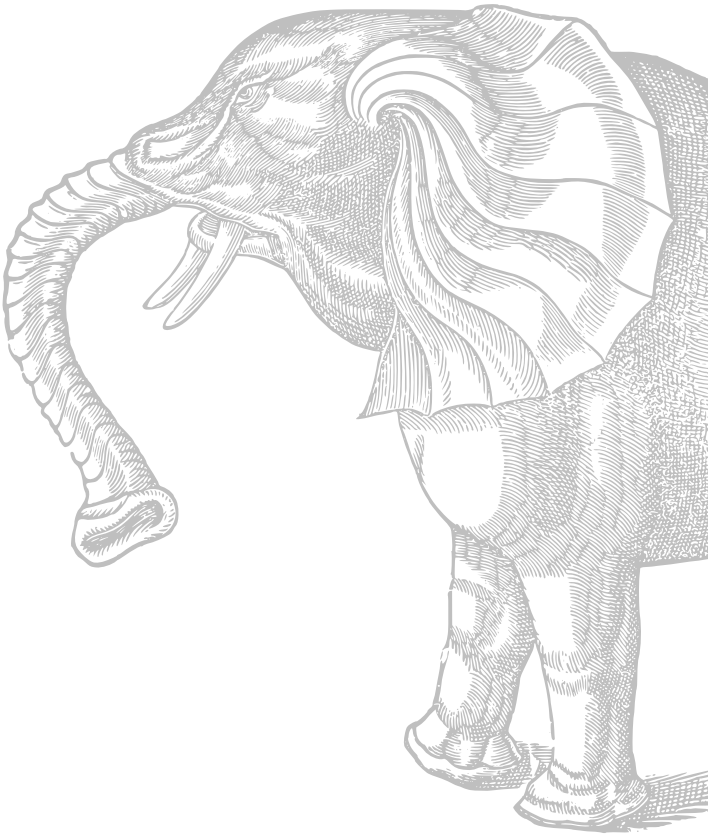
Egor Rogov

PostgreSQL 14 Internals



These are Parts I–IV of the book.
The last part will follow soon:

V. Index Types

Postgres Professional
Moscow, 2022

The elephant on the cover is a fragment of an illustration from Edward Topsell's *The History of Four-footed Beasts and Serpents*, published in London in 1658

PostgreSQL 14 Internals
by Egor Rogov

Translated from Russian by Liudmila Mantrova

© Postgres Professional, 2022

This book in PDF is available at postgrespro.com/community/books/internals

Contents at a Glance

About This Book 15
1 Introduction 21

Part I Isolation and MVCC **41**
2 Isolation 43
3 Pages and Tuples 70
4 Snapshots 92
5 Page Pruning and HOT Updates 106
6 Vacuum and Autovacuum 118
7 Freezing 143
8 Rebuilding Tables and Indexes 156

Part II Buffer Cache and WAL **167**
9 Buffer Cache 169
10 Write-Ahead Log 189
11 WAL Modes 209

Part III Locks **225**
12 Relation-Level Locks 227
13 Row-Level Locks 239
14 Miscellaneous Locks 263
15 Locks on Memory Structures 274

Part IV Query Execution **283**
16 Query Execution Stages 285
17 Statistics 308
18 Table Access Methods 333
19 Index Access Methods 354
20 Index Scans 373
21 Nested Loop 397

22 Hashing 417
23 Sorting and Merging 440

Index 464

Table of Contents

About This Book **15**

1 Introduction **21**
1.1 Data Organization 21
Databases 21
System Catalog 22
Schemas 23
Tablespaces 24
Relations 25
Files and Forks 26
Pages 30
TOAST 30
1.2 Processes and Memory 35
1.3 Clients and the Client-Server Protocol 37

Part I Isolation and MVCC **41**

2 Isolation **43**
2.1 Consistency 43
2.2 Isolation Levels and Anomalies Defined by the SQL Standard 45
Lost Update 46
Dirty Reads and Read Uncommitted 46
Non-Repeatable Reads and Read Committed 47
Phantom Reads and Repeatable Read 47
No Anomalies and Serializable 48
Why These Anomalies? 48
2.3 Isolation Levels in PostgreSQL 49
Read Committed 50
Repeatable Read 59
Serializable 65
2.4 Which Isolation Level to Use? 68

3 Pages and Tuples **70**
3.1 Page Structure . . . 70
Page Header . . . 70
Special Space . . . 71
Tuples . . . 71
Item Pointers . . . 72
Free Space . . . 73
3.2 Row Version Layout . . . 73
3.3 Operations on Tuples . . . 75
Insert . . . 76
Commit . . . 79
Delete . . . 81
Abort . . . 82
Update . . . 83
3.4 Indexes . . . 84
3.5 TOAST . . . 85
3.6 Virtual Transactions . . . 85
3.7 Subtransactions . . . 86
Savepoints . . . 86
Errors and Atomicity . . . 89

4 Snapshots **92**
4.1 What is a Snapshot? . . . 92
4.2 Row Version Visibility . . . 93
4.3 Snapshot Structure . . . 94
4.4 Visibility of Transactions' Own Changes . . . 98
4.5 Transaction Horizon . . . 100
4.6 System Catalog Snapshots . . . 103
4.7 Exporting Snapshots . . . 104

5 Page Pruning and HOT Updates **106**
5.1 Page Pruning . . . 106
5.2 HOT Updates . . . 110
5.3 Page Pruning for HOT Updates . . . 113
5.4 HOT Chain Splits . . . 115
5.5 Page Pruning for Indexes . . . 116

6 Vacuum and Autovacuum **118**
6.1 Vacuum . . . 118
6.2 Database Horizon Revisited . . . 121
6.3 Vacuum Phases . . . 124
Heap Scan . . . 124
Index Vacuuming . . . 124
Heap Vacuuming . . . 125
Heap Truncation . . . 126
6.4 Analysis . . . 126
6.5 Automatic Vacuum and Analysis . . . 127
About the Autovacuum Mechanism . . . 127
Which Tables Need to be Vacuumed? . . . 129
Which Tables Need to Be Analyzed? . . . 131
Autovacuum in Action . . . 132
6.6 Managing the Load . . . 136
Vacuum Throttling . . . 136
Autovacuum Throttling . . . 137
6.7 Monitoring . . . 138
Monitoring Vacuum . . . 138
Monitoring Autovacuum . . . 141

7 Freezing **143**
7.1 Transaction ID Wraparound . . . 143
7.2 Tuple Freezing and Visibility Rules . . . 144
7.3 Managing Freezing . . . 147
Minimal Freezing Age . . . 148
Age for Aggressive Freezing . . . 149
Age for Forced Autovacuum . . . 151
Age for Failsafe Freezing . . . 153
7.4 Manual Freezing . . . 153
Freezing by Vacuum . . . 154
Freezing Data at the Initial Loading . . . 154

8 Rebuilding Tables and Indexes **156**
8.1 Full Vacuuming . . . 156
Why is Routine Vacuuming not Enough? . . . 156
Estimating Data Density . . . 157

Freezing 160
8.2 Other Rebuilding Methods 162
Alternatives to Full Vacuuming 162
Reducing Downtime during Rebuilding 162
8.3 Preventive Measures 163
Read-Only Queries 163
Data Updates 164

Part II Buffer Cache and WAL 167

9 Buffer Cache 169
9.1 Caching 169
9.2 Buffer Cache Design 170
9.3 Cache Hits 172
9.4 Cache Misses 176
Buffer Search and Eviction 177
9.5 Bulk Eviction 179
9.6 Choosing the Buffer Cache Size 182
9.7 Cache Warming 185
9.8 Local Cache 187

10 Write-Ahead Log 189
10.1 Logging 189
10.2 WAL Structure 190
Logical Structure 190
Physical Structure 194
10.3 Checkpoint 195
10.4 Recovery 199
10.5 Background Writing 202
10.6 WAL Setup 203
Configuring Checkpoints 203
Configuring Background Writing 206
Monitoring 206

11 WAL Modes 209
11.1 Performance 209

11.2 Fault Tolerance 213
Caching 213
Data Corruption 215
Non-Atomic Writes 217
11.3 WAL Levels 219
Minimal 220
Replica 222
Logical 224

Part III Locks **225**

12 Relation-Level Locks **227**
12.1 About Locks 227
12.2 Heavyweight Locks 229
12.3 Locks on Transaction IDs 231
12.4 Relation-Level Locks 232
12.5 Wait Queue 235

