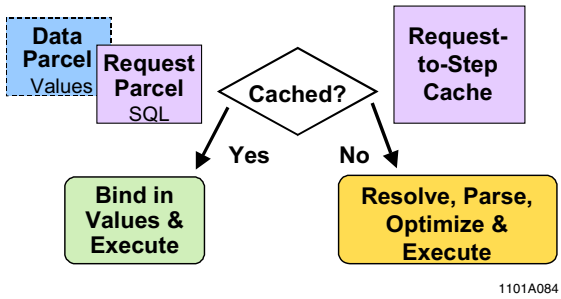
Macros are usually more efficient for repetitive queries than single DML statements. Unlike stored procedures, macros are not designed to return parameters to the requestor: they return only the answer set for the SQL statements they contain.

Macros have the following advantages:

- Network and channel traffic are reduced.
- Execution plans can be cached, reducing parsing engine overhead.
- They ensure the efficient performance of their component SQL statements.
- Reusable database components save client resources.
- They can be used to force data integrity and locking modifiers.
- They can be used to enforce security.

Cached Plans

Caching of repeatable requests in the Request-to-Step Cache reduces performance time because parsing and optimizing do not need to be done when cached requests are repeated. For subsecond queries, cached plans significantly reduce the query time and enhance throughput. The following figure is a high-level flow chart for the request-to-steps cache that resides in each parsing engine on a Teradata system.

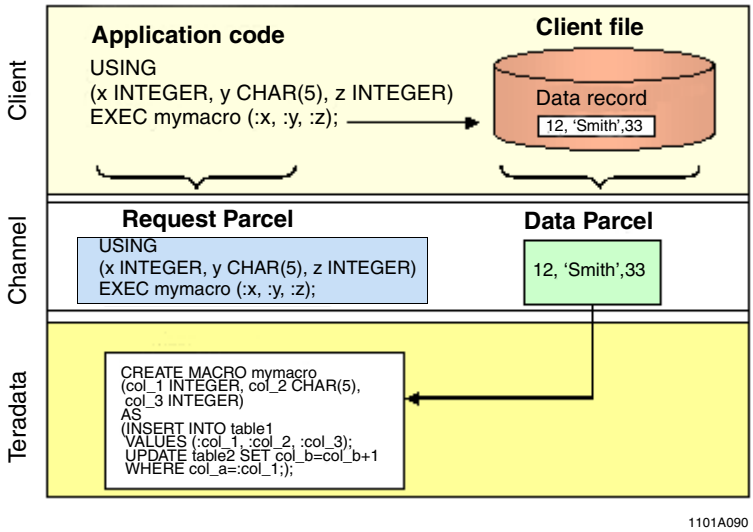


For macros to be considered for caching, Teradata must know that the SQL text they contain is repeatable. All of the following attributes must be identical each time the request is sent to the server:

- Host, workstation, or LAN type
- Default database name
- National character set
- Request text

Parameter variables in the macro support repeatability. These parameters must be specified in both the SQL code itself and in the USING clause that precedes the EXEC statement that invokes the macro.

The following picture illustrates the conditions required for caching the SQL statements for a macro. In this example, the request is made using BTEQ from a mainframe client across a block multiplexor channel. Note that the SQL code is in a request parcel, while the data values are transmitted in a data parcel. Both a request and a data parcel must be transmitted for the SQL to be cached immediately.



The data values are passed from the application to the macro code where they are then processed. Their format depends on the programming conventions for the application. For example, in BTEQ you must use the .IMPORT FILE = command when you perform the macro, and the import file specified in the command contains records with the data values to be passed to the macro. The values in the file are positionally associated, from left to right, with the parameters in the USING clause.

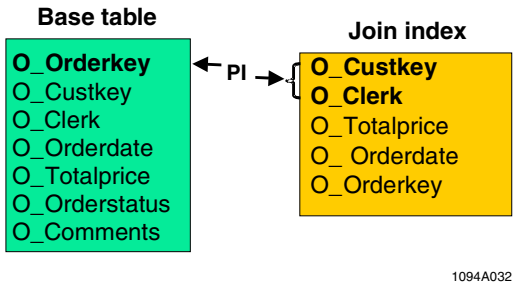
Join Indexes and Tactical Queries

Single-Table Join Indexes

One of the most useful constructs for tactical queries is the single-table join index. Because you can define a primary index for the join index composed of different columns than those used to define the base table primary index, you can create an alternative method of directly accessing data in the associated base table.

For example, you could create a join index on the orders table that includes only a subset of the columns that a particular tactical query application might require. In the example shown in the following graphic, assume that the application has available a value for the customer key and the clerk that placed the order, but does not have a value for orderkey. Specifying the primary index defined for the join index OrderJI supports direct access to order base table using the single-table join index OrderJI.

```
CREATE JOIN INDEX OrderJI AS
SELECT o_custkey, o_clerk, o_totalprice, o_orderdate,
o_orderkey
FROM orders
PRIMARY INDEX(o_custkey,o_clerk)
```



A test against an orders table with 75 million rows, both with and without the OrderJI join index, returned the following response times from the query and join index above:

If the query is run ...	The response time is ...
without the join index	1:55.
with the join index	subsecond.

The larger the base table is, the longer it takes to process via a table scan, and the greater the benefit a join index providing single-AMP access provides. In this example, the join index took 8:23 to create.

A single-table join index like this one is particularly useful when the tactical application does not have the primary index of the base table available, but has an alternative row identifier. This might be the case when a social security number is available, but a member ID, the primary index of the base table, is not. Single-AMP access would still be achievable using the join index if its primary index is defined on social security number.

Aggregate Join Indexes

If your application supports repeated access to the same table using aggregation along the same set of dimensions, you should consider using aggregate join indexes to enhance the performance of those queries.

If you include one or more aggregating columns in the join index select list, then that index is an aggregate join index. Aggregate join indexes are dynamic summary tables, not snapshots. For example, the following aggregate join index computes a running sum on o_totalprice from the orders table:

```
CREATE JOIN INDEX ordersum AS
SELECT o_clerk, SUM(o_totalprice) AS sumprice
FROM orders
GROUP BY o_clerk
PRIMARY INDEX(o_clerk);
```

Each time the o_totalprice column of orders is updated, a new sum is computed and stored in the ordersum aggregate join index. If there are frequent queries throughout the day that request summaries of the total prices for orders placed by a specific clerk, this join index would be able to deliver response times suitable for tactical queries like the following example:

```
SELECT o_clerk, SUM(o_totalprice)
FROM orders
GROUP BY o_clerk
WHERE o_clerk = 'Clerk#000046240';
```

A test against an orders table with 75 million rows, both with and without the ordersum join index, returned the following response times from the query and join index above:

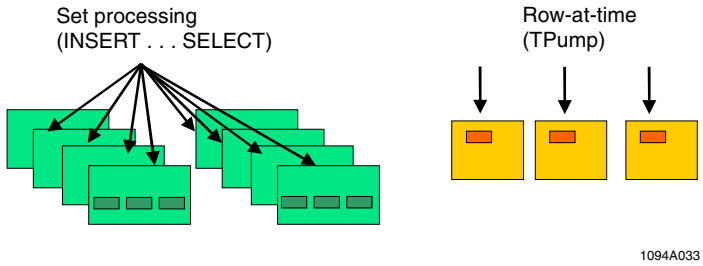
If the query is run …	The response time is …
without the join index	1:53.
with the join index	subsecond.

The join index took 5:48 to create. All tests were performed on a two-node 4800 system with four 550 MHz CPUs per node.

Use the GROUP BY column set as the primary index of the join index if this is the column whose value is specified by the application. Each query that specifies a value for that column will then be processed as a single-AMP request.

Join and Hash Index Maintenance Considerations

Each time a base table row is updated, its corresponding join or hash index data is modified within the same unit of work. Because MultiLoad and FastLoad are incompatible with join and hash indexes, maintenance of a base table that has a join or hash index must be performed using SQL, or the join or hash index must be dropped before the MultiLoad or FastLoad utility is run and the index then rebuilt afterward. As illustrated by the following figure, you can use SQL to take either a set processing approach or a row-at-a-time approach to join and hash index maintenance.



If an INSERT … SELECT statement is used to load data into the base table, then the unit of work is the entire SQL statement. In such a case the join or hash index maintenance is performed in batch mode. Inserts to the base table are spooled and applied to the join or hash index structure in a single step of the query plan. In particular, when an INSERT … SELECT statement specifies the primary index value of the target row set in its select list or in its WHERE clause, a single-AMP merge step is used to process the insert.

When row-at-a-time updates are performed, the join or hash index structure is updated concurrently, once per base table row.

Join and hash index maintenance is optimized to use localized row hash-level locking whenever possible. Table-level locks are applied when the maintenance is performed using all-AMPs operations such as spool merges.

When the table being updated contains a join or hash index, an EXPLAIN of the UPDATE statement illustrates whether an all-AMPs operation and table-level locking are used at the time of the update, or a single-AMP operation with row hash locking.

The following is a list of the conditions that support single-AMP updating and row hash locking on join or hash indexes when the base table is updated a row-at-a-time. Be aware that these optimizations might not be applied to complicated indexes when cardinality estimates are made with low confidence or when the index is defined on three or more tables with all its join conditions based on nonunique and nonindexed columns.

- When a row-at-a-time INSERT on the base table is being performed, the following restrictions apply:
  - The join index can have a different primary index from the base table.
  - Inserts that specify an equality constraint on the primary index column set of a table with joins between some non-primary index columns of the table and the primary index columns of another table are optimized to use row hash locks. For example,

Given the following join index definition:

```
CREATE JOIN INDEX j2 AS
SELECT x1,x2,x3,y2,y3,z2
FROM t1,t2,t3
WHERE x1=y1
AND y2=z1
PRIMARY INDEX (y3);
```

When a row is inserted into t1 with the following INSERT statement,

```
INSERT INTO t1
VALUES (1,1,1);
```

corresponding j2 rows are materialized by a query like the following:

```
SELECT 1,1,1,y2,y3,z2
FROM t2,t3
WHERE y1=1
AND y2=z1;
```

The plan for the INSERT statement is reported by the following EXPLAIN statement.

```
EXPLAIN
INSERT INTO t1
VALUES (1,1,1);
```

```
*** Help information returned. 24 rows.
*** Total elapsed time was 1 second.

Explanation
-----------------------------------------------------------------------
1) First, we lock a distinct GKA."pseudo table" for write on a
   RowHash to prevent global deadlock for GKA.j2.
2) Next, we lock a distinct GKA."pseudo table" for read on a RowHash
   to prevent global deadlock for GKA.t3.
3) We lock GKA.j2 for write, and we lock GKA.t3 for read.
4) We execute the following steps in parallel.
   1) We do an INSERT into GKA.t1.
   2) We do a single-AMP RETRIEVE step from GKA.t2 by way of the
      primary index "GKA.t2.y1 = 1" with no residual conditions
      into Spool 3, which is redistributed by hash code to all AMPs.
      Then we do a SORT to order Spool 3 by row hash.  The size of
      Spool 3 is estimated with low confidence to be 10 rows.  The
      estimated time for this step is 0.03 seconds.
```

```
5) We do an all-AMPs JOIN step from GKA.t3 by way of a RowHash match
   scan., which is joined to Spool 3 (Last Use).  GKA.t3 and Spool 3
   are joined using a merge join, with a join condition of ("y2 =
   GKA.t3.z1").  The result goes into Spool 2, which is redistributed
   by hash code to all AMPs.  Then we do a SORT to order Spool 2 by
   row hash.  The size of Spool 2 is estimated with no confidence to
   be 1 row.  The estimated time for this step is 0.06 seconds.
6) We do a MERGE into GKA.j2 from Spool 2 (Last Use).
7) Finally, we send out an END TRANSACTION step to all AMPs involved
   in processing the request.
```

As long as the number of rows that qualify `t2.y1=1` (see step 2-1) is within the 10 percent of the number of AMPs threshold and the number of rows resulting from the t2 and t3 join step 5 is also within this threshold, this INSERT statement does not incur any table-level locks.

- For those inserts that specify an equality constraint on non-primary index or that involve joins with non-primary index columns, temporary hash indexes can be created by Teradata to process them in such a way that single-AMP retrieves are used and non-primary index joins are converted to primary index joins. See the following examples:

  Given the following join index definition:

  ```
  CREATE JOIN INDEX j1 AS
  SELECT x1,x2,x3,y2,y3
  FROM t1,t2
  WHERE x2= y1
  PRIMARY INDEX (x1);
  ```

  When a row is inserted into t2 with the following INSERT statement

  ```
  INSERT INTO t2
  VALUES (1,1,1);
  ```

  corresponding j1 rows are materialized by a query like the following:

  ```
  SELECT x1, x2, x3 ,1 ,1
  FROM t1
  WHERE x2=1;
  ```

  The plan for the INSERT statement is:

  ```
  EXPLAIN
  INSERT INTO t2
  VALUES(1,1,1);
  ```

```
 *** Help information returned. 16 rows.
 *** Total elapsed time was 1 second.

Explanation
-----------------------------------------------------------------------
  1) First, we lock a distinct MVIEW."pseudo table" for write on a
     Row hash to prevent global deadlock for MVIEW.J1.
  2) Next, we lock a distinct MVIEW."pseudo table" for read on a
     Row hash to prevent global deadlock for MVIEW.t1.
  3) We lock MVIEW.J1 for write, and we lock MVIEW.t1 for read.
  4) We execute the following steps in parallel.
       1) We do an INSERT into MVIEW.t2.
       2) We do an all-AMPs RETRIEVE step from MVIEW.t1 by way of an
          all-rows scan with a condition of ("MVIEW.t1.x2 = 1") into
          Spool 2, which is built locally on the AMPs.  The size of
          Spool 2 is estimated with no confidence to be 2 rows.  The
          estimated time for this step is 0.03 seconds.
```

```
  5) We do a MERGE into MVIEW.J1 from Spool 2 (Last Use).
  6) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

By defining a single-table non-covering join index on t1.x2, the all-AMP retrieve step 4-2 is replaced by a single-AMP retrieve step 4-2 followed by the all-AMP join step 5 as shown in the explain output below. As long as the number of rows that qualify `t1.x2=1` is within the 10 percent threshold, no table level locks are incurred for this INSERT statement.