13 Row-Level Locks **239**
13.1 Lock Design 239
13.2 Row-Level Locking Modes 240
Exclusive Modes 240
Shared Modes 242
13.3 Multitransactions 243
13.4 Wait Queue 245
Exclusive Modes 245
Shared Modes 251
13.5 No-Wait Locks 254
13.6 Deadlocks 256
Deadlocks by Row Updates 258
Deadlocks Between Two UPDATE Statements 259

14 Miscellaneous Locks **263**
14.1 Non-Object Locks 263
14.2 Relation Extension Locks 265
14.3 Page Locks 265

14.4 Advisory Locks 266
14.5 Predicate Locks 268

15 Locks on Memory Structures 274
15.1 Spinlocks 274
15.2 Lightweight Locks 275
15.3 Examples 275
Buffer Cache 275
WAL Buffers 277
15.4 Monitoring Waits 278
15.5 Sampling 280

Part IV Query Execution 283

16 Query Execution Stages 285
16.1 Demo Database 285
16.2 Simple Query Protocol 288
Parsing 288
Transformation 289
Planning 292
Execution 300
16.3 Extended Query Protocol 302
Preparation 302
Parameter Binding 303
Planning and Execution 304
Getting the Results 306

17 Statistics 308
17.1 Basic Statistics 308
17.2 NULL Values 312
17.3 Distinct Values 313
17.4 Most Common Values 315
17.5 Histogram 318
17.6 Statistics for Non-Scalar Data Types 322
17.7 Average Field Width 323
17.8 Correlation 323

17.9 Expression Statistics . . . 324
Extended Expression Statistics . . . 325
Statistics for Expression Indexes . . . 326
17.10 Multivariate Statistics . . . 327
Functional Dependencies Between Columns . . . 327
Multivariate Number of Distinct Values . . . 329
Multivariate MCV Lists . . . 331

18 Table Access Methods 333
18.1 Pluggable Storage Engines . . . 333
18.2 Sequential Scans . . . 335
Cost Estimation . . . 336
18.3 Parallel Plans . . . 340
18.4 Parallel Sequential Scans . . . 341
Cost Estimation . . . 341
18.5 Parallel Execution Limitations . . . 345
Number of Background Workers . . . 345
Non-Parallelizable Queries . . . 348
Parallel Restricted Queries . . . 350

19 Index Access Methods 354
19.1 Indexes and Extensibility . . . 354
19.2 Operator Classes and Families . . . 357
Operator Classes . . . 357
Operator Family . . . 362
19.3 Indexing Engine Interface . . . 364
Access Method Properties . . . 365
Index Properties . . . 369
Column Properties . . . 370

20 Index Scans 373
20.1 Regular Index Scans . . . 373
Cost Estimation . . . 374
Good Scenario: High Correlation . . . 375
Bad Scenario: Low Correlation . . . 378
20.2 Index-Only Scans . . . 381
Indexes with the Include Clause . . . 384

20.3 Bitmap Scans . . . 385
Bitmap Accuracy . . . 387
Operations on Bitmaps . . . 388
Cost Estimation . . . 389
20.4 Parallel Index Scans . . . 393
20.5 Comparison of Various Access Methods . . . 395

21 Nested Loop 397
21.1 Join Types and Methods . . . 397
21.2 Nested Loop Joins . . . 398
Cartesian Product . . . 399
Parameterized Joins . . . 403
Caching Rows (Memoization) . . . 407
Outer Joins . . . 411
Anti- and Semi-joins . . . 412
Non-Equi-joins . . . 415
Parallel Mode . . . 415

22 Hashing 417
22.1 Hash Joins . . . 417
One-Pass Hash Joins . . . 417
Two-Pass Hash Joins . . . 422
Dynamic Adjustments . . . 425
Hash Joins in Parallel Plans . . . 429
Parallel One-Pass Hash Joins . . . 430
Parallel Two-Pass Hash Joins . . . 432
Modifications . . . 435
22.2 Distinct Values and Grouping . . . 437

23 Sorting and Merging 440
23.1 Merge Joins . . . 440
Merging Sorted Sets . . . 440
Parallel Mode . . . 443
Modifications . . . 444
23.2 Sorting . . . 445
Quicksort . . . 447
Top-N Heapsort . . . 448

External Sorting 450
Incremental Sorting 454
Parallel Mode 456
23.3 Distinct Values and Grouping 458
23.4 Comparison of Join Methods 460

Index **464**

About This Book

> Books are not made to be believed, but to be subjected to inquiry.
>
> — Umberto Eco, *The Name of the Rose*

For Whom Is This Book?

This book is for those who will not settle for a black-box approach when working with a database. If you are eager to learn, prefer not to take expert advice for granted, and would like to figure out everything yourself, follow along.

I assume that the reader has already tried using PostgreSQL and has at least some general understanding of how it works. Entry-level users may find the text a bit difficult. For example, I will not tell anything about how to install the server, enter psql commands, or set configuration parameters.

I hope that the book will also be useful for those who are familiar with another database system, but switch over to PostgreSQL and would like to understand how they differ. A book like this would have saved me a lot of time several years ago. And that's exactly why I finally wrote it.

What This Book Will Not Provide

This book is not a collection of recipes. You cannot find ready-made solutions for every occasion, but if you understand inner mechanisms of a complex system, you will be able to analyze and critically evaluate other people's experience and come to your own conclusions. For this reason, I explain such details that may at first seem to be of no practical use.

But this book is not a tutorial either. While delving deeply into some fields (in which I am more interested myself), it may say nothing at all about the other.

By no means is this book a reference. I tried to be precise, but I did not aim at replacing documentation, so I could easily leave out some details that I considered insignificant. In any unclear situation read the documentation.

This book will not teach you how to develop the PostgreSQL core. I do not expect any knowledge of the C language, as this book is mainly intended for database administrators and application developers. But I do provide multiple references to the source code, which can give you as many details as you like, and even more.

What This Book Does Provide

In the introductory chapter, I briefly touch upon the main database concepts that will serve as the foundation for all the further narration. I do not expect you to get much new information from this chapter but still include it to complete the big picture. Besides, this overview can be found useful by those who are migrating from other database systems.

Part I is devoted to questions of data consistency and isolation. I first cover them from the user's perspective (you will learn which isolation levels are available and what are the implications) and then dwell on their internals. For this purpose, I have to explain implementation details of multiversion concurrency control and snapshot isolation, paying special attention to cleanup of outdated row versions.

Part II describes buffer cache and WAL, which is used to restore data consistency after a failure.

Part III goes into details about the structure and usage of various types of locks: lightweight locks for RAM, heavyweight locks for relations, and row-level locks.

Part IV explains how the server plans and executes SQL queries. I will tell you which data access methods are available, which join methods can be used, and how the collected statistics are applied.

Part V extends the discussion of indexes from the already covered B-trees to other access methods. I will explain some general principles of extensibility that define the boundaries between the core of the indexing system, index access methods, and data types (which will bring us to the concept of operator classes), and then elaborate on each of the available methods.

PostgreSQL includes multiple "introspective" extensions, which are not used in routine work, but give us an opportunity to peek into the server's internal behavior. This book uses quite a few of them. Apart from letting us explore the server internals, these extensions can also facilitate troubleshooting in complex usage scenarios.

Conventions

I tried to write this book in a way that would allow reading it page by page, from start to finish. But it is hardly possible to uncover all the truth at once, so I had to get back to one and the same topic several times. Writing that "it will be considered later" over and over again would inevitably make the text much longer, that's why in such cases I simply put the page number in the margin to refer you to further discussion. A similar number pointing backwards will take you to the page where something has been already said on the subject.

p. 17

Both the text and all the code examples in this book apply to PostgreSQL 14. Next to some paragraphs, you can see a version number in the page margin. It means that the provided information is relevant starting from the indicated PostgreSQL version, while all the previous versions either did not have the described feature at all, or used a different implementation. Such notes can be useful for those who have not upgraded their systems to the latest release yet.

v. 14

I also use the margins to show the default values of the discussed parameters. The names of both regular and storage parameters are printed in italics: *work_mem*.

4MB

In footnotes, I provide multiple links to various sources of information. There are several of them, but first and foremost, I list the PostgreSQL documentation,[1] which is a wellspring of knowledge. Being an essential part of the project, it is always kept up-to-date by PostgreSQL developers themselves. However, the primary reference is definitely the source code.[2] It is amazing how many answers you can find by simply reading comments and browsing through README files, even if you do not know C. Sometimes I also refer to commitfest entries:[3] you can always trace the

[1] postgresql.org/docs/14/index.html
[2] git.postgresql.org/gitweb/?p=postgresql.git;a=summary
[3] commitfest.postgresql.org

history of all changes and understand the logic of decisions taken by developers if you read the related discussions in the psql-hackers mailing list, but it requires digging through piles of emails.