```
CREATE JOIN INDEX stji_t1_x2 AS
SELECT x2, ROWID
FROM t1;

EXPLAIN
INSERT INTO t2
VALUES (1,1,1);
```

```
 *** Help information returned. 27 rows.
 *** Total elapsed time was 1 second.

Explanation
-----------------------------------------------------------------------
  1) First, we lock a distinct MVIEW."pseudo table" for write on a
     Row hash to prevent global deadlock for MVIEW.j1.
  2) Next, we lock a distinct MVIEW."pseudo table" for read on a
     Row hash to prevent global deadlock for MVIEW.t1.
  3) We lock MVIEW.j1 for write, and we lock MVIEW.t1 for read.
  4) We execute the following steps in parallel.
       1) We do an INSERT into MVIEW.t2.
       2) We do a single-AMP RETRIEVE step from MVIEW.stji_t1_x2 by way
          of the primary index "MVIEW.stji_t1_x2.x2 = 1" with no
          residual conditions into Spool 3, which is redistributed by
          hash code to all AMPs.  Then we do a SORT to order Spool 3 by
          row hash.  The result spool file will not be cached in memory.
          The size of Spool 3 is estimated with low confidence to be 10
          rows.  The estimated time for this step is 0.03 seconds.
  5) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of a
     Row hash match scan., which is joined to MVIEW.t1.  Spool 3 and
     MVIEW.t1 are joined using a merge join, with a join condition of (
     "Field_1026 = MVIEW.t1.RowID").  The input table MVIEW.t1 will not
     be cached in memory.  The result goes into Spool 2, which is built
     locally on the AMPs.  Then we do a SORT to order Spool 2 by row
     hash.  The result spool file will not be cached in memory.  The
     size of Spool 2 is estimated with index join confidence to be 10
     rows.  The estimated time for this step is 0.33 seconds.
  6) We do a MERGE into MVIEW.j1 from Spool 2 (Last Use).
  7) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

Given the following join index definition:

```
CREATE JOIN INDEX j1 AS
SELECT x1,x2,x3,y2,y3
FROM t1,t2
WHERE x2=y1
PRIMARY INDEX (x1);
```

When a row is deleted from t2, `delete t2 where t2.y1=1,` corresponding j1 rows are materialized by a query like the following:

```
SELECT x1,x2,x3,y2,y3
FROM t1,t2
WHERE x2=y1
AND y1=1;
```

The plan for the DELETE statement is provided by the following EXPLAIN report:

```
EXPLAIN
DELETE t2
WHERE y1=1;
```

```
 *** Help information returned. 26 rows.
 *** Total elapsed time was 1 second.

Explanation
-----------------------------------------------------------------------
  1) First, we lock a distinct MVIEW."pseudo table" for write on a
     RowHash to prevent global deadlock for MVIEW.j1.
  2) Next, we lock a distinct MVIEW."pseudo table" for read on a
     RowHash to prevent global deadlock for MVIEW.t1.
  3) We lock MVIEW.j1 for write, and we lock MVIEW.t1 for read.
  4) We do an all-AMPs RETRIEVE step from MVIEW.t1 by way of an
     all-rows scan with a condition of ("MVIEW.t1.x2 = 1") into Spool 2,
     which is redistributed by hash code to all AMPs.  Then we do a
     SORT to order Spool 2 by row hash.  The size of Spool 2 is
     estimated with low confidence to be 2 rows.  The estimated time for
     this step is 0.03 seconds.
  5) We do a single-AMP JOIN step from MVIEW.t2 by way of the unique
     primary index "MVIEW.t2.y1 = 1" with no residual conditions, which
     is joined to Spool 2 (Last Use).  MVIEW.t2 and Spool 2 are joined
     using a merge join, with a join condition of ("x2 = MVIEW.t2.y1").
     The result goes into Spool 1, which is redistributed by hash code
     to all AMPs.  Then we do a SORT to order Spool 1 by row hash.  The
     size of Spool 1 is estimated with low confidence to be 2 rows.  The
     estimated time for this step is 0.07 seconds.
  6) We execute the following steps in parallel.
       1) We do a MERGE DELETE to MVIEW.j1 from Spool 1 (Last Use).
       2) We do a single-AMP DELETE from MVIEW.t2 by way of the unique
          primary index "MVIEW.t2.y1 = 1" with no residual conditions.
  7) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

In this example, if a single-table non-covering join index is defined on t1.x2, calling it stji_t1_x2, the non-PI-to-PI join that is processed by an all-AMP retrieve from t1 for the qualified rows (step 4) followed by a single-AMP join (step 5) can be processed as follows:

1. a single-AMP join from t2 by way of PI `t2.y1=1` to stji_t1_x2 by way of PI `stji_t1_x2.x2 = 1`, results going into Spool 2. Spool 2 is hash redistributed by t1.rowid and qualifies as a group-AMPs spool.
2. a few-AMPs join back from Spool 2 to t1 on ROWID.

As long as the number of rows resulted from the join `t2.y1=t1.x2 where y1=1 and x2=1` is within the 10 percent threshold, no table level locks are incurred for the DELETE statement.

Given the following join index definition:

```
CREATE JOIN INDEX j2 AS
SELECT x1,x2,x3,y1,y3
FROM t1, t2
WHERE x2=y2;
```

When a row is deleted from t2, `delete t2 where t2.y1=1`, corresponding j2 rows are materialized by a query which involves a non-PI-to-non-PI join:

```
SELECT x1,x2,x3,y1,y3
FROM t1, t2
WHERE x2=y2
AND y1=1;
```

The plan for the DELETE statement is:

```
EXPLAIN
DELETE t2
WHERE y1=1;
```

```
 *** Help information returned. 27 rows.
 *** Total elapsed time was 1 second.

Explanation
-----------------------------------------------------------------------
  1) First, we lock a distinct MVIEW."pseudo table" for write on a
     RowHash to prevent global deadlock for MVIEW.j2.
  2) Next, we lock a distinct MVIEW."pseudo table" for read on a
     RowHash to prevent global deadlock for MVIEW.t1.
  3) We lock MVIEW.j2 for write, and we lock MVIEW.t1 for read.
  4) We do a single-AMP RETRIEVE step from MVIEW.t2 by way of the
     unique primary index "MVIEW.t2.y1 = 1" with no residual conditions
     into Spool 2, which is duplicated on all AMPs.  The input table
     will not be cached in memory, but it is eligible for synchronized
     scanning.  The size of Spool 2 is estimated with high confidence
     to be 20 rows.  The estimated time for this step is 0.16 seconds.
```

```
5) We do an all-AMPs JOIN step from Spool 2 (Last Use) by way of an
   all-rows scan, which is joined to MVIEW.t1.  Spool 2 and MVIEW.t1
   are joined using a product join, with a join condition of (
   "MVIEW.t1.x2 = y2").  The input table MVIEW.t1 will not be cached
   in memory, but it is eligible for synchronized scanning.  The
   result goes into Spool 1, which is built locally on the AMPs.
   Then we do a SORT to order Spool 1 by row hash.  The size of Spool
   1 is estimated with low confidence to be 2 rows.  The estimated
   time for this step is 36.66 seconds.
6) We execute the following steps in parallel.
     1) We do a MERGE DELETE to MVIEW.j2 from Spool 1 (Last Use).
     2) We do a single-AMP DELETE from MVIEW.t2 by way of the unique
        primary index "MVIEW.t2.y1 = 1" with no residual conditions.
7) Finally, we send out an END TRANSACTION step to all AMPs involved
   in processing the request.
-> No rows are returned to the user as the result of statement 1.
```

If a single-table non-covering join index is defined on t1.x2, calling it stji_t1_x2, the (t1.x2=t2.y2) join that is processed by duplicating the qualified t2 row in step 4 to join with t1 in step 5 can be processed as follows:

1. a single-AMP retrieve from t2 by way of PI t2.y1=1 into Spool 2. Spool 2 is hash redistributed by t2.y2 and qualifies as a group-AMPs spool.
2. a few-AMPs join from Spool 2 to stji_t2_x2 on y2=x2, results going into Spool 3. Spool 3 is redistributed by t1.rowid and also qualifies as a group-AMPs spool.
3. a few-AMPs join back from Spool 3 to t1 on ROWID.

As long as the number of rows that qualify the join `t1.x2=t2.y2 where t1.y1=1` is within the 10 percent threshold, no table level locks are incurred for the DELETE statement.

Given the following join index definition:

```
CREATE JOIN INDEX j3 AS
SELECT x1,x2,x3,y2,y3,z2,z3
FROM t1, t2, t3
WHERE x2=y2
AND y1=z1
PRIMARY INDEX (y3);
```

When a row is inserted into t1, `INSERT INTO t1 VALUES (1,1,1),` corresponding j3 rows are materialized by a query of the kind:

```
SELECT 1,1,1,y2,y3,z2,z3
FROM t2,t3
WHERE y2=1
AND y1=z1
```

The plan for the INSERT statement is reported as follows:

```
EXPLAIN
INSERT INTO t1
VALUES (1,1,1);

 *** Help information returned. 26 rows.
 *** Total elapsed time was 1 second.

Explanation
-----------------------------------------------------------------------
  1) First, we lock a distinct MVIEW."pseudo table" for write on a
     Row hash to prevent global deadlock for MVIEW.J3.
  2) Next, we lock a distinct MVIEW."pseudo table" for read on a
     Row hash to prevent global deadlock for MVIEW.t3.
  3) We lock a distinct MVIEW."pseudo table" for read on a Row hash to
     prevent global deadlock for MVIEW.t2.
  4) We lock MVIEW.J3 for write, we lock MVIEW.t3 for read, and we lock
     MVIEW.t2 for read.
  5) We execute the following steps in parallel.
       1) We do an INSERT into MVIEW.t1.
       2) We do an all-AMPs JOIN step from MVIEW.t3 by way of a Row hash
          match scan. with no residual conditions, which is joined to
          MVIEW.t2 with a condition of ("MVIEW.t2.y2 = 1").  MVIEW.t3
          and MVIEW.t2 are joined using a merge join, with a join
          condition of ("MVIEW.t2.y1 = MVIEW.t3.z1").  The input tables
          MVIEW.t3 and MVIEW.t2 will not be cached in memory, but
          MVIEW.t3 is eligible for synchronized scanning.  The result
          goes into Spool 2, which is redistributed by hash code to all
          AMPs.  Then we do a SORT to order Spool 2 by row hash.  The
          result spool file will not be cached in memory.  The size of
          Spool 2 is estimated with low confidence to be 1 row.  The
          estimated time for this step is 0.06 seconds.
  6) We do a MERGE into MVIEW.J3 from Spool 2 (Last Use).
  7) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> No rows are returned to the user as the result of statement 1.
```

If there is a single-table non-covering join index on t2.y2, calling it stji_t2_y2, the all-AMP t2-t3 join in the above EXPLAIN output step 5-2 can be processed as follows:

1. a single-AMP retrieve from stji_t2_y2 by way of PI "stji_t2_y2.y2 = 1" into Spool 2. Spool 2 is hash redistributed by t2.rowid and qualifies as a group-AMPs spool.
2. a few-AMPs join back from Spool 2 to t2 on ROWID, results going in local Spool 3. Spool 3 qualifies as a group-AMPs spool.
3. a few-AMPs join from Spool 3 to t3 on t2.y1=t3.z1.

As long as the number of rows that qualify the join "t2.y1=t3.z1 where t2.y2=1" is within the threshold, no table level locks are incurred for the INSERT statement.

- The index must be a single-table join index or hash index with the following exceptions:
  - The join index primary index is composed of a column set from the target table of the simple insert.
  - The join index primary index is composed of a column set that is joined, either directly or indirectly through transitive closure, to the target table of the simple insert.

  In these cases, multitable join indexes are also optimized for single-AMP merge steps with row hash locking.

  For example, given the following tables, the join index definitions that follow qualify for single-AMP merge optimization:

```
CREATE TABLE t1 (
x1 INTEGER,
x2 INTEGER,
x3 INTEGER)
PRIMARY INDEX (x1);

CREATE TABLE t2 (
y1 INTEGER,
y2 INTEGER,
y3 INTEGER)
UNIQUE PRIMARY INDEX (y1);

CREATE TABLE t3 (
z1 INTEGER,
z2 INTEGER,
z3 INTEGER)
PRIMARY INDEX (z1);

CREATE JOIN INDEX j1 AS
SELECT x1,x2,x3,y2,y3
FROM t1,t2
WHERE x2=y1
PRIMARY INDEX (x1);