> Side notes that can lead the discussion astray (which I could not help but include into the book) are printed like this, so they can be easily skipped.

Naturally, the book contains multiple code examples, mainly in SQL. The code is provided with the prompt =>; the server response follows if necessary:

```
=> SELECT now();
               now
-------------------------------
 2022-11-25 22:57:04.866944+03
(1 row)
```

If you carefully repeat all the provided commands in PostgreSQL 14, you should get exactly the same results (down to transaction IDs and other inessential details). Anyway, all the code examples in this book have been generated by the script containing exactly these commands.

When it is required to illustrate concurrent execution of several transactions, the code run in another session is indented and marked off by a vertical line.

```
=> SHOW server_version;
 server_version
----------------
 14.4
(1 row)
```

To try out such commands (which is useful for self-study, just like any experimentation), it is convenient to open two psql terminals.

The names of commands and various database objects (such as tables and columns, functions, or extensions) are highlighted in the text using a sans-serif font: UPDATE, pg_class.

If a utility is called from the operating system, it is shown with a prompt that ends with $:

```
postgres$ whoami
postgres
```

I use Linux, but without any technicalities; having some basic understanding of this operating system will be enough.

Acknowledgments

It is impossible to write a book alone, and now I have an excellent opportunity to thank good people.

I am very grateful to Pavel Luzanov who found the right moment and offered me to start doing something really worthwhile.

I am obliged to Postgres Professional for the opportunity to work on this book beyond my free time. But there are actual people behind the company, so I would like to express my gratitude to Oleg Bartunov for sharing ideas and infinite energy, and to Ivan Panchenko for thorough support and LaTeX.

I would like to thank my colleagues from the education team for the creative atmosphere and discussions that shaped the scope and format of our training courses, which also got reflected in the book. Special thanks to Pavel Tolmachev for his meticulous review of the drafts.

Many chapters of this book were first published as articles in the Habr blog,[1] and I am grateful to the readers for their comments and feedback. It showed the importance of this work, highlighted some gaps in my knowledge, and helped me improve the text.

I would also like to thank Liudmila Mantrova who has put much effort into polishing this book's language. If you do not stumble over every other sentence, the credit goes to her. Besides, Liudmila took the trouble to translate this book into English, for which I am very grateful too.

[1] habr.com/en/company/postgrespro/blog

I do not provide any names, but each function or feature mentioned in this book has required years of work done by particular people. I admire PostgreSQL developers, and I am very glad to have the honor of calling many of them my colleagues.

1

Introduction

1.1 Data Organization

Databases

PostgreSQL is a program that belongs to the class of database management systems. When this program is running, we call it a PostgreSQL *server*, or *instance*.

Data managed by PostgreSQL is stored in databases.[1] A single PostgreSQL instance can serve several databases at a time; together they are called a *database cluster*.

To be able to use the cluster, you must first *initialize*[2] (create) it. The directory that contains all the files related to the cluster is usually called PGDATA, after the name of the environment variable pointing to this directory.

> Installations from pre-built packages can add their own "abstraction layers" over the regular PostgreSQL mechanism by explicitly setting all the parameters required by utilities. In this case, the database server runs as an operating system service, and you may never come across the PGDATA variable directly. But the term itself is well-established, so I am going to use it.

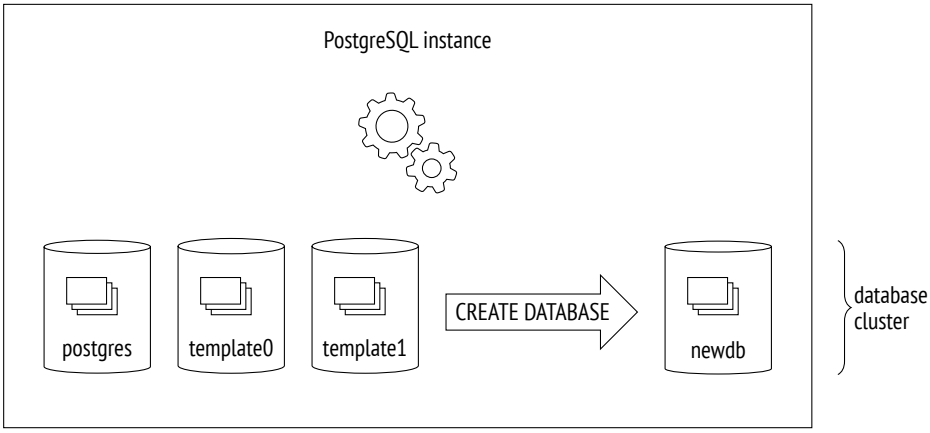
After cluster initialization, PGDATA contains three identical databases:

template0 is used for cases like restoring data from a logical backup or creating a database with a different encoding; it must never be modified.

template1 serves as a template for all the other databases that a user can create in the cluster.

[1] postgresql.org/docs/14/managing-databases.html
[2] postgresql.org/docs/14/app-initdb.html

postgres is a regular database that you can use at your discretion.



System Catalog

Metadata of all cluster objects (such as tables, indexes, data types, or functions) is stored in tables that belong to the *system catalog*.[1] Each database has its own set of tables (and views) that describe the objects of this database. Several system catalog tables are common to the whole cluster; they do not belong to any particular database (technically, a dummy database with a zero ID is used), but can be accessed from all of them.

The system catalog can be viewed using regular SQL queries, while all modifications in it are performed by DDL commands. The psql client also offers a whole range of commands that display the contents of the system catalog.

Names of all system catalog tables begin with pg_, like in pg_database. Column names start with a three-letter prefix that usually corresponds to the table name, like in datname.

In all system catalog tables, the column declared as the primary key is called oid (object identifier); its type, which is also called oid, is a 32-bit integer.

[1] postgresql.org/docs/14/catalogs.html

The implementation of oid object identifiers is virtually the same as that of sequences, but it appeared in PostgreSQL much earlier. What makes it special is that the generated unique IDs issued by a common counter are used in different tables of the system catalog. When an assigned ID exceeds the maximum value, the counter is reset. To ensure that all values in a particular table are unique, the next issued oid is checked by the unique index; if it is already used in this table, the counter is incremented, and the check is repeated.[1]

Schemas

Schemas[2] are namespaces that store all objects of a database. Apart from user schemas, PostgreSQL offers several predefined ones:

public is the default schema for user objects unless other settings are specified.

pg_catalog is used for system catalog tables.

information_schema provides an alternative view for the system catalog as defined by the SQL standard.

pg_toast is used for objects related to TOAST. *p. 30*

pg_temp comprises temporary tables. Although different users create temporary tables in different schemas called pg_temp_*N*, everyone refers to their objects using the pg_temp alias.

Each schema is confined to a particular database, and all database objects belong to this or that schema.

If the schema is not specified explicitly when an object is accessed, PostgreSQL selects the first suitable schema from the *search path*. The search path is based on the value of the *search_path* parameter, which is implicitly extended with pg_catalog and (if necessary) pg_temp schemas. It means that different schemas can contain objects with the same names.

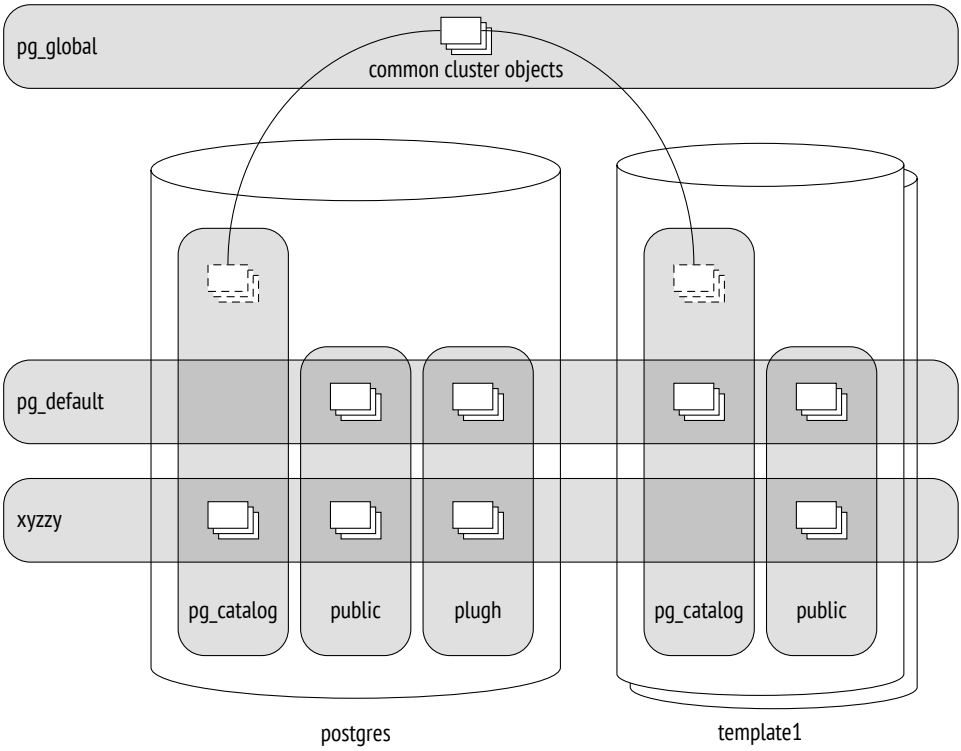
[1] backend/catalog/catalog.c, GetNewOidWithIndex function
[2] postgresql.org/docs/14/ddl-schemas.html

Tablespaces

Unlike databases and schemas, which determine logical distribution of objects, *tablespaces* define physical data layout. A tablespace is virtually a directory in a file system. You can distribute your data between tablespaces in such a way that archive data is stored on slow disks, while the data that is being actively updated goes to fast disks.

One and the same tablespace can be used by different databases, and each database can store data in several tablespaces. It means that logical structure and physical data layout do not depend on each other.

Each database has the so-called *default tablespace*. All database objects are created in this tablespace unless another location is specified. System catalog objects related to this database are also stored there.



During cluster initialization, two tablespaces are created:

pg_default is located in the PGDATA/base directory; it is used as the default tablespace unless another tablespace is explicitly selected for this purpose.

pg_global is located in the PGDATA/global directory; it stores system catalog objects that are common to the whole cluster.

When creating a custom tablespace, you can specify any directory; PostgreSQL will create a symbolic link to this location in the PGDATA/pg_tblspc directory. In fact, all paths used by PostgreSQL are relative to the PGDATA directory, which allows you to move it to a different location (provided that you have stopped the server, of course).

The illustration on the previous page puts together databases, schemas, and tablespaces. Here the postgres database uses tablespace xyzzy as the default one, whereas the template1 database uses pg_default. Various database objects are shown at the intersections of tablespaces and schemas.

Relations

For all of their differences, *tables* and *indexes*—the most important database objects—have one thing in common: they consist of rows. This point is quite self-evident when we think of tables, but it is equally true for B-tree nodes, which contain indexed values and references to other nodes or table rows.

Some other objects also have the same structure; for example, *sequences* (virtually one-row tables) and *materialized views* (which can be thought of as tables that "keep" the corresponding queries). Besides, there are regular *views*, which do not store any data but otherwise are very similar to tables.

In PostgreSQL, all these objects are referred to by the generic term *relation*.

> In my opinion, it is not a happy term because it confuses database tables with "genuine" relations defined in the relational theory. Here we can feel the academic legacy of the project and the inclination of its founder, Michael Stonebraker, to see everything as a relation. In one of his works, he even introduced the concept of an "ordered relation" to denote a table in which the order of rows is defined by an index.

The system catalog table for relations was originally called pg_relation, but following the object orientation trend, it was soon renamed to pg_class, which we are now used to. Its columns still have the REL prefix though.

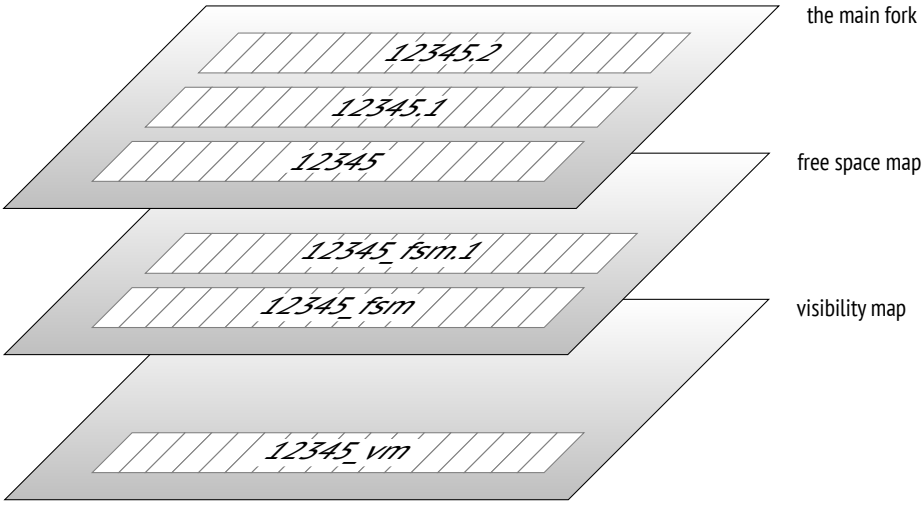
Files and Forks

All information associated with a relation is stored in several different *forks*,[1] each containing data of a particular type.

At first, a fork is represented by a single *file*. Its filename consists of a numeric ID (oid), which can be extended by a suffix that corresponds to the fork's type.

The file grows over time, and when its size reaches 1 GB, another file of this fork is created (such files are sometimes called *segments*). The sequence number of the segment is added to the end of its filename.

The file size limit of 1 GB was historically established to support various file systems that could not handle large files. You can change this limit when building PostgreSQL (./configure --with-segsize).



[1] postgresql.org/docs/14/storage-file-layout.html

Thus, a single relation is represented on disk by several files. Even a small table without indexes will have at least three files, by the number of mandatory forks.

Each tablespace directory (except for pg_global) contains separate subdirectories for particular databases. All files of the objects belonging to the same tablespace and database are located in the same subdirectory. You must take it into account because too many files in a single directory may not be handled well by file systems.

There are several standard types of forks.

The main fork represents actual data: table rows or index rows. This fork is available for any relations (except for views, which contain no data).

Files of the main fork are named by their numeric IDs, which are stored as relfilenode values in the pg_class table.

Let's take a look at the path to a file that belongs to a table created in the pg_default tablespace:

```
=> CREATE UNLOGGED TABLE t(
  a integer,
  b numeric,
  c text,
  d json
);
=> INSERT INTO t VALUES (1, 2.0, 'foo', '{}');
=> SELECT pg_relation_filepath('t');
 pg_relation_filepath
----------------------
 base/16384/16385
(1 row)
```

The base directory corresponds to the pg_default tablespace, the next subdirectory is used for the database, and it is here that we find the file we are looking for:

```
=> SELECT oid FROM pg_database WHERE datname = 'internals';
  oid
-------
 16384
(1 row)
```

```
=> SELECT relfilenode FROM pg_class WHERE relname = 't';
 relfilenode
-------------
       16385
(1 row)
```

Here is the corresponding file in the file system:

```
=> SELECT size
FROM pg_stat_file('/usr/local/pgsql/data/base/16384/16385');
 size
------
 8192
(1 row)
```

The initialization fork[1] is available only for unlogged tables (created with the UNLOGGED clause) and their indexes. Such objects are the same as regular ones, except that any actions performed on them are not written into the write-ahead log. It makes these operations considerably faster, but you will not be able to restore consistent data in case of a failure. Therefore, PostgreSQL simply deletes all forks of such objects during recovery and overwrites the main fork with the initialization fork, thus creating a dummy file.

p. 189

The t table is created as unlogged, so the initialization fork is present. It has the same name as the main fork, but with the _init suffix:

```
=> SELECT size
FROM pg_stat_file('/usr/local/pgsql/data/base/16384/16385_init');
 size
------
    0
(1 row)
```

The free space map[2] keeps track of available space within pages. Its volume changes all the time, growing after vacuuming and getting smaller when new row versions appear. The free space map is used to quickly find a page that can accommodate new data being inserted.

[1] postgresql.org/docs/14/storage-init.html
[2] postgresql.org/docs/14/storage-fsm.html
backend/storage/freespace/README

All files related to the free space map have the _fsm suffix. Initially, no such files are created; they appear only when necessary. The easiest way to get them is to vacuum a table: p. 125

```
=> VACUUM t;
=> SELECT size
FROM pg_stat_file('/usr/local/pgsql/data/base/16384/16385_fsm');
 size
-------
 24576
(1 row)
```

To speed up search, the free space map is organized as a tree; it takes at least three pages (hence its file size for an almost empty table).

The free space map is provided for both tables and indexes. But since an index row cannot be added into an arbitrary page (for example, B-trees define the place of insertion by the sort order), PostgreSQL tracks only those pages that have been fully emptied and can be reused in the index structure.

The visibility map[1] can quickly show whether a page needs to be vacuumed or frozen. For this purpose, it provides two bits for each table page.