CREATE JOIN INDEX j2 AS
SELECT x1,x2,x3,y2,y3,z2,z3
FROM t1,t2,t3
WHERE x2=y1
AND y1=z1
PRIMARY INDEX (x2);
```

The following table indicates several base table inserts, their corresponding single-AMP merge join index inserts, and the qualifying condition that permits the single-AMP optimization:

Base Table Insert	Join Index Insert	Qualifying Condition

`INSERT INTO t1`
`VALUES (1,1,1);`
	`INSERT INTO j1`
	`SELECT 1,1,1,y1,y3`
	`FROM t2, t3`
	`WHERE y1=1;`
	
	`INSERT INTO j2`
	`SELECT 1,1,1,y1,y3,z1,z3`
	`FROM t2,t3`
	`WHERE y1=1`
	`AND y1=z1;`
		The primary index of the join index is composed of a column set from the target table of the simple insert.

`INSERT INTO t2`
`VALUES (1,1,1);`
	`INSERT INTO j2`
	`SELECT x1,x2,x3,1,1,z2,z3`
	`FROM t1,t3`
	`WHERE x2=1`
	`AND 1=z1`
	`AND x2=z1;`
		The primary index of the join index is composed of a column set that is joined directly or indirectly through transitive closure to the target table of the simple insert.

`INSERT INTO t3`
`VALUES (1,1,1);`
	`INSERT INTO j2`
	`SELECT x1,x2,x3,y2,y3,1,1`
	`FROM t1,t2`
	`WHERE x2=y1`
	`AND y1=1;`	

- When a row-at-a-time UPDATE on the base table is being performed, the following restrictions apply:
  - The value for the primary index of the join index must be in the WHERE clause of the statement.
  - The UPDATE cannot change the primary index of the join index.
  - When it is cost effective to access the affected join or hash index rows by means of a NUSI, it is done using row hash locks and a direct update step. If only a few rows are updated (a few-AMPs operation), row hash READ locks are placed on the NUSI subtable for the index rows that are read. Row hash locks are also applied to the base table using the rowID values extracted from the index rows.

- When a row-at-a-time DELETE on the base table is performed, the following restrictions apply:
  - The value for the primary index of the join index must be in the WHERE clause of the DELETE statement.
  - The deleted row must not be from the inner table of an outer join in the CREATE JOIN INDEX statement with the following exceptions:
    - The outer join condition in the join index is specified on a UPI column from the inner table.
    - The outer join condition in the join index is specified on a NUPI column from the inner table.
  - When it is cost effective to access the affected join or hash index rows by means of a NUSI, it is done using row hash locks and a direct delete step. If only a few rows are deleted (a few-AMPs operation), row hash READ locks are placed on the NUSI subtable for the index rows that are read. Row hash locks are also applied to the base table using the rowID values extracted from the index rows.

Under all other conditions, a single-row update causes a table-level WRITE lock to be placed on the hash or join index.

If table-level locks are reported in the EXPLAIN text, then consider using set processing approaches with one or more secondary indexes as an alternative.

Examples of Row-at-a-Time INSERT Maintenance Overhead

Using the guidelines provided in "Join and Hash Index Maintenance Considerations" on page 16-27, you can create a single-table join index that performs well with continuous row-at-a-time inserts to its base table. The maintenance to that join index structure, if TPump were performing continuous inserts, would be one additional single-AMP operation accompanied by a single row-level WRITE lock.

The following EXPLAIN report shows that the maintenance to the join index is performed as a single insert done in parallel with the base table insert.

```
EXPLAIN
INSERT INTO customer
VALUES (11111, 'Name11111', 22, 378.55,'BUILDING');

Explanation
-----------------------------------------------------------------------
  1) First, we execute the following steps in parallel.
     1) We do an INSERT into CAB.customer.
     2) We do an INSERT into CAB.JIsingle.
  2) Finally, we send out an END TRANSACTION step to all AMPs involved in
  processing the request.
```

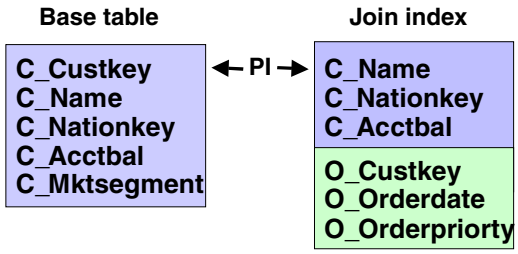
A further example of join index maintenance shows that row-at-a-time inserts into a base table that participates in a multitable join index exert a table-level lock on the join index except for the following cases:

- The primary index of the join index is composed of a column set from the target table of the simple insert.
- The primary index of the join index is composed of a column set that is joined either directly or indirectly through transitive closure to the target table of the simple insert.

In the following example, the primary index of a compressed multitable join index is a subset of the base table being updated, so a single-AMP merge can be used.

CREATE JOIN INDEX