The first bit is set for pages that contain only up-to-date row versions. Vacuum skips such pages because there is nothing to clean up. Besides, when a transaction tries to read a row from such a page, there is no point in checking its visibility, so an index-only scan can be used. p. 124 p. 381

The second bit is set for pages that contain only frozen row versions. I will use the term *freeze map* to refer to this part of the fork. v. 9.6 p. 143

Visibility map files have the _vm suffix. They are usually the smallest ones:

```
=> SELECT size
FROM pg_stat_file('/usr/local/pgsql/data/base/16384/16385_vm');
 size
------
 8192
(1 row)
```

The visibility map is provided for tables, but not for indexes. p. 84

[1] postgresql.org/docs/14/storage-vm.html

Pages

p. 70 To facilitate I/O, all files are logically split into *pages* (or *blocks*), which represent the minimum amount of data that can be read or written. Consequently, many internal PostgreSQL algorithms are tuned for page processing.

The page size is usually 8 kB. It can be configured to some extent (up to 32 kB), but only at build time (./configure --with-blocksize), and nobody usually does it. Once built and launched, the instance can work only with pages of the same size; it is impossible to create tablespaces that support different page sizes.

Regardless of the fork they belong to, all the files are handled by the server in p. 169 roughly the same way. Pages are first moved to the buffer cache (where they can be read and updated by processes) and then flushed back to disk as required.

TOAST

Each row must fit a single page: there is no way to continue a row on the next page. To store long rows, PostgreSQL uses a special mechanism called TOAST[1] (The Oversized Attributes Storage Technique).

TOAST implies several strategies. You can move long attribute values into a separate service table, having sliced them into smaller "toasts." Another option is to compress a long value in such a way that the row fits the page. Or you can do both: first compress the value, and then slice and move it.

If the main table contains potentially long attributes, a separate TOAST table is created for it right away, one for all the attributes. For example, if a table has a column of the numeric or text type, a TOAST table will be created even if this column will never store any long values.

For indexes, the TOAST mechanism can offer only compression; moving long attributes into a separate table is not supported. It limits the size of the keys that can p. 357 be indexed (the actual implementation depends on a particular operator class).

[1] postgresql.org/docs/14/storage-toast.html
include/access/heaptoast.h

By default, the TOAST strategy is selected based on the data type of a column. The easiest way to review the used strategies is to run the \d+ command in psql, but I will query the system catalog to get an uncluttered output:

```
=> SELECT attname, atttypid::regtype,
  CASE attstorage
    WHEN 'p' THEN 'plain'
    WHEN 'e' THEN 'external'
    WHEN 'm' THEN 'main'
    WHEN 'x' THEN 'extended'
  END AS storage
FROM pg_attribute
WHERE attrelid = 't'::regclass AND attnum > 0;
 attname | atttypid | storage
---------+----------+----------
 a       | integer  | plain
 b       | numeric  | main
 c       | text     | extended
 d       | json     | extended
(4 rows)
```

PostgreSQL supports the following strategies:

plain means that TOAST is not used (this strategy is applied to data types that are known to be "short," such as the integer type).

extended allows both compressing attributes and storing them in a separate TOAST table.

external implies that long attributes are stored in the TOAST table in an uncompressed state.

main requires long attributes to be compressed first; they will be moved to the TOAST table only if compression did not help.

In general terms, the algorithm looks as follows.[1] PostgreSQL aims at having at least four rows in a page. So if the size of the row exceeds one fourth of the page, excluding the header (for a standard-size page it is about 2000 bytes), we must apply the TOAST mechanism to some of the values. Following the workflow described below, we stop as soon as the row length does not exceed the threshold anymore:

[1] backend/access/heap/heaptoast.c

1. First of all, we go through attributes with external and extended strategies, starting from the longest ones. Extended attributes get compressed, and if the resulting value (on its own, without taking other attributes into account) exceeds one fourth of the page, it is moved to the TOAST table right away. External attributes are handled in the same way, except that the compression stage is skipped.

2. If the row still does not fit the page after the first pass, we move the remaining attributes that use external or extended strategies into the TOAST table, one by one.

3. If it did not help either, we try to compress the attributes that use the main strategy, keeping them in the table page.

4. If the row is still not short enough, the main attributes are moved into the TOAST table.

v. 11 The threshold value is 2000 bytes, but it can be redefined at the table level using the *toast_tuple_target* storage parameter.

It may sometimes be useful to change the default strategy for some of the columns. If it is known in advance that the data in a particular column cannot be compressed (for example, the column stores JPEG images), you can set the external strategy for this column; it allows you to avoid futile attempts to compress the data. The strategy can be changed as follows:

```
=> ALTER TABLE t ALTER COLUMN d SET STORAGE external;
```

If we repeat the query, we will get the following result:

```
 attname | atttypid | storage
---------+----------+----------
 a       | integer  | plain
 b       | numeric  | main
 c       | text     | extended
 d       | json     | external
(4 rows)
```

TOAST tables reside in a separate schema called pg_toast; it is not included into the search path, so TOAST tables are usually hidden. For temporary tables, pg_toast_temp_*N* schemas are used, by analogy with pg_temp_*N*.

Let's take a look at the inner mechanics of the process. Suppose table t contains three potentially long attributes; it means that there must be a corresponding TOAST table. Here it is:

```
=> SELECT relnamespace::regnamespace, relname
FROM pg_class
WHERE oid = (
  SELECT reltoastrelid
  FROM pg_class WHERE relname = 't'
);
 relnamespace |    relname
--------------+----------------
 pg_toast     | pg_toast_16385
(1 row)

=> \d+ pg_toast.pg_toast_16385
TOAST table "pg_toast.pg_toast_16385"
   Column   |  Type   | Storage
------------+---------+---------
 chunk_id   | oid     | plain
 chunk_seq  | integer | plain
 chunk_data | bytea   | plain
Owning table: "public.t"
Indexes:
    "pg_toast_16385_index" PRIMARY KEY, btree (chunk_id, chunk_seq)
Access method: heap
```

It is only logical that the resulting chunks of the toasted row use the plain strategy: there is no second-level TOAST.

Apart from the TOAST table itself, PostgreSQL creates the corresponding index in the same schema. This index is *always* used to access TOAST chunks. The name of the index is displayed in the output, but you can also view it by running the following query:

```
=> SELECT indexrelid::regclass FROM pg_index
WHERE indrelid = (
  SELECT oid
  FROM pg_class WHERE relname = 'pg_toast_16385'
);
          indexrelid
-------------------------------
 pg_toast.pg_toast_16385_index
(1 row)
```

```
=> \d pg_toast.pg_toast_16385_index
Unlogged index "pg_toast.pg_toast_16385_index"
  Column   |  Type   | Key? | Definition
-----------+---------+------+------------
 chunk_id  | oid     | yes  | chunk_id
 chunk_seq | integer | yes  | chunk_seq
primary key, btree, for table "pg_toast.pg_toast_16385"
```

Thus, a TOAST table increases the minimum number of fork files used by the table up to eight: three for the main table, three for the TOAST table, and two for the TOAST index.

Column c uses the extended strategy, so its values will be compressed:

```
=> UPDATE t SET c = repeat('A',5000);
=> SELECT * FROM pg_toast.pg_toast_16385;
 chunk_id | chunk_seq | chunk_data
----------+-----------+------------
(0 rows)
```

The TOAST table is empty: repeated symbols have been compressed by the LZ algorithm, so the value fits the table page.

And now let's construct this value of random symbols:

```
=> UPDATE t SET c = (
  SELECT string_agg( chr(trunc(65+random()*26)::integer), '')
  FROM generate_series(1,5000)
)
RETURNING left(c,10) || '...' || right(c,10);
        ?column?
-------------------------
 LIDPTCFYKM...YZSXTPJBVN
(1 row)
UPDATE 1
```

This sequence cannot be compressed, so it gets into the TOAST table:

```
=> SELECT chunk_id,
  chunk_seq,
  length(chunk_data),
  left(encode(chunk_data,'escape')::text, 10) || '...' ||
  right(encode(chunk_data,'escape')::text, 10)
FROM pg_toast.pg_toast_16385;
```

```
 chunk_id | chunk_seq | length |        ?column?
----------+-----------+--------+------------------------
    16390 |         0 |   1996 | LIDPTCFYKM...NLEHTFPEYD
    16390 |         1 |   1996 | JFHGWVQWCO...PEQZGVCSID
    16390 |         2 |   1008 | EMIXMJJHXQ...YZSXTPJBVN
(3 rows)
```

We can see that the characters are sliced into chunks. The chunk size is selected in such a way that the page of the TOAST table can accommodate four rows. This value varies a little from version to version depending on the size of the page header.

When a long attribute is accessed, PostgreSQL automatically restores the original value and returns it to the client; it all happens seamlessly for the application. If long attributes do not participate in the query, the TOAST table will not be read at all. It is one of the reasons why you should avoid using the asterisk in production solutions.

If the client queries one of the first chunks of a long value, PostgreSQL will read the required chunks only, even if the value has been compressed. v. 13

Nevertheless, data compression and slicing require a lot of resources; the same goes for restoring the original values. That's why it is not a good idea to keep bulky data in PostgreSQL, especially if this data is being actively used and does not require transactional logic (like scanned accounting documents). A potentially better alternative is to store such data in the file system, keeping in the database only the names of the corresponding files. But then the database system cannot guarantee data consistency.

1.2 Processes and Memory

A PostgreSQL server instance consists of several interacting processes.

The first process launched at the server start is postgres, which is traditionally called postmaster. It spawns all the other processes (Unix-like systems use the fork system call for this purpose) and supervises them: if any process fails, postmaster restarts it (or the whole server if there is a risk that the shared data has been damaged).

Because of its simplicity, the process model has been used in PostgreSQL from the very beginning, and ever since there have been unending discussions about switching over to threads.

The current model has several drawbacks: static shared memory allocation does not allow resizing structures like buffer cache on the fly; parallel algorithms are hard to implement and less efficient than they could be; sessions are tightly bound to processes. Using threads sounds promising, even though it involves some challenges related to isolation, OS compatibility, and resource management. However, their implementation would require a radical code overhaul and years of work, so conservative views prevail for now: no such changes are expected in the near future.

Server operation is maintained by background processes. Here are the main ones:

startup restores the system after a failure.

p. 127 **autovacuum** removes stale data from tables and indexes.

p. 210 **wal writer** writes WAL entries to disk.

p. 196 **checkpointer** executes checkpoints.

p. 202 **writer** flushes dirty pages to disk.

stats collector collects usage statistics for the instance.

wal sender sends WAL entries to a replica.

wal receiver gets WAL entries on a replica.

Some of these processes are terminated once the task is complete, others run in the background all the time, and some can be switched off.

Each process is managed by configuration parameters, sometimes by dozens of them. To set up the server in a comprehensive manner, you have to be aware of its inner workings. But general considerations will only help you select more or less adequate initial values; later on, these settings have to be fine-tuned based on monitoring data.

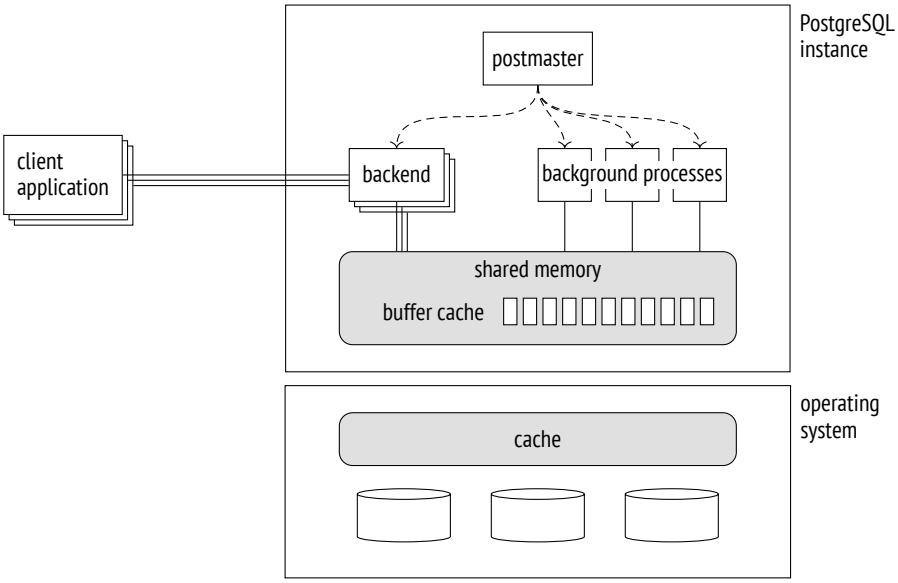
To enable process interaction, postmaster allocates *shared memory*, which is available to all the processes.

Since disks (especially HDD, but SSD too) are much slower than RAM, PostgreSQL
p. 169 uses caching: some part of the shared RAM is reserved for recently read pages, in hope that they will be needed more than once and the overhead of repeated disk

access will be reduced. Modified data is also flushed to disk after some delay, not immediately.

Buffer cache takes the greater part of the shared memory, which also contains other buffers used by the server to speed up disk access.

The operating system has its own cache too. PostgreSQL (almost) never bypasses the operating system mechanisms to use direct I/O, so it results in double caching.



In case of a failure (such as a power outage or an operating system crash), the data kept in RAM is lost, including that of the buffer cache. The files that remain on disk have their pages written at different points in time. To be able to restore data consistency, PostgreSQL maintains the *write-ahead log* (WAL) during its operation, which makes it possible to repeat lost operations when necessary. *p. 189*

1.3 Clients and the Client-Server Protocol

Another task of the postmaster process is to listen for incoming connections. Once a new client appears, postmaster spawns a separate *backend process*.[1] The client

[1] backend/tcop/postgres.c, PostgresMain function

establishes a connection and starts a *session* with this backend. The session continues until the client disconnects or the connection is lost.

The server has to spawn a separate backend for each client. If many clients are trying to connect, it can turn out to be a problem.

p. 302
p. 301

- Each process needs RAM to cache catalog tables, prepared statements, intermediate query results, and other data. The more connections are open, the more memory is required.
- If connections are short and frequent (a client performs a small query and disconnects), the cost of establishing a connection, spawning a new process, and performing pointless local caching is unreasonably high.
- The more processes are started, the more time is required to scan their list, and this operation is performed very often. As a result, performance may decline as the number of clients grows.

p. 94

This problem can be resolved by *connection pooling*, which limits the number of spawned backends. PostgreSQL has no such built-in functionality, so we have to rely on third-party solutions: pooling managers integrated into the application server or external tools (such as PgBouncer[1] or Odyssey[2]). This approach usually means that each server backend can execute transactions of different clients, one after another. It imposes some restrictions on application development since it is only allowed to use resources that are local to a transaction, not to the whole session.

To understand each other, a client and a server must use one and the same interfacing protocol.[3] It is usually based on the standard libpq library, but there are also other custom implementations.

Speaking in the most general terms, the protocol allows clients to connect to the server and execute SQL queries.

A connection is always established to a particular database on behalf of a particular role, or user. Although the server supports a database cluster, it is required to establish a separate connection to each database that you would like to use in your

[1] pgbouncer.org
[2] github.com/yandex/odyssey
[3] postgresql.org/docs/14/protocol.html

application. At this point, *authentication* is performed: the backend process verifies the user's identity (for example, by asking for the password) and checks whether this user has the right to connect to the server and to the specified database.

SQL queries are passed to the backend process as text strings. The process parses the text, optimizes the query, executes it, and returns the result to the client.

Part I

Isolation and MVCC

2

Isolation

2.1 Consistency

The key feature of relational databases is their ability to ensure data *consistency*, that is, data *correctness*.

It is a known fact that at the database level it is possible to create *integrity constraints*, such as NOT NULL or UNIQUE. The database system ensures that these constraints are never broken, so data integrity is never compromised.

If all the required constraints could be formulated at the database level, consistency would be guaranteed. But some conditions are too complex for that, for example, they touch upon several tables at once. And even if a constraint can be defined in the database, but for some reason it is not, it does not mean that this constraint may be violated.

Thus, data consistency is stricter than integrity, but the database system has no idea what "consistency" actually means. If an application breaks it without breaking the integrity, there is no way for the database system to find out. Consequently, it is the application that must lay down the criteria for data consistency, and we have to believe that it is written correctly and will never have any errors.

But if the application always executes only correct sequences of operators, where does the database system come into play?

First of all, a correct sequence of operators can temporarily break data consistency, and—strange as it may seem—it is perfectly normal.

A hackneyed but clear example is a transfer of funds from one account to another. A consistency rule may sound as follows: *a money transfer must never change the total*

balance of the affected accounts. It is quite difficult (although possible) to formulate this rule as an integrity constraint in SQL, so let's assume that it is defined at the application level and remains opaque to the database system. A transfer consists of two operations: the first one draws some money from one of the accounts, whereas the second one adds this sum to another account. The first operation breaks data consistency, whereas the second one restores it.