```
CREATE JOIN INDEX JImulti AS
SELECT (o_custkey, c_nationkey,
        c_acctbal, c_name),
       (a_orderdate, o_orderpriority)
FROM orders, customer
WHERE o_custkey = c_custkey
PRIMARY INDEX (c_name);
```

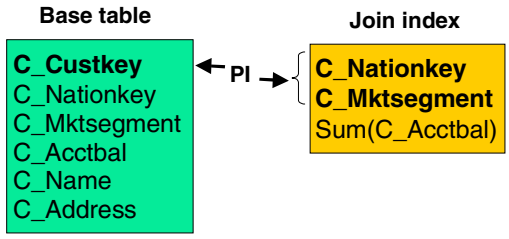


If neither exception is true, then consider using set processing approaches to update the base table (see "Set Processing Alternative" on page 16-39).

An aggregate join index incurs the same maintenance overhead as a multitable join index for row-at-a-time base table inserts. The following example uses a single-table aggregate join index.

CREATE JOIN INDEX

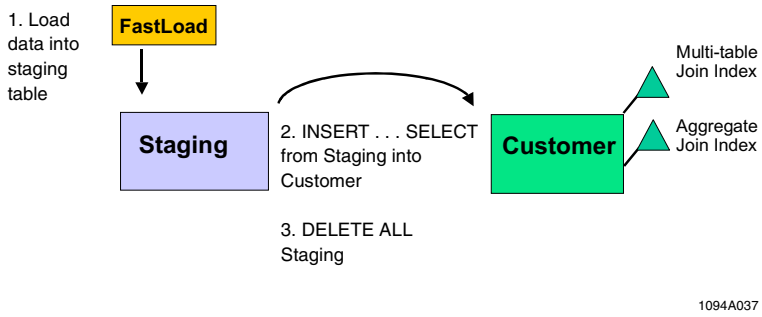
```
CREATE JOIN INDEX cust_JI AS
SELECT c_nationkey, c_mktsegment,
       SUM(c_acctbal) AS accl_bal_sum
FROM customer
GROUP BY c_nationkey, c_mktsegment
PRIMARY INDEX (c_nationkey,
               c_mktsegment;
```



For multitable join indexes, the Optimizer tries to use row hash locks wherever possible, resulting in fewer table-level locks on the join index table. If there are just a few rows in the join index that are impacted, the Optimizer places several row hash locks on just those AMPs and then uses a group-AMPs, rather than an all-AMPs operation.

Set Processing Alternative

For situations where row-at-a-time maintenance imposes table-level locking, consider updating the base table with periodic set processing approaches. The following example shows a set processing approach to updating the customer table that uses frequent, short INSERT … SELECTs into the base table. As the base table updates occur, Teradata also updates any associated multitable and aggregate join indexes.



Section 7:

Appendixes

Appendix A:

Notation Conventions

This appendix describes the notation conventions used in this book.

Throughout this manual, two conventions are used to describe the SQL syntax and code:

- Syntax diagrams, used to describe SQL syntax form, including options. See "Syntax Diagram Conventions" on page A-2.
- Square braces in the text, used to represent options. The indicated parentheses are required when you specify options.

  For example:
  - DECIMAL [(n[,m])] means the decimal data type can be defined optionally:
    - without specifying the precision value *n* or scale value *m*
    - specifying precision (n) only
    - specifying both values (n,m)
    - you cannot specify scale without first defining precision.
  - CHARACTER [(n)] means that use of (n) is optional.

  The values for *n* and *m* are integers in all cases

Symbols from the predicate calculus, set theory, and dependency theory are also used occasionally to describe logical operations. See "Predicate Calculus and Set Theory Notation Used in This Manual" on page A-7 and "Dependency Theory Notation Used in This Manual" on page A-8.

Syntax Diagram Conventions

Notation Conventions

The following table defines the notation used in this section:

<table>
<tr><th>Item</th><th>Definition / Comments</th></tr>
<tr><td>Letter</td><td>An uppercase or lowercase alphabetic character ranging from A through Z.</td></tr>
<tr><td>Number</td><td>A digit ranging from 0 through 9.

Do not use commas when entering a number with more than three digits.</td></tr>
<tr><td>Word</td><td>Variables and reserved words.

<table>
<tr><th>IF a word is shown in . . .</th><th>THEN it represents . . .</th></tr>
<tr><td>UPPERCASE LETTERS</td><td>a keyword.

Syntax diagrams show all keywords in uppercase, unless operating system restrictions require them to be in lowercase.

If a keyword is shown in uppercase, you may enter it in uppercase or mixed case.</td></tr>
<tr><td>lowercase letters</td><td>a keyword that you must enter in lowercase, such as a UNIX command.</td></tr>
<tr><td>lowercase italic letters</td><td>a variable such as a column or table name.

You must substitute a proper value.</td></tr>
<tr><td>lowercase bold letters</td><td>a variable that is defined immediately following the diagram that contains it.</td></tr>
<tr><td>UNDERLINED LETTERS</td><td>the default value.

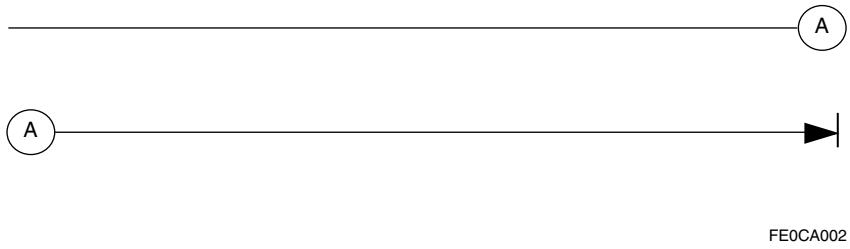
This applies both to uppercase and to lowercase words.</td></tr>
</table>
</td></tr>
<tr><td>Spaces</td><td>Use one space between items, such as keywords or variables.</td></tr>
<tr><td>Punctuation</td><td>Enter all punctuation exactly as it appears in the diagram.</td></tr>
</table>

Paths

The main path along the syntax diagram begins at the left, and proceeds, left to right, to the vertical bar, which marks the end of the diagram. Paths that do not have an arrow or a vertical bar only show portions of the syntax.

The only part of a path that reads from right to left is a loop.

Paths that are too long for one line use continuation links. Continuation links are small circles with letters indicating the beginning and end of a link:



When you see a circled letter in a syntax diagram, go to the corresponding circled letter and continue.

Required Items

Required items appear on the main path:

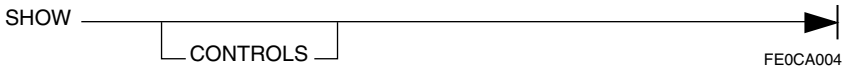


If you can choose from more than one item, the choices appear vertically, in a stack. The first item appears on the main path:



Optional Items

Optional items appear below the main path:



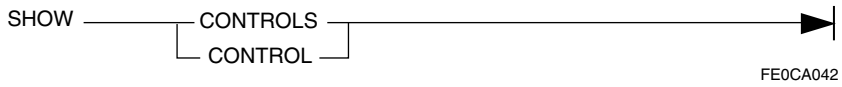
If choosing one of the items is optional, all the choices appear below the main path:



You can choose one of the options, or you can disregard all of the options.

Abbreviations

If a keyword or a reserved word has a valid abbreviation, the unabbreviated form always appears on the main path. The shortest valid abbreviation appears beneath.

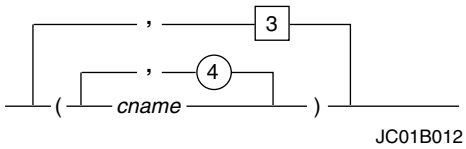


In the above syntax, the following formats are valid:

- SHOW CONTROLS
- SHOW CONTROL

Loops

A loop is an entry or a group of entries that you can repeat one or more times. Syntax diagrams show loops as a return path above the main path, over the item or items that you can repeat.



The following rules apply to loops:

IF . . .	THEN . . .
there is a maximum number of entries allowed	the number appears in a circle on the return path. In the example, you may enter *cname* a maximum of 4 times.
there is a minimum number of entries required	the number appears in a square on the return path. In the example, you must enter at least 3 groups of column names.
a separator character is required between entries	the character appears on the return path. If the diagram does not show a separator character, use one blank space. In the example, the separator character is a comma.
a delimiter character is required around entries	the beginning and end characters appear outside the return path.

IF . . .	THEN . . .
	Generally, a space is not needed between delimiter characters and entries. In the example, the delimiter characters are the left and right parentheses.

Excerpts

Sometimes a piece of a syntax phrase is too large to fit into the diagram. Such a phrase is indicated by a break in the path, marked by | terminators on either side of the break. A name for the excerpted piece appears between the break marks in boldface type.

The named phrase appears immediately after the complete diagram, as illustrated by the following example.

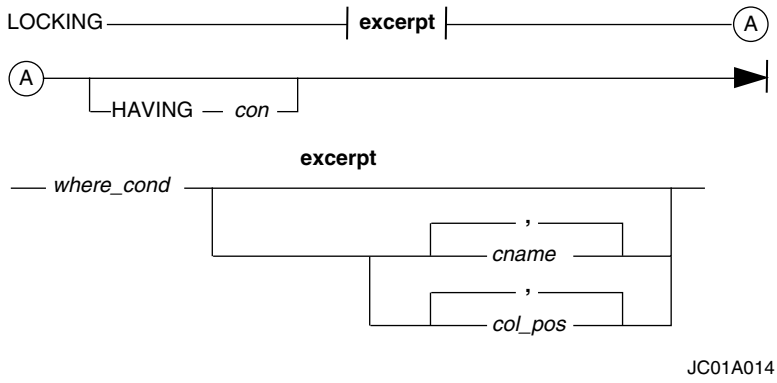


Table Column Definition and Constraint Abbreviations

The following set of abbreviations is used throughout this manual to denote various characteristics of, or constraints on, table columns.

Abbreviation	Definition
FK	Foreign key
NC	No changes allowed
ND	No duplicates allowed
NN	No nulls allowed
PK	Primary key
SA	System-assigned value
UA	User-assigned value

Predicate Calculus and Set Theory Notation Used in This Manual

Relational databases are based on the theory of relations as developed in set theory. Predicate calculus is often the most unambiguous way to express certain relational concepts.

Occasionally this manual uses the following predicate calculus and set theory notation to explain concepts.

This symbol ...	Represents this phrase ...
Predicate Calculus Notation	
iff	If and only if
∀	For all
∃	There exists
Set Theory Notation	
∅	Empty set
∪	Union
∩	Intersection

Dependency Theory Notation Used in This Manual

The decomposition of relations during normalization relies on the notation of dependency theory.

Occasionally this manual uses the following dependency theory notation to explain concepts.

This symbol ...	Represents this phrase ...
$\rightarrow$	Functionally determines
$\overline{\rightarrow}$	Does not functionally determine
$\Rightarrow$	Multivalued determines

Appendix B:

Teradata Database Maxima

This appendix provides the following Teradata database limits.

- System maxima
- Database maxima
- Session maxima

The reported maxima apply only to Teradata server software. Platform-dependent client maxima are not documented in this appendix.

System Maxima

The system specifications in the following table apply to an entire Teradata database configuration.

Parameter	Value
Maximum number of databases and users	4.2×10^9
Total data capacity	43 Gbytes per AMP or optional 120 Gbytes per AMP
Maximum number of active transactions	2 048
Maximum data format descriptor size	30 characters
Maximum error message text size in failure parcel	255 bytes
Maximum number of sectors per data block	255
Maximum data block size	130 560 bytes
Maximum number of sessions per PE	120
Maximum number of gateways per node	1
Maximum number of sessions per Gateway	Tunable. 1 200 maximum certified
Maximum number of parcels in one message	256
Maximum message size	Approximately 65 000 bytes **Note:** This limit applies to messages to and from client systems and to some internal Teradata messages.
Maximum number of PEs per system	1 024
Maximum number of AMP and PE vprocs, in any combination, per system	16 384

Database Maxima

The database specifications in the following table apply to a single Teradata database. The values presented are maxima for their respective parameters individually and not in combination.

Parameter	Value
Maximum number of journal tables per database	1
Maximum number of data tables per database	4.2×10^9
Maximum number of columns per table or view	2 048
Maximum number of columns created over the life of a table	2 560
Maximum number of rows per table	Limited by disk capacity
Maximum row size	Approximately 64 000 bytes
Maximum field size	Approximately 64 000 bytes
Maximum database, user, table, view, macro, index, constraint, or column name size	30 bytes
Maximum number of columns per primary or secondary index	64
Maximum SQL request size	1 Mbyte (Includes SQL statement text, USING data, and parcel overhead)
Maximum SQL response size	Approximately 64 000 bytes
Maximum expanded text size for macros and views	2 Mbytes
Maximum SQL title size	60 characters
Maximum number of characters in a string constant	32 000
Maximum number of columns per DML statement ORDER BY clause	16
Maximum number of primary indexes per table	1
Maximum number of partitions for a partitioned primary index	65 535
Maximum number of secondary, hash, and join indexes, in any combination, per table	32

Parameter	Value
Maximum number of columns and indexes on which statistics can be recollected for a table.	512
Maximum number of parameters specified in a macro	2 048
Maximum number of table-level constraints per table	100
Maximum number of referential constraints per table	64
Maximum number of tables that can be joined per query block	64
Maximum number of subquery nesting levels per query	64
Maximum number of columns in foreign and parent keys	16
Maximum number of columns referenced per single table in a hash or join index	64
Maximum number of columns referenced in the fixed part of a compressed hash or join index	64
Maximum number of columns referenced in the repeating part of a compressed hash or join index	64
Maximum number of columns in an uncompressed join index	2048
Maximum number of columns in a compressed join index	128
Maximum number of compressed values per column	255 plus nulls
Maximum number of fields in a USING row descriptor	2 550

Session Maxima

The session specifications in the following table apply to a single session.

Parameter	Value
Active request result spool files	16
Parallel steps Parallel steps can be used to process a request submitted within a transaction (which may be either explicit or implicit). The maximum number of steps generated per request is determined as follows:	
• Per request, if no channels **Note:** Channels are not required for a prime key request with an equality constraint.	20 steps
• A request that involves redistribution of rows to other AMPs, such as a join or an INSERT … SELECT	Requires 4 channels
• A request that does not involve row distribution	Requires 2 channels
Maximum number of materialized global temporary tables per session	2 000
Maximum number of volatile tables per session	1 000

Appendix C:

Designing With Task-Oriented Profiles

This appendix demonstrates how you can use task-oriented views to isolate users from direct contact with the database.

The following topics are described.

- "Isolating Users from the Database Using Views" on page C-2
- "Flow Diagram of the Database Structure" on page C-3
- "Policies and Procedures" on page C-4

Isolating Users from the Database Using Views

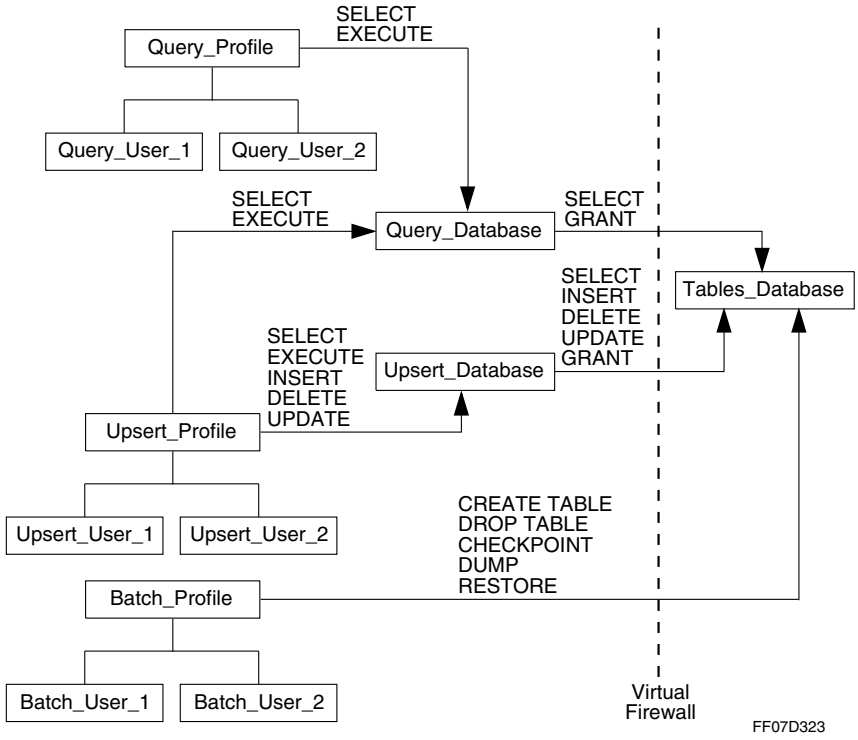
This topic presents a well-designed database structure based on a structure formulated and used by a Teradata customer.

The design incorporates different task-oriented profiles that access only a series of views and macros defined within several task-oriented databases.

Only these task-oriented access definition databases can access the base tables defined for the database. This design structure builds a virtual firewall between all users and the base tables they access.

Flow Diagram of the Database Structure

The following graphic illustrates the work flows and individual task-oriented profiles and database objects defined for this database structure.



Policies and Procedures

Introduction

This topic describes the policies and procedures for using the database structure illustrated in "Flow Diagram of the Database Structure" on page C-3.

Note that Tables_Database contains only base tables and their associated index subtables.

End User Profiles

The example database defines three types of end user profile. The following table describes those types.

Profile	Description	Permitted Statements and Commands
Query_Profile	Defines access rights to the views, macros, and stored procedures defined in Query_Database.	• EXECUTE • SELECT
Upsert_Profile	Defines access rights to the views, macros, and stored procedures defined in Upsert_Database.	• CALL • DELETE • EXECUTE • INSERT • SELECT • UPDATE
Batch_Profile	Defines access rights to the tables defined in Tables_Database.	• CHECKPOINT • CREATE TABLE • DROP TABLE • DUMP • RESTORE

Rules for End User Profiles

The following rules apply to all user profiles except Batch_Profile.

- All users belong to one or more task profiles.
- Users inherit their access rights from the task profiles to which they belong.
- A user can belong to more than one task profile.
- Access rights are granted at DATABASE or USER levels *only*.
- Query_Profile and Upsert_Profile only have access to databases that contain macros, stored procedure definitions, and views exclusively.
- Batch_Profile is the only profile that permits the base tables in Tables_Database to be accessed directly.

Rules for Batch_Profile

The following rules apply only to Batch_Profile.

- Users inherit their access rights from Batch_Profile.
- Access rights are granted at Database or User levels *only*.
- Batch_Profile has direct access to the base tables in Tables_Database because of the functions performed using it.
- Batch_Profile users can perform the following functions against Tables_Database
  - CHECKPOINT transactions
  - CREATE TABLEs
  - DROP TABLEs
  - DUMP databases
  - RESTORE databases

Rules for Query_Database

Note that Query_Database objects are organized in such a way that the entire database can be archived in one operation. You can also archive Upsert_Database in the same operation if you choose to do so.

The following rules apply to Query_Database.

- Both Query_Profile and Upsert_Profile users can perform the following functions against Query_Database.
  - CALL permitted stored procedures
  - EXECUTE permitted macros
  - SELECT rows from Tables_Database objects through views
- The database contains only views, stored procedure definitions, and macros that permit an end user to query or grant access rights on Tables_Database objects indirectly.

Rules for Upsert_Database

Note that Upsert_Database objects are organized in such a way that the entire database can be archived in one operation. You can also archive Query_Database in the same operation if you choose to do so.

The word *upsert* derives from *up*date-in*sert* and implicitly refers to delete as well.

The following rules apply to Upsert_Database.

- Only Upsert_Profile users exclusively can perform the following functions against Upsert_Database.
  - CALL permitted stored procedures
  - DELETE rows in Tables_Database objects through views
  - EXECUTE permitted macros
  - INSERT rows into Tables_Database objects through views
  - SELECT rows from Tables_Database objects through views
  - UPDATE rows in Tables_Database objects through views

The database contains only views, stored procedure definitions, and macros that permit an end user to perform the following actions on Tables_Database objects indirectly.

- DELETE rows
- INSERT rows
- SELECT rows
- UPDATE rows
- GRANT access rights on database objects

Appendix D:

Summary Physical Design Scenario

This appendix presents an overview of the physical design process.

Physical Design Scenario

Introduction

The following scenario assumes that you are pursuing a 3NF logical model for your database schema.

Prerequisites for the Process Review

Use the following checklist to ensure you are ready to enter the physical design phase of your project.

√	Physical Design Phase Entry Criteria
	Fully documented 3NF logical model
	Fully documented column and table demographics
	Good understanding of Teradata architecture-specific physical design principles

Process Review

The following table lists the high-level steps in the physical design process.

Step	Action
1	Identify all index candidates.
2	Review table demographics, then verify or modify index choices.
3	Evaluate denormalization possibilities, then verify or modify index choices.
4	Evaluate derived data, then verify or modify index choices.
5	Implement the system and initiate production processing.
6	Track changing demographics.
7	Review indexes periodically to ensure they are still the best choices.
8	Perform appropriate EXPLAIN statements periodically to track whether defined indexes are being selected for query processing by the Optimizer.
9	Reevaluate table use.
10	Document any changes.
11	Archive older versions of your findings for use in analyzing trends.
12	Modify schedules and other temporal choices for collecting column statistics to ensure the Optimizer continues to generate optimal query plans.

Appendix E:

Supporting Information for Join Index Performance Analysis

This appendix provides data to support the join index cost/benefit and performance analyses presented in Chapter 11: "Join and Hash Indexes."

The following types of information are presented in this appendix.

- Configuration of the test system used to generate the figures presented.
- DDL for the tables and join indexes used in the analysis.
- DML for the queries, inserts, updates, and deletions used in the analysis.

Test System Configuration

Hardware Configuration

The system used to generate these numbers had the following general configuration.

- Two 550 MHz 4-processor nodes with 1 MB cache and 2 GB memory per node
- RAID1 (mirrored) disk arrays with a segment size of 968
- BYNET 2
- 20 AMPs, each with 2 LUNs (4 physical disks)
- 4 network-attached PEs

Tunable Parameter Settings

The following table lists the tunable parameter settings used on the test system throughout the test cycle.

Parameter	Setting	Comment
ReadAhead flag setting	TRUE	Disk I/O
DBS Don't Cache flag setting	TRUE	Use DBC Cache Threshold setting
DBC Cache Threshold	10	Default setting
FSG Cache Percent	80	Default setting
RSS logging settings	• SPMA • IPMA • SVPR	15 second logging interval
Datablock size	64 KB	
Free Space Percent	0	
MiniCylPack	10 cylinders	Low Cylinder

Test Table Information

The join indexes were typically created on two base tables, A and B. The tables have the following characteristics.

- Row size of 100 bytes (except where row size is manipulated as a variable)
- Initial block size of 65 024 bytes
- Base and join index table cardinalities

Type of Join Index	Cardinality of Join Index Table	Cardinality of Base Table 100A	Cardinality of Base Table 100B
Simple	45×10^6	45×10^6	60×10^6
Aggregate	45×10^2		

Test Suites

Introduction

The join index performance numbers presented in Chapter 11: "Join and Hash Indexes" are derived from three different test suites. The test suites are described briefly in the following topics.

Test Suite 1

Suite 1 is designed to measure the maintenance costs of different join indexes. The test suite performs various maintenance activities on a table with different kinds of join index. The maintenance activities examined include inserts, deletes, and updates and different rates.

Suite 1 was also performed for an in-place aggregate join index with varying row sizes.

The suite also examined the costs of creating the various kinds of join index used in the analysis.

Test Suite 2

Suite 2 is designed to measure the benefits accrued by a set of queries performed with and without various types of join index.

This suite also examined the costs of creating the various kinds of join index used in the analysis.

Test Suite 3

Suite 3 was restricted to examining the relative performance advantages of various methods of inserting rows into a table with an in-place aggregate join index.

Test Table DDL

Introduction

This topic presents the DDL for the tables used in this analysis.

CREATE TABLE DDL

The CREATE TABLE statement for the base table table100A is given below.