If the first operation succeeds, but the second one does not (because of some failure), data consistency will be broken. Such situations are unacceptable, but it takes a great deal of effort to detect and address them at the application level. Luckily it is not required—the problem can be completely solved by the database system itself if it knows that these two operations constitute an indivisible whole, that is, a *transaction*.

But there is also a more subtle aspect here. Being absolutely correct on their own, transactions can start operating incorrectly when run in parallel. That's because operations belonging to different transactions often get intermixed. There would be no such issues if the database system first completed all operations of one transaction and then moved on to the next one, but performance of sequential execution would be implausibly low.

> A truly simultaneous execution of transactions can only be achieved on systems with suitable hardware: a multi-core processor, a disk array, and so on. But the same reasoning is also true for a server that executes commands sequentially in the time-sharing mode. For generalization purposes, both these situations are sometimes referred to as *concurrent execution*.

Correct transactions that behave incorrectly when run together result in concurrency *anomalies*, or *phenomena*.

Here is a simple example. To get consistent data from the database, the application must not see any changes made by other uncommitted transactions, at the very minimum. Otherwise (if some transactions are rolled back), it would see the database state that has never existed. Such an anomaly is called a *dirty read*. There are also many other anomalies, which are more complex.

When running transactions concurrently, the database must guarantee that the result of such execution will be the same as the outcome of one of the possible se-

quential executions. In other words, it must *isolate* transactions from one another, thus taking care of any possible anomalies.

To sum it up, a transaction is a set of operations that takes the database from one correct state to another correct state (*consistency*), provided that it is executed in full (*atomicity*) and without being affected by other transactions (*isolation*). This definition combines the requirements implied by the first three letters of the ACID acronym. They are so intertwined that it makes sense to discuss them together. In fact, the durability requirement is hardly possible to split off either: after a crash, the system may still contain some changes made by uncommitted transactions, and you have to do something about it to restore data consistency. *p. 189*

Thus, the database system helps the application maintain data consistency by taking transaction boundaries into account, even though it has no idea about the implied consistency rules.

Unfortunately, full isolation is hard to implement and can negatively affect performance. Most real-life systems use weaker isolation levels, which prevent some anomalies, but not all of them. It means that the job of maintaining data consistency partially falls on the application. And that's exactly why it is very important to understand which isolation level is used in the system, what is guaranteed at this level and what is not, and how to ensure that your code will be correct in such conditions.

2.2 Isolation Levels and Anomalies Defined by the SQL Standard

The SQL standard specifies four isolation levels.[1] These levels are defined by the list of anomalies that may or may not occur during concurrent transaction execution. So when talking about isolation levels, we have to start with anomalies.

We should bear in mind that the standard is a theoretical construct: it affects the practice, but the practice still diverges from it in lots of ways. That's why all ex-

[1] postgresql.org/docs/14/transaction-iso.html

amples here are rather hypothetical. Dealing with transactions on bank accounts, these examples are quite self-explanatory, but I have to admit that they have nothing to do with real banking operations.

It is interesting that the actual database theory also diverges from the standard: it was developed after the standard had been adopted, and the practice was already well ahead.

Lost Update

The *lost update* anomaly occurs when two transactions read one and the same table row, then one of the transactions updates this row, and finally the other transaction updates the same row without taking into account any changes made by the first transaction.

Suppose that two transactions are going to increase the balance of one and the same account by $100. The first transaction reads the current value ($1,000), then the second transaction reads the same value. The first transaction increases the balance (making it $1,100) and writes the new value into the database. The second transaction does the same: it gets $1,100 after increasing the balance and writes this value. As a result, the customer loses $100.

Lost updates are forbidden by the standard at all isolation levels.

Dirty Reads and Read Uncommitted

The *dirty read* anomaly occurs when a transaction reads uncommitted changes made by another transaction.

For example, the first transaction transfers $100 to an empty account but does not commit this change. Another transaction reads the account state (which has been updated but not committed) and allows the customer to withdraw the money—even though the first transaction gets interrupted and its changes are rolled back, so the account is empty.

The standard allows dirty reads at the Read Uncommitted level.

Non-Repeatable Reads and Read Committed

The *non-repeatable read* anomaly occurs when a transaction reads one and the same row twice, whereas another transaction updates (or deletes) this row between these reads and commits the change. As a result, the first transaction gets different results.

For example, suppose there is a consistency rule that *forbids having a negative balance in bank accounts*. The first transaction is going to reduce the account balance by $100. It checks the current value, gets $1,000, and decides that this operation is possible. At the same time, another transaction withdraws all the money from this account and commits the changes. If the first transaction checked the balance again at this point, it would get $0 (but the decision to withdraw the money is already taken, and this operation causes an overdraft).

The standard allows non-repeatable reads at the Read Uncommitted and Read Committed levels.

Phantom Reads and Repeatable Read

The *phantom read* anomaly occurs when one and the same transaction executes two identical queries returning a set of rows that satisfy a particular condition, while another transaction adds some other rows satisfying this condition and commits the changes in the time interval between these queries. As a result, the first transaction gets two different sets of rows.

For example, suppose there is a consistency rule that *forbids a customer to have more than three accounts*. The first transaction is going to open a new account, so it checks how many accounts are currently available (let's say there are two of them) and decides that this operation is possible. At this very moment, the second transaction also opens a new account for this client and commits the changes. If the first transaction double-checked the number of open accounts, it would get three (but it is already opening another account, and the client ends up having four of them).

The standard allows phantom reads at the Read Uncommitted, Read Committed, and Repeatable Read isolation levels.

No Anomalies and Serializable

The standard also defines the Serializable level, which does not allow any anomalies. It is not the same as the ban on lost updates and dirty, non-repeatable, and phantom reads. In fact, there is a much higher number of known anomalies than the standard specifies, and an unknown number of still unknown ones.

The Serializable level must prevent *any* anomalies. It means that the application developer does not have to take isolation into account. If transactions execute correct operator sequences when run on their own, concurrent execution cannot break data consistency either.

To illustrate this idea, I will use a well-known table provided in the standard; the last column is added here for clarity:

	lost update	dirty read	non-repeatable read	phantom read	other anomalies
Read Uncommitted	–	yes	yes	yes	yes
Read Committed	–	–	yes	yes	yes
Repeatable Read	–	–	–	yes	yes
Serializable	–	–	–	–	–

Why These Anomalies?

Of all the possible anomalies, why does the standard mentions only some, and why exactly these ones?

No one seems to know it for sure. But it is not unlikely that other anomalies were simply not considered when the first versions of the standard were adopted, as theory was far behind practice at that time.

Besides, it was assumed that isolation had to be based on locks. The widely used *two-phase locking protocol* (2PL) requires transactions to lock the affected rows during execution and release the locks upon completion. In simplistic terms, the more locks a transaction acquires, the better it is isolated from other transactions. And consequently, the worse is the system performance, as transactions start queuing to get access to the same rows instead of running concurrently.

I believe that to a great extent the difference between the standard isolation levels is defined by the number of locks required for their implementation.

If the rows to be updated are locked for writes but not for reads, we get the Read Uncommitted isolation level, which allows reading data before it is committed.

If the rows to be updated are locked for both reads and writes, we get the Read Committed level: it is forbidden to read uncommitted data, but a query can return different values if it is run more than once (non-repeatable reads).

Locking the rows to be read and to be updated for all operations gives us the Repeatable Read level: a repeated query will return the same result.

However, the Serializable level poses a problem: it is impossible to lock a row that does not exist yet. It leaves an opportunity for phantom reads to occur: a transaction can add a row that satisfies the condition of the previous query, and this row will appear in the next query result.

Thus, regular locks cannot provide full isolation: to achieve it, we have to lock conditions (predicates) rather than rows. Such *predicate* locks were introduced as early as 1976 when System R was being developed; however, their practical applicability is limited to simple conditions for which it is clear whether two different predicates may conflict. As far as I know, predicate locks in their intended form have never been implemented in any system. *p. 268*

2.3 Isolation Levels in PostgreSQL

Over time, lock-based protocols for transaction management got replaced with the *Snapshot Isolation* (SI) protocol. The idea behind this approach is that each transaction accesses a consistent snapshot of data as it appeared at a particular point in time. The snapshot includes all the current changes committed before the snapshot was taken.

Snapshot isolation minimizes the number of required locks. In fact, a row will be locked only by concurrent update attempts. In all other cases, operations can be executed concurrently: writes never lock reads, and reads never lock anything. *p. 239*

PostgreSQL uses a *multiversion* flavor of the SI protocol. Multiversion concurrency control implies that at any moment the database system can contain several versions of one and the same row, so PostgreSQL can include an appropriate version into the snapshot rather than abort transactions that attempt to read stale data.