```
CREATE SET TABLE jim.table100a,
NO FALLBACK,
DATABLOCKSIZE=65024
 (pkey_field         INTEGER,
  int2_field         INTEGER,
  int3_field         INTEGER,
  int4_field         INTEGER,
  int5_field         INTEGER,
  int6_field         INTEGER,
  int7_field         INTEGER,
  int8_field         INTEGER,
  int9_field         INTEGER,
  char10_1_100field  CHARACTER(10),
  char20_2_100field  CHARACTER(20),
  char20_3_100field  CHARACTER(20))
UNIQUE PRIMARY INDEX (pkey_field);
```

Similar statements were used for the other base tables table100B, table100C and table100D. Tables table100C and table100D were used only for four table joins. The master table for 100 byte rows also uses a similar create table statement.

Test Join Index DDL

Introduction

This topic presents the DDL for the join indexes used in this analysis.

In-Place Simple Join Index

This CREATE JOIN INDEX statement creates a simple in-place join index.

```
CREATE JOIN INDEX ji_tpg_ji AS
SELECT table100a.int3_field, table100b.int2_field
FROM table100a, table100b
WHERE table100a.pkey_field=table100b.pkey_field;
```

In-Place Aggregate Join Index

This CREATE JOIN INDEX statement creates an aggregate in-place join index. Notice that the join condition is between the primary index of table100A and the primary index of table100B. Therefore, there is no need to redistribute any rows.

```
CREATE JOIN INDEX aji_tpg_ji AS
SELECT table100a.int3_field, SUM(table100b.int2_field
 (FLOAT)) AS sumb
FROM table100a, table100b
WHERE table100a.pkey_field = table100b.pkey_field
GROUP BY 1;
```

The only difference between this DDL and the DDL for an in-place simple join index is the presence of a SUM field and the corresponding GROUP BY clause in the create statement for the simple join index.

Because there are 10 000 rows for every value of the int3_field (the grouping key), there are 4 500 rows in the Aggregate Join Index.The following text is an EXPLAIN fragment for this DDL. Notice that there is no redistribution of rows and the tables are joined in place.

```
17) … 4) We do an all-AMPs JOIN step from JIM.TABLE100B by way of a
 RowHash match scan with no residual conditions, which is
 joined to JIM.TABLE100A. JIM.TABLE100B and JIM.TABLE100A are
 joined using a merge join, with a join condition of (
 "JIM.TABLE100A.PKEY_FIELD = JIM.TABLE100B.PKEY_FIELD"). …
```

Foreign Aggregate Join Index

This CREATE JOIN INDEX statement creates a foreign aggregate join index. Notice that the join condition is between the primary index of table100A and a column that is *not* the primary index of table100B. Therefore there is no need to redistribute table100A, but table100B needs to be redistributed.

```
CREATE JOIN INDEX aji_tpg_jf AS
SELECT table100a.int3_field, SUM(table100b.int2_field
 (FLOAT)) AS sumb
FROM table100a, table100b
WHERE table100a.pkey_field = table100b.int2_field
GROUP BY 1;
```

The following text is an EXPLAIN fragment for this DDL. Notice that in step 17.4 we redistribute the rows of table100B. You can also see from step 18 that table100A stays in place; therefore, in a foreign join only one table is redistributed.

```
17) …4) We do an all-AMPs RETRIEVE step from JIM.TABLE100B by way of
 an all-rows scan with no residual conditions into Spool 3,
 which is redistributed by hash code to all AMPs. Then we do a
 SORT to order Spool 3 by row hash. …
 18) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of a
 RowHash match scan, which is joined to JIM.TABLE100A. Spool 3 and
 JIM.TABLE100A are joined using a merge join, with a join condition
 of ("JIM.TABLE100A.PKEY_FIELD = INT2_FIELD"). …
```

Ad Hoc Aggregate Join Index

This CREATE JOIN INDEX statement creates an ad hoc aggregate join index. Notice that the join condition is between a column that is *not* the primary index of table100A and a column that is *not* the primary index of table100B. Therefore both table100A and table100B need to be redistributed.

```
CREATE JOIN INDEX aji_tpg_ja AS
SELECT table100a.int3_field, SUM(table100b.int2_field
 (FLOAT)) AS sumb
FROM table100a, table100b
WHERE table100a.int4_field = table100b.int5_field
GROUP BY 1;
```

The following text is an EXPLAIN fragment for this DDL. Notice that in step 17.4 table100A is redistributed. Notice also that in step 17.5 it is table100B that is being redistributed. This demonstrates that both tables are distributed when you perform an ad hoc join.

```
17) … 4) We do an all-AMPs RETRIEVE step from JIM.TABLE100A by way of
 an all-rows scan with no residual conditions into Spool 3,
 which is redistributed by hash code to all AMPs. Then we do a
 SORT to order Spool 3 by row hash. …
 5) We do an all-AMPs RETRIEVE step from JIM.TABLE100B by way of
 an all-rows scan with no residual conditions into Spool 4,
 which is redistributed by hash code to all AMPs. Then we do a
 SORT to order Spool 4 by row hash. …
 18) We do an all-AMPs JOIN step from Spool 3 (Last Use) by way of a
 RowHash match scan, which is joined to Spool 4 (Last Use). Spool
 3 and Spool 4 are joined using a merge join, with a join condition
 of ("INT4_FIELD = INT5_FIELD"). …
```

Single-Table Simple Join Index

This CREATE JOIN INDEX statement creates a single-table simple join index.

```
CREATE JOIN INDEX aji_tpg_ja AS
SELECT table100a.int3_field, table100a.int2_field
FROM table100a;
```

Single-Table Aggregate Join Index

This CREATE JOIN INDEX statement creates a single-table aggregate join index.

```
CREATE JOIN INDEX aji_tpg_jis AS
SELECT table100a.int3_field, SUM(table100a.int2_field
 (FLOAT)) AS sumb
FROM table100a
GROUP BY 1;
```

Four-Table In-Place Aggregate Join Index

This CREATE JOIN INDEX statement creates a four-table in-place aggregate join index.

```
CREATE JOIN INDEX aji_tpg_ji4t AS
SELECT table100a.int3_field, SUM(table100b.int6_field
 (FLOAT)) AS sumb,
  SUM(table100c.int6_field (FLOAT)) AS sumc,
  SUM(table100d.int6_field (FLOAT)) AS newint6_field
FROM table100a, table100b, table100c, table100d
WHERE table100a.pkey_field = table100b.pkey_field
AND table100b.pkey_field = table100c.pkey_field
AND table100c.pkey_field = table100d.pkey_field
GROUP BY 1;
```

In-Place Outer-Join Join Index

This CREATE JOIN INDEX statement creates an in-place outer-join join index.

```
CREATE JOIN INDEX ji_tpg_jio AS
SELECT table100b.pkey_field, table100a.int3_field,
 table100a.int6_field
FROM table100a RIGHT OUTER JOIN table100b
ON table100a.pkey_field = table100b.pkey_field;
```

Test Join Index DML

Introduction

This topic presents the DML statements used to test the join indexes analyzed in this study.

After each insert or delete operation, the tables are restored to their original cardinality to ensure a constant baseline for the tests.

INSERT 0.2 Percent

This INSERT statement inserts 90 000 rows into table100a.

```
INSERT INTO jim.table100a
SELECT *
FROM master.table100
WHERE master.table100.pkey_field
BETWEEN 45000001 AND 45090000;
```

The column named pkey_field is unique. Its value begins at 1 and increments by 1 for as many rows as you have. In this case, there are 45 million rows in table100A, and the INSERT statement inserts an additional 90 000 rows. Therefore, the insert fraction percentage is 0.2 percent ((90 000/45 000 000)(100)).

INSERT 1 Percent

This INSERT statement inserts 450 000 rows into table100a.

```
INSERT INTO jim.table100a
SELECT *
FROM master.table100
WHERE master.table100.pkey_field
BETWEEN 45000001 AND 45450000;
```

In this case, the insert fraction is 1 percent ((450 000/45 000 000)(100)).

INSERT 4 Percent

This INSERT statement inserts 1 800 000 rows into table100a.

```
INSERT INTO jim.table100a
SELECT *
FROM master.table100
WHERE master.table100.pkey_field
BETWEEN 45000001 AND 46800000;
```

In this case, the insert fraction is 4 percent ((1 800 000/45 000 000)(100)).

INSERT Right Table 0.2 Percent

This INSERT statement inserts 90 000 rows into table100b, which is the right table in the right outer join.

```
INSERT INTO jim.table100b
SELECT *
FROM master.table100
WHERE master.table100.pkey_field
BETWEEN 45000001 AND 45090000;
```

This was tested only for the outer in-place join index, because the other cases should not produce different values than the left table insert operation.

UPDATE 0.2 Percent

This UPDATE statement updates 90 000 rows in table100a.

```
UPDATE jim.table100a
SET int3_field = int3_field + 2
WHERE pkey_field > 44910000;
```

Here the number of rows being updated is (45 000 000 – 44 910 000), or 90 000 rows. Therefore, two percent (90,000/45,000,000) of the rows are updated.

To update 1 percent of the rows, modify the WHERE clause in the this UPDATE statement to `pkey_field > 44550000`.

To update 4 percent of the rows, modify the WHERE clause as indicated in the following UPDATE statement.

```
UPDATE jim.table100a
SET int3_field = int3_field + 2
WHERE pkey_field > pkey_field > 43200000;
```

UPDATE 0.2 Percent Using a Non-Primary Index Column

This UPDATE statement updates 90 000 rows in table100a without specifying its primary index in the WHERE clause. Notice that int2_field is *not* the primary index of this table.

```
UPDATE jim.table100a
SET int3_field = int3_field + 2
WHERE int2_field > 44910000;
```

DELETE 0.2 Percent

This DELETE statement deletes 90 000 rows from table100a.

```
DELETE
FROM jim.table100a
WHERE pkey_field > 44910000;
```

You can modify the WHERE clause in this DELETE statement with the same values used for the update operation to delete 1 percent and 4 percent of the rows from table100a.

Test SELECT Statements

The following set of SELECT statements was used to test the relative benefit of several types of join index. The same queries were run twice: first with the specified join index and then without it.

SELECT to Measure the Benefit of an In-Place Join Index

This SELECT statement tests the efficiency of an in-place simple join index.

```
SELECT table100a.pkey_field, table100b.pkey_field AS sumb,
 table100a.int6_field
FROM table100a, table100b
WHERE table100a.pkey_field = table100b.pkey_field;
```

SELECT to Measure the Benefit of an In-Place Aggregate Join Index

This SELECT statement tests the efficiency of an in-place aggregate join index.

```
SELECT table100a.int3_field, SUM(table100b.int2_field
 (FLOAT)) AS sumb
FROM table100a, table100b
WHERE table100a.pkey_field = table100b.pkey_field
GROUP BY 1;
```

SELECT to Measure the Benefit of a Foreign Aggregate Join Index

This SELECT statement tests the efficiency of a foreign aggregate join index.

```
SELECT table100a.int3_field, SUM(table100b.int2_field
 (FLOAT)) AS sumb
FROM table100a, table100b
WHERE table100a.pkey_field = table100b.int2_field
GROUP BY 1;
```

SELECT to Measure the Benefit of an Ad Hoc Aggregate Join Index

This SELECT statement tests the efficiency of an ad hoc aggregate join index.

```
SELECT table100a.int3_field, SUM(table100b.int2_field
 (FLOAT)) AS sumb
FROM table100a, table100b
WHERE table100a.int4_field = table100b.int5_field
GROUP BY 1;
```

SELECT to Measure the Benefit of an Single-Table Simple Join Index

This SELECT statement tests the efficiency of a single-table simple join index.

```
SELECT table100a.int3_field, table100a.int2_field
FROM table100a;
```

SELECT to Measure the Benefit of a Single-Table Aggregate Join Index

This SELECT statement tests the efficiency of a single-table aggregate join index.

```
SELECT table100a.int3_field, SUM(table100a.int2_field
 (FLOAT)) AS sumb
FROM table100a
GROUP BY 1;
```

SELECT to Measure the Benefit of a 4-Table In-Place Aggregate Join Index

This SELECT statement tests the efficiency of a 4-table in-place aggregate join index.

```
SELECT table100a.int3_field, SUM(table100b.int6_field
 (FLOAT)) AS sumb, SUM(table100c.int6_field (FLOAT)) AS sumc,
 SUM(table100d.int6_field (FLOAT)) AS newint6_field
FROM table100a, table100b, table100c, table100d
WHERE table100a.pkey_field = table100b.pkey_field
AND table100b.pkey_field = table100c.pkey_field
AND table100c.pkey_field = table100d.pkey_field
GROUP BY 1;
```

SELECT to Measure the Benefit of an In-Place Right Outer Join Index

This SELECT statement tests the efficiency of an in-place right outer join index.

```
SELECT table100b.pkey_field, table100a.int3_field,
 table100a.int6_field
FROM table100a
RIGHT OUTER JOIN table100b ON table100a.pkey_field =
 table100b.pkey_field;
```

Appendix F:

Sample Worksheet Forms

This appendix contains a complete set of forms that you can photocopy and use for your database design projects.

The following forms are included.

- Domains on page F-2
- Constraints on page F-3
- System on page F-4
- Report/Query Analysis Form on page F-5
- Table Access Summary By Column on page F-6
- Table on page F-7
- Row Size on page F-8

Domains

ELDM Page:

Page: Of

System:

Domain Name	Domain Description	Data Type	Max Bytes	Print Format	Constraint Number

Data Types

Byte	B(n)	Interval Month	IMO	Long Graphic Varying	LVG
Byteint	I1	Interval Day	ID	Numeric	NUM(n,m)
Character	C(n)	Interval Day To Hours	IDH	Real	R
Character Varying	CV(n)	Interval Day To Minute	IDM	Smallint	I2
Date	D	Interval Day To Second	IDS	Time	T
Decimal	DEC(n,m)	Interval Hours	IH	Timestamp	TS
Double Precision	DP	Interval Hour To Minute	IHM	Time With TImeZone	TTZ
Float	F(n)	Interval Hour To Second	IHS	Timestamp With TImeZone	TSTZ
Graphic	G(n)	Interval Minute	IM	Varbyte	VB
Integer	I	Interval Minute To Second	IMS	Varchar	VC(n)
Interval Year	IY	Interval Second	IS	Vargraphic	VG(n)
Interval Year To Monts	IYM	Long Character Varying	LVC		

FF07D332

Constraints

ELDM Page: _____

Page: _____ Of _____

System: _____

Constraint Number	Constraint Description

FF07D333

System

Page: _____ Of _____

ELDM Page: _____

System: _____

System Name: _____

System Description: _____

_____

_____

Application ID	Application Name	User Contacts

FF07D335

Report/Query Analysis Form

Frequency or Importance Ranking: _____

Report/Query Description: _____

_____

_____

Table: **Number Of Output Rows:**

Input Value Or Source	
Access Column(s)	

Table: **Number Of Output Rows:**

Input Value Or Source	
Access Column(s)	

Table: **Number Of Output Rows:**

Input Value Or Source	
Access Column(s)	

Table: **Number Of Output Rows:**

Input Value Or Source	
Access Column(s)	

FF07D337

Table Access Summary By Column

Page:_____ Of _____

ELDM Page: _____

System: _____

Table Name: _____

Column Names: _____ _____

_____ _____

_____ _____

Value Accesses			Join Accesses		
Transaction ID	**Rows**	**Freq**	**Transaction ID**	**Rows**	**Freq**
Value Accesses Totals:			**Join Accesses Totals:**		

FF07D334

Table

Page: _____ Of _____

ELDM Page: _____

System: _____

Table Name	Table Type	Cardinality	Data Protection

Column Name	
PK/FK	
Constraint Number	
Value Access Frequency	
Joint Access Frequency	
Joint Access Rows	
Distinct Values	
Maximum Rows/Value	
Maximum Rows Null	
Typical Rows/Value	
Change Rating	
PI/SI	
Sample Data	

FF07D336

Row Size Calculation Form

Table Name:

Variable Data Detail

Column Name	Type	Max	Avg
	SUM(a)		

Data Type	Number of Columns	Sizing Factor	Total
Byteint	*	1	=
Smallint	*	2	=
Integer	*	4	=
Date	*	4	=
Time(ANSI)	*	6	=
Time With Time Zone	*	8	=
Timestamp	*	10	=
Timestamp With Time Zone	*	12	=
Interval Year	*	2	=
Interval Year to Month	*	4	=
Interval Month	*	2	=
Interval Month to Day	*	2	=
Interval Day	*	2	=
Interval Day to Hour	*	4	=
Interval Day to Minute	*	8	=
Interval Day to Second	*	10	=
Interval Hour	*	2	=
Interval Hour to Minute	*	4	=
Interval Hour to Second	*	8	=
Interval Minute	*	2	=
Interval Minute to Second	*	6	=
Interval Second	*	6	=
Dec 1-2	*	1	=
3-4	*	2	=
5-9	*	4	=
10-18	*	8	=
Float	*	8	=
Fixed		SUM(n)	=
Variable		SUM(a)	=
		Logical Size	=
		Overhead	= 14
Variable Column Offsets			=
______ Compress Columns			
______ Nullable Columns			
______ /8 (Quotient only)			=
		Physical Size**	=

SUM(a)=SUM of the AVERAGE number of bytes expected for the variable column.
SUM(n)=SUM of the CHAR and GRAPHIC column bytes.
**Round up to an even number of bytes

FF07D347

Index

Numerics

1NF, definition 5–13
2NF, definition 5–15
3NF, definition 5–18
4NF, definition 5–23
5NF, definition 5–26
80:20 rule 14–25

A

Abbreviations
  key to architectural flow diagram 2–9
Activities modeling
  process activities 2–12
  transition design phase 2–12
Ad hoc join index
  definition 11–84
Aggregate operations
  nonunique secondary index 10–25
Aggregates, null and 13–19
Alternate key, definition 5–4, 8–10
ANSI DateTime, null and 13–19
ANSI/SPARC architecture
  abbreviations 2–9
  detailed 2–7
  discussion 2–6
  high-level 2–6
  levels
    external 2–6, 2–7
    internal 2–7, 2–8
  references 2–9
ANSI/SPARC architecture levels
  conceptual 2–7, 2–8
Application form
  completing 7–24
  information in 7–24
  purpose 7–24
Applications
  definition 7–22
Archive and Restore
  restrictions with hash index 11–42
  restrictions with join index 11–42
Arithmetic function, nulls and 13–18
Arithmetic operators, nulls and 13–18
ATM
  and referential integrity 5–37
  application form 7–24
  constraints form 7–20
  domains form 7–14, 7–17
  forms set F–1
  goals 7–1
  logical design 7–1
  process activity 7–2
  purpose 7–1
  report/query analysis form 7–26
  system form 7–22
  table form 7–33
ATM terms
  columns 7–5
  constraints 7–3
  domains 7–3
  foreign key 7–6
  normalization 7–6
  primary key 7–5
  rows 7–4
  tables 7–4
Attributes
  definition 4–5, 5–4
  derivative 4–6
  foreign key 4–5
  mapping 6–17
  nonkey 4–5
  primary key 4–5
  see also Column attributes

B

Backups, designing database for 15–17
Batch referential integrity 5–41
BCNF, definition 5–19
Benefit of a join index 11–73, 11–99
  computing 11–99
  definition 11–72
Binary relationship 4–9
Bit mapping
  complex conditional expressions 10–21
  computing for NUSIs 10–22
  restrictions on 10–20
  using with NUSIs 10–20

Boot disk
contents 15–3
Disk 0 15–3
file systems 15–3
Boyce-Codd normal form 5–19
Break-even percentage
definition 14–45
how to calculate 14–45
Break-even point, definition 14–44
BTEQ
output spool space 14–96
query manager 2–7
Business rules
and triggers 7–20
See also Constraints
BYTE data type 7–8
Byte vs. octet
presence bits 14–37
BYTEINT data type 7–8

C

Candidate key, definition 5–4, 8–10
Capacity planning
considerations
BYTE data type 14–76
CHARACTER data types 14–82
DateTime data types 14–76
GRAPHIC data types 14–83
INTEGER data type 14–75
INTERVAL data types 14–77
NUMERIC data types 14–76
reference index 8–25
sizing NUSIs 14–92
sizing RIs 14–94
sizing rows 14–84
sizing USIs 14–91
sizing base tables 14–89
sizing hash indexes
Space requirements
hash indexes 14–89
sizing join indexes
Space requirements
join indexes 14–89
Capacity planning, considerations
sizing columns 14–12
Cardinality
in a relationship 4–9
CHARACTER data type 7–8
CHECK constraint rules 12–16
CHECK constraints 12–16
Child key, definition 8–10
Circular dependencies
elimination 5–15
COLLECT STATISTICS statement
multitable join index 11–44
single-table join index 11–44
Column attributes
COMPRESS 14–21, 14–22
NULL 14–21
Columns
candidates for covered access 10–25
change ratings 7–44
choosing for primary index 9–14
compressing 14–12
compressing hash index 11–75
compressing join index 11–75
compression guidelines 14–22
definition 7–5
derived, types 5–67
fixed length 14–6
importance of naming 7–10
maximum and typical values 7–45
naming conventions 7–12
offsets 14–6
planning column size 14–12
presence bits
purpose 14–37
rules for comparing 7–9
sizing 14–12
uncompressed 14–6
VARCHAR 14–6
Comparison operators, null and 13–18
Composite key
definition 5–4
COMPRESS
guideline for break-even point 14–46
how it works 14–22
performance benefits 14–20
presence bits 14–37
space saving benefits 14–57, 14–59, 14–61, 14–63, 14–65, 14–67, 14–69, 14–71, 14–73
Compression
break-even percentage 14–45
break-even point 14–44
hash index 11–34, 11–74
how to calculate compression ratio 14–52
how to calculate efficiency 14–52
how to calculate percent compression 14–47
join index 11–74
Zipf law 14–24
Compression guidelines 14–22
Compression ratio
definition 14–52
relationship to percent compression 14–52

Connectivity in a relationship 4–9
Constraint codes
  definitions 7–21
Constraint names 12–16
Constraints
  and business rules 7–20
  ATM form 7–20
  CHECK 12–16
  check time 12–22
  defining using CREATE TABLE and ALTER TABLE statements 7–20
  enforcement through views 12–22
  FOREIGN KEY 12–19
  joins 11–69
  maintaining database integrity 7–16
  named 12–16
  PRIMARY KEY 12–20
  REFERENCES 12–17
  referential 7–16
  UNIQUE 12–21
  when enforced 12–22
Constraints form
  information in 7–20
  purpose 7–20
Cost of a join index 11–71, 11–73
  creation 11–72, 11–79
  definition 11–71
  disk resources 11–72
  maintenance 11–73, 11–83, 11–85, 11–88, 11–95
Cost/benefit analysis for a join index 11–99
Covered access
  column candidacy 10–25
CRASHDUMPS database
  DIP utility 15–8
  space requirements 15–12
  usage 15–4
Cursors
  database integrity issues 12–23
  positioned 12–23
  updatable 12–23
Cylinder index 8–47

D

Data
  optimal access 9–14, 10–29
  types of derived columns 5–67
  uniform distribution 9–14
  see also Derived data
Data block 8–48
Data demographics
  calculating for multicolumn tables 7–52
  information required for multicolumn objects 7–50
  information required for single column objects 7–42, 7–50
  trends 7–52
Data Dictionary, estimating space requirements 15–11
Data disks
  allocating space to primary indexes 9–56
  permanent space usage 15–4, 15–6
  reserved space usage 15–4
  spool space usage 15–7
  temporary space usage 15–7
Data independence in database design 2–6
Data manipulation language. See DML
Data mart
  common problems 1–8
  data mart-centrism 1–8, 1–9
  definition 1–4
  dependent 1–4
  independent 1–4
  logical 1–4
  why they fail 1–8
Data modeling
  enterprise 3–2
  guidelines 2–2
Data models
  applications for entity-relationship 4–3
  entity-relationship 4–2
  relational 4–2
Data types
  for describing a domain 7–8
  hashing 8–50
Data warehouse
  ad hoc queries 1–24
  classical definition 1–10
  detail data 1–19
  DSS 1–14
  negative effects of storing only summary data 1–17
  OLTP 1–13
  proactive use of detail data 1–19
  special design considerations 1–11
  Teradata definition 1–10
Database design
  activity transaction modeling 2–12
  analyzing requirements 2–10
  ATM 2–11
  data independence 2–6
  designing for backups 15–17

differences between OLTP and decision support 2–4
heart of the data warehouse 1–2
isolating large tables 15–18
isolating users from database C–2
large database
definition 15–17
life cycle 2–2
logical design phase 2–11
physical design phase 2–13
requirements analysis 3–7, 3–9
small database, definition 15–17
see also Physical design
Database structures
end user profiles, Batch_Profile C–4
end user profiles, Query_Profile C–4
end user profiles, rules for C–4
end user profiles, rules for Batch_Profile only C–5
end user profiles, Upsert_Profile C–4
policies and procedures C–4
Query_Database, rules for C–5
Upsert_Database, rules for C–5
work flows, profiles, objects C–3
DATE data type 7–8
DBC
permanent space usage 15–4
DECIMAL data type 7–8
Decision support
and nonunique indexes 8–6
differs from OTLP 2–4
workarounds 5–56
Degree
of relationship 4–9
Denormalization
alternatives to 5–51
considerations 5–49
negative and positive effects 5–49
optimizing performance with 5–51
reasons for not using derived data 5–68
reasons for not using prejoins 5–65
reasons for using a join index 5–66
reasons for using prejoins 5–65
reasons not to use 5–60
reasons to use 5–60
repeating groups 5–59
types 5–48
using a join index 5–66
using global temporary tables 5–69
using logical model instead of 5–65
using views with aggregation 5–61
Dependency preservation
definition 6–20
reducing update problems 6–20
Dependency theory
terms
determinant 5–8
full functional dependence 5–8
functional dependence 5–8
multivalued dependence 5–8
transitive dependence 5–9
Derived columns
and normalization 5–67
types 5–67
Derived data
denormalizing standalone 5–67
handling guidelines 5–67
reasons for not using to denormalize 5–68
types of in derived columns 5–67
using global temporary tables instead of 5–69
Determinant 5–8
DIFFERENCE operator 5–7
Dimensional modeling
contrasting with entity-relational modeling 5–54
definition 5–52
shortcomings 5–57
snowflake schema 5–54
star schema 5–53
strengths 5–55
summary 5–57
value 5–57
DIP utility 15–8
Disk resources
cost
cost of a join index 11–72
Dispatcher
definition 8–2
DIVIDE operator 5–7
DML
basic statements 8–13
role of WHERE clause 8–13
Domain data types
BYTE 7–8
BYTEINT 7–8
CHARACTER 7–8
DATE 7–8
DECIMAL 7–8
DOUBLE PRECISION 7–8
FLOAT 7–8
GRAPHIC 7–8
INTEGER 7–8
INTERVAL 7–9
LONG VARCHAR 7–8
REAL 7–8
SMALLINT 7–8
TIME 7–9
TIMESTAMP 7–9

VARBYTE 7–8
VARCHAR 7–8
Domain type data codes 7–18
Domain/Key normal form 5–27
Domains
and referential integrity 5–45
ATM form 7–14, 7–17
concept 7–7
defining during logical design phase 7–9
definition 5–4, 7–3
key relationships 7–8
properties 7–7
rules
foreign keys 5–47
primary keys 5–46
storing definitions 7–14
Domains form
information in 7–17
purpose 7–17
DOUBLE PRECISION data type 7–8
Dumps 15–3

E

Enterprise data model 3–2
dimensions 3–3
perspectives 3–3
rules 3–4
Entity
definition 4–4
relationship ratios 4–5
Entity Integrity Rule 5–34, 5–37
Entity-relational modeling
advantages 5–57
contrasting with dimensional modeling 5–54
reasons for advocating 5–55
Existence dependency
in a relationship 4–10
EXPLAIN
using to determine bit mapping 10–23
EXPLAIN statement
"Last Use" 14–96
information provided by 8–8
reasons for using 8–5

F

Fallback
hash index 11–35
hash maps 8–37
join index 8–5, 11–6
reconfiguration 8–37
space considerations 14–93
FastLoad
restrictions with hash index 11–41, 11–42
restrictions with join index 11–41, 11–42
Few-AMP joins and tactical queries 16–13
Field, definition 5–4
Fifth normal form, definition 5–26
First normal form, definition 5–13
Flag byte
definition 14–6
FLOAT data type 7–8
Foreign join index
definition 11–84
Foreign key
applicable rules 12–20
attributes 4–5
definition 5–5, 5–36, 7–6, 8–10
domain rules and referential integrity 5–47
naming columns 7–14
nulls in 5–37
placing guidelines 4–11
uses 5–36
using 5–36
FOREIGN KEY constraints 12–19
Forms
application 7–24
constraints 7–20
domain 7–17
report/query analysis 7–26
row size 14–87
system 7–22
table 7–33
Fourth normal form
definition 5–23
joins 5–23
Functional dependence
definition 5–8
full 5–8
inference axioms 5–9

G

Generator
  definition 8–2
Global temporary tables
  using instead of derived data 5–69
  using instead of prejoins 5–69
  using to denormalize 5–69
  using to enhance performance 5–69
GRAPHIC data type 7–8
Grouping key, definition 8–10

H

Hash collisions 8–34
Hash index
  applications 11–34
  compressing columns 11–75
  compressing rows 11–74
  compression 11–34, 11–74
  defining Fallback 11–35
  definition 11–33
  organizing storage 11–74
  restrictions
    Archive and Restore 11–42
    permanent journal recovery 11–41
    TPump 11–41
    triggers 11–41
  space overhead formula 11–75
  unsupported features 11–42
    Archive and Restore 11–42
    FastLoad 11–42
    permanent journal recovery 11–42
    Triggers
      restrictions with hash index 11–42
  using queries 11–33
Hash key, definition 8–10
Hash maps
  Destination Selection Word 8–37
  eliminating skew 8–35
  Fallback
    current configuration 8–37
    reconfiguration 8–37
  maintaining 8–35
  primary
    current configuration 8–36
    reconfiguration 8–37
  purpose 8–35
Hashed row
  cylinder index 8–47
  data block 8–48
  master index 8–45
Hashing
  advantages of vs. indexing
  advantages over indexing 8–28
  algorithm 8–4, 8–32
  and indexes 8–3
  collisions
    minimizing 8–34
  disadvantages 8–28
  distributing join index rows 11–48
  hash mapping 8–4, 8–50
  HASHAMP function
    description 8–42
  HASHBAKAMP function
    description 8–42
  HASHBUCKET function
    description 8–42
  HASHROW function
    description 8–42
    using 8–42
  primary index 8–27
  row hash value 8–4
  unique secondary indexes 8–27
  uniqueness value 8–4, 8–33
  USI 8–27
  USIs 10–4
  see also Indexing vs. hashing
Hashing algorithm 8–4
  definition 8–32
  Destination Selection Word 8–32, 8–37
  DSW 8–32
  input values 8–32
  uniqueness value 8–33, 14–5

I

IDENTITY columns 14–13
Inclusion Dependency normal form 5–27
Index
  definition 8–11
  unique secondary 8–6
Index Wizard
  column change ratings 7–44
Indexes
  advantages of using 8–4
  and hashing 8–3
  and keys
    differences between 8–11
  comparing types 8–51
  definition 8–3
  disadvantages of using 8–5

hash index 8–6
applications 11–34
definition 11–33
hash mapping 8–4, 8–50
how used in WHERE clause predicates 8–14
join index 8–6
applications 11–4
definition 8–1, 11–2
described 8–7, 11–2
purpose 8–20, 8–23, 11–3
multitable join index 8–7
nonunique primary 8–6
nonunique secondary 8–6
partitioned and nonpartitioned 8–6
primary 8–1, 8–5, 8–16, 9–2
secondary 8–1, 8–6, 10–3
dual and composite 10–16
selection factors 8–54
selectivity 8–3
single-table join index 8–7
sizing hash indexes 14–89
sizing join indexes 14–89
unique primary 8–6
unique secondary 8–6
Indexing
how it works 8–26
Indexing vs. hashing 8–26
how they differ 8–27
indexing
advantages 8–31
tradeoffs between 8–28
Indicator Mode operation 15–14
Inference axioms
dependencies 5–10
list of for functional dependencies 5–9
list of for multivalued dependencies 5–10
INITIATE INDEX ANALYSIS statement 7–44
In-place join index
definition 11–84
Instance, definition 5–5
INTEGER data type 7–8
Intelligent key, definition 5–5
INTERSECTION operator 5–7
INTERVAL data types 7–9
Interviewing for requirements 3–12

J

Join index
ad hoc 11–84
aggregate
defining 11–12
functions 11–26
multitable 11–13
Optimizer criteria for using 11–28
purpose 11–12
related problem solving strategies 11–26
restrictions on aggregate functions 11–20
single-table 11–13
and base tables 11–4
applications 11–4
benefit 11–99
computing 11–99
definition 11–72
benefit equation 11–101
benefit percentage
calculating 11–100
equation 11–101
collecting statistics 11–44, 11–69
collecting statistics for multitable index 11–44
collecting statistics for single-table index 11–45
collecting statistics for single-table index on its base table 11–46
compared with primary and secondary 8–51
compressing columns 11–75
compressing rows 11–74
compression 11–74
cost
cost of creation 11–79
cost of disk resources 11–72
cost of maintenance 11–73
definition 11–71, 11–72
cost/benefit analysis 11–71, 11–73, 11–99
computing benefits 11–99
creating an aggregate join index using CREATE JOIN INDEX statement 11–26
creation cost 11–79
defining Fallback 11–6
definition 11–2
described 8–7
design considerations 11–68
distributing rows 11–48
foreign 11–84
foreign key=primary key 11–69
how stored 8–22, 8–24
improving performance 11–43
in-place 11–84
maintenance
computing approximate cost 11–93
generalizations 11–97
maintenance cost 11–83

as a function of hits/block 11–85
as a function of insert method 11–95
as a function of row size 11–88
multitable 8–7
organizing storage 11–74
payback 11–101
computing 11–72
payback factor 11–101
payback factor equation 11–101
purpose 8–20, 8–23, 11–3
query ratio
computing 11–100
reasons for using to denormalize 5–66
restrictions
Archive and Restore 11–42
number of columns per base table 11–17
number per base table 11–17
number selected per table 11–20
on aggregate functions 11–20
ORDER BY clause 11–22
outer join definitions 11–17
permanent journal recovery 11–41
TPump 11–41
triggers 11–41
WHERE clause
predicates 11–21
rules
OUTER JOIN 11–17
selecting primary index for 11–43
selecting secondary index for 11–43
simple
defining 11–11
multitable 11–12
purpose 11–8
related problem solving strategies 11–24
single-table 11–12
using outer joins 11–68
single-table 8–7, 11–23
functions 11–23
purpose 11–23
sorting 11–69
space overhead formula 11–75
sparse 11–5
summary of functions 11–3
summary of types 11–12
types 8–21
unsupported features
Archive and Restore 11–42
FastLoad 11–42
MultiLoad 11–42
permanent journal recovery 11–42
Triggers 11–42
using COLLECT STATISTICS statement 11–44
using COLLECT STATISTICS statement, single-table version 11–44
using queries 11–4
using to denormalize 5–66
JOIN operator 5–7
Joins
constraints 11–69
definition 5–7
few-AMP 16–13
lossless
definition 6–5
nonkey attributes 6–16
relational 6–5, 6–11
lossy. See Semantic disintegrity
nested 16–14
relational 6–2
Prejoins
see also Join indexes

K

Keys
alternate 5–4, 8–10
and indexes
differences between 8–11
candidate 5–4, 8–10
child 8–10
composite 5–4
definition 5–5, 8–10
foreign 5–5, 8–10
grouping 8–10
hash 8–10
intelligent 5–5
natural 5–5
order 8–10
parent 8–10
primary 5–5, 8–10
search 8–10
sort 8–10
surrogate 5–6, 5–33

L

Legacy database
  analysis 3–16
Lexer
  definition 8–2
Life cycle, database design 2–2
Limits
  database B–3
  session B–5
  system B–2
Logical design
  assigning domains 7–9
  ATM 7–1
  formalizing relationships 2–11
  normalization 2–11
  using instead of denormalization 5–65
LONG VARCHAR data type 7–8

M

Many-to-many relationships
  definition 4–13
  drawing tables
    guidelines 4–13
Master index 8–45
Maxima
  database B–3
  session B–5
  system B–2
Memory swapping 15–3
Mission objectives 3–10
Mission statement 3–9
Models
  many-to-many 4–13
MultiLoad
  restrictions with hash index 11–41, 11–42
  restrictions with join index 11–41, 11–42
Multitable join index 8–7
Multivalued dependence
  definition 5–8
  inference axioms 5–9

N

Named constraints 12–16
Names
  columns 7–12
  foreign key columns 7–14
Names, constraint 12–16
n-ary relationship 4–9
Natural key, definition 5–5
Nested join
  tactical queries 16–14
Net capacity usage
  definition 14–52
  detailed calculation 14–55
  how to calculate row header space 14–53
  how to calculate row vacated space 14–53
  how to calculate table header space 14–53
  how to calculate with fallback 14–53
  how to calculate with no fallback 14–53
Nonkey
  attributes 4–5, 5–13, 6–16
Nonpartitioned primary index
Nonunique primary index 8–5, 8–6, 8–16
Nonunique secondary index 8–6, 8–18
Normal forms
  1NF 5–13
  2NF 5–15
  3NF 5–18
  4NF 5–23
  5NF 5–26
  BCNF 5–19
    definition 5–19
  Boyce-Codd 5–19
  domain/key 5–27
  fifth, definition 5–26
  first, definition 5–13
  fourth, definition 5–23
  higher 5–22
  IDNF 5–27
  inclusion dependency 5–27
  PJ/NF 5–26
  projection-join 5–26
  second, definition 5–15
  sixth 5–27
  third, definition 5–18
Normalization
  advantages 5–57
  and derived columns 5–67
  and primary key values 7–15
  cost 5–3
  definition of terms 5–4, 7–6
    alternate key 5–4
    composite key 5–4
  goals 5–29
  in logical database design phase 2–11

layers 5–11
objectives 5–11
of relations 5–12
procedure 5–30
summary of advantages 5–28
Normalization, definition of terms
attributes 5–4
candidate key 5–4
domain 5–4
field 5–4
foreign key 5–5
instance 5–5
primary key 5–5
relational schema 5–5
relations 5–5
repeating group 5–5
super key 5–6
tuple 5–6
NPPI
nonpartitioned primary index 8–5
NPPI, see nonpartitioned primary index
NULL
presence bits 14–37
Null
aggregates and 13–19
ANSI DateTime and 13–19
arithmetic functions and 13–18
arithmetic operators and 13–18
collation sequence 13–22
comparison operators and 13–18
excluding 13–22
searching for 13–22
searching for, null and non-null 13–23
NUPI 8–5, 8–6, 8–16, 9–4, 9–15, 10–3
assigning 8–6
Duplicate NUPI row read I/O and number of rows inserted 9–50
duplicate row checks 9–54
eliminating duplicate row checks 9–54
guidelines for maximum number of NUPI duplicates 9–52, 9–53
minimizing number of duplicate row checks 9–54
multicolumn 9–54
multicolumn vs. single column 9–55
performance effects of duplicate row checks 9–50
NUSI 8–6, 8–18, 14–92
tactical queries and non-localization of work 16–12

O

Octet vs. byte
presence bits 14–37
OLTP 1–13
and indexes 8–5
differs from decision support 2–4
primary key USI 10–30
support for database processing 5–55
One-to-many relationships
definition 4–12
One-to-one relationships
definition 4–11
Optimizer
definition 8–2
ORDER BY clause
restrictions with join index 11–22
value ordering 11–48
Order independence in relations 5–6
Order key, definition 8–10

P

Parent key
forming referential constraints 5–40
Parent key, definition 8–10
Pareto, Wilfredo
80:20 rule 14–25
law of the trivial many and the critical few 14–25
Parser
definition 8–2
PARTITION columns 14–15
Partition number 14–6
component of RowID for PPI tables 14–5
Partitioned primary index
partition number 14–6
Payback for a join index
computing 11–101
definition 11–72
PE software
dispatcher 8–2
generator 8–2
lexer 8–2
optimizer 8–2
parser 8–2
resolver 8–2
syntaxer 8–2
Percent compression
definition 14–47
relationship to compression ratio 14–52

Performance considerations
primary index 9–46
nonunique 9–47
optimizing nonunique 9–54
unique 9–46
PERM space. See Permanent space
Permanent journal recovery
restrictions with hash index 11–41, 11–42
restrictions with join index 11–41, 11–42
Permanent space
allocating for system users 15–6
estimating 15–15
QCD tables 14–99
Physical design
limiting column values 7–13
realizing database 2–13
PPI
partitioned primary index 8–5
PPI, see partitioned primary index
Presence bits
mappings for COMPRESS and NULL 14–39
purpose 14–37
Presence bytes
how to calculate 14–54
Primary index
analyzing demographics 9–24
compared with secondary and join index 8–51
creating using CREATE TABLE 9–17
dimensions 8–5
distribution demographics
distinct values 9–24
maximum rows null 9–25
maximum rows per value 9–25
hashing 8–27
how stored 8–17
locating rows using 9–40
maximum number of columns 9–3
maximum number per table 8–16
nonpartitioned 8–17, 8–52, 9–3, 9–6
nonunique
advantages and disadvantages 9–55
assigning 8–6
assigning minor entities to 9–4
definition 8–6
optimizing performance 9–54
performance considerations 9–47
optimal data access 9–14
partitioned 8–16, 8–17, 8–52, 9–3, 9–7
performance considerations 9–46
PPI vs. NPPI 8–5
principal selection criteria for column s 9–19
principal selection criteria for columns
access demographics 9–19
distribution demographics 9–22
volatility 9–23
purpose 8–16, 9–2
retrieving row 9–42
secondary selection criteria for columns 9–45
selecting for join index 11–43
specifying default 9–9
summary of properties 9–17
summary of selection criteria 9–39
types 8–16
uniform data distribution 9–14
unique 8–6
assigning major entities to 9–4
correcting uneven distribution 8–44
definition 8–6
performance considerations 9–46
using HASHAMP 8–44
using HASHROW 8–42, 8–44
UPI vs. NUPI 8–5
using space 9–56
see also UPI and NUPI
Primary index, nonunique
see also NUPI
Primary index
unique
see also UPI
Primary key
and referential integrity 5–46
assigning USI 10–30
attributes 4–5
defining bounds 5–34
definition 5–5, 5–32, 7–5, 8–10
domain rules and referential integrity 5–46
identifying 5–32
relationship in first normal form 5–13
selecting guidelines 5–34
values and normalization 7–15
PRIMARY KEY constraints 12–20
PRODUCT operator 5–7
Project mission statement 3–9
PROJECTION operator 5–7
Projection-Join normal form 5–26

Q

QCD
  capacity planning 14–99
  permanent space 14–99
  tables 14–99
QCF
  and Query Capture Database (QCD) 8–9
  capturing query plan information 8–9
Queries
  ad hoc 1–24
  complex 1–20
  simple 1–20
  tactical 1–24
  using hash index 11–33
  using join index 11–4
Query complexity
  relationship with value of response 1–21
Query manager, BTEQ 2–7
Query ratio for a join index
  computing
    Join index
      query ratio definition 11–100
  definition 11–100

R

Ratios
  entity relationships 4–5
REAL data type 7–8
Reference index 5–44, 14–94
  definition 8–25, 14–94
  sizing considerations 14–94
  sizing equation 14–94
REFERENCES constraints 12–17
REFERENCES, CREATE TABLE option
  applicable rules 12–19
Referential constraints
  checks 5–43
  criteria 5–42
  forming using foreign key and parent key
    Foreign key
      forming referential constraints 5–40
Referential integrity
  batch 5–41
  constraints 7–16
  data integrity and 5–40
  definition 5–37
  domains 5–45
  enforcing 5–38
  importance 5–39
  maintaining using foreign key 5–37
  referencing (child) table 5–39
  referencing (parent) table 5–39
Referential Integrity Rule 5–37, 5–38, 5–40
Relational schema, definition 5–5
Relations
  conserving 6–2
  decomposing 5–29
  definition 5–5
  logical operations on 5–7
  logical operators
    DIFFERENCE 5–7
    DIVIDE 5–7
    INTERSECTION 5–7
    JOIN 5–7
    PRODUCT 5–7
    PROJECTION 5–7
    RESTRICTION/SELECTION 5–7
    UNION 5–7
  lossless join 6–11
  normalized 5–11, 5–12
  properties 5–6
  uniqueness property 5–6
Relationships
  decomposing 6–20
  definition 4–5
  degree
    definition 4–9
  dependency preservation 6–20
  entities 4–9
  identifying using primary keys 7–15
  many-to-many 4–13
  modeling 4–11, 4–12, 4–13
  one-to-many 4–12
  one-to-one 4–11
  placing foreign key 4–11
  ratios 4–5
  terms
    binary 4–9
    cardinality 4–9
    connectivity 4–9
    existence dependency 4–10
    n-ary 4–9
    tertiary 4–9
    unary 4–9
  terms associated with 4–9
Repeating group, definition 5–5
Report/Query Analysis
  access column 7–26
  ATM form 7–26
  join column 7–26
  terminology 7–26

Report/Query Analysis form
ATM form 7–26
completing 7–27
purpose 7–26
Requirements analysis 3–7
compiling research information 2–10
gathering information 2–10
gathering input screens 2–10
interviewing 2–10, 3–12
mission objectives 3–10
preparation for interviewing 3–9
project mission statement 3–9
writing requirements specification 2–10
Requirements specification, writing 2–10
Resolver
definition 8–2
RESTRICTION/SELECTION operator 5–7
Row distribution
uneven 9–22, 9–24, 9–25, 9–39
Row fields, definitions
column offsets 14–6
fixed length columns 14–6
flag byte 14–6
PPI RowID 14–5
presence bits 14–6
row length 14–5
row reference array 14–5
uncompressed columns 14–6
VARCHAR columns 14–6
Row hash
component of RowID 14–5
RowID 8–4
value storage 8–4
Row header space
calculating for net capacity usage 14–53
Row size form 14–87
Row vacated space
calculating for net capacity usage 14–53
ROWID column 14–14
Rows
compressing hash index 11–74
compressing join index 11–74
defining layout for NUSIs 10–10
defining layout for USIs 10–5
definition 7–4
length of NPPI rows vs. PPI rows 14–5
locating using primary index 9–40
maximum null 9–25
retrieving using primary index 9–42
size form 14–87
sizing 14–84
Rules
CHECK constraint 12–16
constraint names 12–16
foreign key 12–20
foreign key domains and referential integrity 5–47
primary key domains and referential integrity 5–46
REFERENCES, CREATE TABLE option 12–19

S

Search key, definition 8–10
Second normal form, definition 5–15
Secondary index 8–6
accessing 10–32
compared with primary and join index 8–51
differences in processing USIs and NUSIs 10–36
dual and composite 10–16
hashing unique 8–27
how stored 8–19
maximum number per table 8–18
nonunique
accessing 10–11
aggregate operations 10–25
ANDed predicate set 10–17
assigning to nonunique columns 10–3
bit mapping, advantages 10–20
bit mapping, restrictions on 10–20
complex conditional expressions 10–21
computing bit maps 10–22
defining row layout fields 10–10
determining using EXPLAIN 10–23
hash-ordered 10–27
ORed predicate set 10–18
row layout 10–9
sizing considerations 14–92
sizing considerations, special 14–92
sizing equation 14–92
space requirements 14–93
usage summary 10–39
using 8–6
value-ordered
limitations on 10–27
optimal data access 9–14, 10–29
purpose 8–18, 10–3
selecting for join index 11–43
selection criteria 10–30
selectivity 10–29
space considerations 10–3

unique
accessing 10–6
assigning to primary keys 10–3, 10–30
defining row layout fields 10–5
hashing 10–4
purpose 8–6
sizing considerations 14–91
sizing equation 14–91
space requirements 14–91
unique row layout 10–4
usage summary 10–39
usage summary 10–38
Secondary index, unique
defining 8–18
Secondary index
see also USI and NUSI 8–18
Secondary index, nonunique
see also NUSI
Secondary index, unique
see also USI
Selectivity
index
considerations 10–13
dual secondary 10–16
high 8–3, 10–13, 10–29
low 8–3, 10–13
weak 10–29
Semantic disintegrity
attribute mapping 6–17
definition 6–17
Single-table join index 8–7
Sizing equations
NUSI 14–92
reference index 14–94
tables 14–89
USI 14–91
Skew 9–22, 9–24, 9–25, 9–39
SMALLINT data type 7–8
Snowflake schema, dimensional modeling 5–54
Sort key, definition 8–10
Space requirements
base tables 14–89
CRASHDUMPS database 15–12
estimating for Data Dictionary 15–11
estimating spool space 15–8
estimating temporary 15–9
estimating transient journal 15–9
Fallback 14–93
non-fallback 14–93
secondary index
non-unique 14–93
temporary 15–12
user spool space 15–12
Sparse join index 11–5
Specifications
database B–3
session B–5
system B–2
Spool space
allocating 14–97
estimating requirements 15–8
formula rules 15–12
guidelines for allocating 14–97
intermediate 14–96
output 14–96
requirements 15–12
source 14–97
tables area 15–12
SQL statements
INITIATE INDEX ANALYSIS 7–44
Standalone derived data
denormalizing 5–67
handling guidelines 5–67
Star schema, dimensional modeling 5–53
Summary data
low information content 1–18
Super key
definition 5–6
Surrogate key 5–33
definition 5–6
Syntax, how to read A–1
Syntaxer
definition 8–2
SYSADMIN
allocating permanent storage space 15–7
system user 15–6
System disks
contents 15–3
Disk 0, boot disk 15–3
Disk 1, dumps/memory swapping 15–3
types 15–3
System form
ATM form 7–22
information in 7–22
purpose 7–22
System users
SYSADMIN 15–7
SYSTEMFE 15–7
User DBC 15–6
System-derived columns 14–13
SYSTEMFE
allocating permanent storage space 15–7
system user 15–6

System-generated columns 14–13
IDENTITY columns 14–13
PARTITION 14–15
ROWID 14–14

T

Table form
completing 7–33
completing access information 7–41
completing basic information 7–38
completing column-level information 7–39
completing data demographics, multicolumn objects 7–50
completing data demographics, single column objects 7–42
completing miscellaneous column-level information 7–40
information in 7–34
information type
accessing 7–34
data demographics 7–34
miscellaneous 7–34
maximum and typical column values 7–45
purpose 7–33
Table header
components 14–7
description 14–7
field 1 14–8
field 2 14–9
field 3 14–9
field 4 14–9
field 5 14–10
field 6 14–11
field 7 14–11
relation to compressed column values 14–7, 14–21
Table header space
calculating for net capacity usage 14–53
Tables
access methods
full table scan 8–53
join index 8–53
nonunique primary index 8–52
nonunique secondary index 8–53
unique primary index 8–52
unique secondary index 8–52
activities modeling 7–4
associative
definition 4–8
naming 4–8
ATM form 7–33
base 11–3, 11–4
compressibility 14–37
determining user overhead 15–11
Fallback
space considerations 14–93
join index
number per 11–17
large
definition 15–17
non-fallback
space considerations 14–93
non-prime
definition 4–7
nullability 14–37
prime
definition 4–7
sizing
determining empirically 14–101
evaluating legacy 14–102
summary of principal points 14–103
variables 14–88
sizing base 14–89
sizing equations 14–89
small, definition 15–17
summary of access methods 8–52
Tables area
CRASHDUMPS database 15–12
user spool space 15–12
user temporary space 15–12
Tables, QCD 14–99
Tactical queries 1–24
effect of data volume growth on response times 16–6
effect of growth in number of concurrent users on response time 16–6
effects of configuration expansion on response times 16–6
nested joins 16–14
NUSI access and non-localization of work 16–12
scalability 16–5
USI access and work localization 16–11
Target Level Emulation Facility
capturing cost data 8–9
TEMP space. See Temporary space
Temporary space
allocating 15–8
estimating requirements 15–9
requirements 15–12
Teradata database
size considerations 15–2
Teradata Index Wizard 8–8
column change ratings 7–44

Teradata RDBMS
database specifications B–3
session specifications B–5
system specifications B–2
Teradata Statistics Wizard 8–7, 11–5
Tertiary relationship 4–9
Third normal form, definition 5–18
TIME data type 7–9
TIMESTAMP data type 7–9
TLE 8–9
TPump
restrictions with hash index 11–41
restrictions with join index 11–41
Transient journal
calculating block size 15–10
estimating requirements 15–9
Transitive dependence 5–9
Triggers
and business rules 7–20
restrictions with hash index 11–41
restrictions with join index 11–41, 11–42
Tuple, definition 5–6

U

Unary relationship 4–9
UNION operator 5–7
UNIQUE constraints 12–21
Unique primary index 8–5, 8–6, 8–16
Unique secondary index 5–42, 8–6, 8–18, 10–3, 14–91, 16–11
defining 8–18
hashing 8–27
Uniqueness
description 5–6
determining primary index disk usage 9–56
property of relations 5–6
Uniqueness value
assigning values to 8–4
forming RowID 8–4, 14–5
hashing algorithm 8–33, 14–5
UPI 5–42, 8–5, 8–6, 8–16, 9–4, 9–15
User DBC 15–6
allocating permanent storage space 15–6, 15–15
permanent space usage 15–6
spool space usage 15–7
temporary space usage 15–7
User spool space, requirements 15–12
User temporary space, requirements 15–12
Users. See System users
USI 5–42, 8–6, 8–18, 10–3, 14–91
defining 8–18
hashing 8–27
tactical queries and work localization 16–11
Utilities
DIP 15–8

V

VARBYTE data type 7–8
VARCHAR data type 7–8
Views
reasons for not using prejoins to denormalize 5–65
reasons for using prejoins to denormalize 5–65
using global temporary tables instead 5–69
using prejoins to denormalize 5–64
using to denormalize 5–61
using to isolate users C–2

W

WHERE clause
how used by Optimizer 8–14
in DML 8–13
predicates
invoking full table scans 8–15
invoking indexes 8–14
restriction on join index 11–21
Workarounds, decision support 5–56

Z

Zipf law
compression 14–24
power-law function 14–24
Zipf, George Kingsley 14–24