Based on snapshots, PostgreSQL isolation differs from the requirements specified in the standard—in fact, it is even stricter. Dirty reads are forbidden by design. Technically, you can specify the Read Uncommitted level, but its behavior will be the same as that of Read Committed, so I am not going to mention this level anymore.
p. 154 Repeatable Read allows neither non-repeatable nor phantom reads (even though it does not guarantee full isolation). But *in some cases*, there is a risk of losing changes at the Read Committed level.

	lost updates	dirty reads	non-repeatable reads	phantom reads	other anomalies
Read Committed	yes	–	yes	yes	yes
Repeatable Read	–	–	–	–	yes
Serializable	–	–	–	–	–

p. 92 Before exploring the internal mechanisms of isolation, let's discuss each of the three isolation levels from the user's perspective.

For this purpose, we are going to create the accounts table; Alice and Bob will have $1,000 each, but Bob will have two accounts:

```
=> CREATE TABLE accounts(
  id integer PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
  client text,
  amount numeric
);
=> INSERT INTO accounts VALUES
  (1, 'alice', 1000.00), (2, 'bob', 100.00), (3, 'bob', 900.00);
```

Read Committed

No dirty reads. It is easy to check that reading dirty data is not allowed. Let's start a transaction. By default, it uses the Read Committed[1] isolation level:

[1] postgresql.org/docs/14/transaction-iso.html#XACT-READ-COMMITTED

```
=> BEGIN;
=> SHOW transaction_isolation;
 transaction_isolation
-----------------------
 read committed
(1 row)
```

To be more exact, the default level is set by the following parameter, which can be changed as required:

```
=> SHOW default_transaction_isolation;
 default_transaction_isolation
-------------------------------
 read committed
(1 row)
```

The opened transaction withdraws some funds from the customer account but does not commit these changes yet. It will see its own changes though, as it is always allowed:

```
=> UPDATE accounts SET amount = amount - 200 WHERE id = 1;
=> SELECT * FROM accounts WHERE client = 'alice';
 id | client | amount
----+--------+--------
  1 | alice  | 800.00
(1 row)
```

In the second session, we start another transaction that will also run at the Read Committed level:

```
=> BEGIN;
=> SELECT * FROM accounts WHERE client = 'alice';
 id | client | amount
----+--------+---------
  1 | alice  | 1000.00
(1 row)
```

Predictably, the second transaction does not see any uncommitted changes—dirty reads are forbidden.

Non-repeatable reads. Now let the first transaction commit the changes. Then the second transaction will repeat the same query:

```
=> COMMIT;
```

```
=> SELECT * FROM accounts WHERE client = 'alice';
 id | client | amount
----+--------+--------
  1 | alice  | 800.00
(1 row)
=> COMMIT;
```

The query receives an updated version of the data—and it is exactly what is understood by the *non-repeatable read* anomaly, which is allowed at the Read Committed level.

A practical insight: in a transaction, you must not take any decisions based on the data read by the previous operator, as everything can change in between. Here is an example whose variations appear in the application code so often that it can be considered a classic anti-pattern:

```
IF (SELECT amount FROM accounts WHERE id = 1) >= 1000 THEN
  UPDATE accounts SET amount = amount - 1000 WHERE id = 1;
END IF;
```

During the time that passes between the check and the update, other transactions can freely change the state of the account, so such a "check" is absolutely useless. For better understanding, you can imagine that random operators of other transactions are "wedged" between the operators of the current transaction. For example, like this:

```
IF (SELECT amount FROM accounts WHERE id = 1) >= 1000 THEN
```

```
    UPDATE accounts SET amount = amount - 200 WHERE id = 1;
    COMMIT;
```

```
  UPDATE accounts SET amount = amount - 1000 WHERE id = 1;
END IF;
```

If everything goes wrong as soon as the operators are rearranged, then the code is incorrect. Do not delude yourself that you will never get into this trouble: anything that can go wrong will go wrong. Such errors are very hard to reproduce, and consequently, fixing them is a real challenge.

How can you correct this code? There are several options:

- Replace procedural code with declarative one.

  For example, in this particular case it is easy to turn an IF statement into a CHECK constraint:

  ```
  ALTER TABLE accounts
    ADD CHECK amount >= 0;
  ```

  Now you do not need any checks in the code: it is enough to simply run the command and handle the exception that will be raised if an integrity constraint violation is attempted.

- Use a single SQL operator.

  Data consistency can be compromised if a transaction gets committed within the time gap between operators of another transaction, thus changing data visibility. If there is only one operator, there are no such gaps.

  PostgreSQL has enough capabilities to solve complex tasks with a single SQL statement. In particular, it offers common table expressions (CTE) that can contain operators like INSERT, UPDATE, DELETE, as well as the INSERT ON CONFLICT operator that implements the following logic: insert the row if it does not exist, otherwise perform an update.

- Apply explicit locks.

  The last resort is to manually set an exclusive lock on all the required rows (SELECT FOR UPDATE) or even on the whole table (LOCK TABLE). This approach always works, but it nullifies all the advantages of MVCC: some operations that could be executed concurrently will run sequentially.

p. 239
p. 232

Read skew. However, it is not all that simple. The PostgreSQL implementation allows other, less known anomalies, which are not regulated by the standard.

Suppose the first transaction has started a money transfer between Bob's accounts:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount - 100 WHERE id = 2;
```

Meanwhile, the other transaction starts looping through all Bob's accounts to calculate their total balance. It begins with the first account (seeing its previous state, of course):

```
=> BEGIN;
=> SELECT amount FROM accounts WHERE id = 2;
 amount
--------
 100.00
(1 row)
```

At this moment, the first transaction completes successfully:

```
=> UPDATE accounts SET amount = amount + 100 WHERE id = 3;
=> COMMIT;
```

The second transaction reads the state of the second account (and sees the already updated value):

```
=> SELECT amount FROM accounts WHERE id = 3;
 amount
---------
 1000.00
(1 row)
=> COMMIT;
```

As a result, the second transaction gets $1,100 because it has read incorrect data. Such an anomaly is called *read skew*.

How can you avoid this anomaly at the Read Committed level? The answer is obvious: use a single operator. For example, like this:

```
SELECT sum(amount) FROM accounts WHERE client = 'bob';
```

I have been stating so far that data visibility can change only between operators, but is it really so? What if the query is running for a long time? Can it see different parts of data in different states in this case?

Let's check it out. A convenient way to do it is to add a delay to an operator by calling the pg_sleep function. Then the first row will be read at once, but the second row will have to wait for two seconds:

```
=> SELECT amount, pg_sleep(2) -- two seconds
FROM accounts WHERE client = 'bob';
```

While this statement is being executed, let's start another transaction to transfer the money back:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100 WHERE id = 2;
=> UPDATE accounts SET amount = amount - 100 WHERE id = 3;
=> COMMIT;
```

The result shows that the operator has seen all the data in the state that corresponds to the beginning of its execution, which is certainly correct:

```
 amount  | pg_sleep
---------+----------
    0.00 |
 1000.00 |
(2 rows)
```

But it is not all that simple either. If the query contains a function that is declared VOLATILE, and this function executes another query, then the data seen by this nested query will not be consistent with the result of the main query.

Let's check the balance in Bob's accounts using the following function:

```
=> CREATE FUNCTION get_amount(id integer) RETURNS numeric
AS $$
  SELECT amount FROM accounts a WHERE a.id = get_amount.id;
$$ VOLATILE LANGUAGE sql;
=> SELECT get_amount(id), pg_sleep(2)
FROM accounts WHERE client = 'bob';
```

We will transfer the money between the accounts once again while our delayed query is being executed:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100 WHERE id = 2;
=> UPDATE accounts SET amount = amount - 100 WHERE id = 3;
=> COMMIT;
```

In this case, we are going to get inconsistent data—$100 has been lost:

```
 get_amount | pg_sleep
------------+----------
     100.00 |
     800.00 |
(2 rows)
```

I would like to emphasize that this effect is possible only at the Read Committed isolation level, and only if the function is VOLATILE. The trouble is that PostgreSQL uses exactly this isolation level and this volatility category by default. So we have to admit that the trap is set in a very cunning way.

Read skew instead of lost updates. The read skew anomaly can also occur within a single operator during an update—even though in a somewhat unexpected way.

Let's see what happens if two transactions try to modify one and the same row. Bob currently has a total of $1,000 in two accounts:

```
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client | amount
----+--------+--------
  2 | bob    | 200.00
  3 | bob    | 800.00
(2 rows)
```

Start a transaction that will reduce Bob's balance:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount - 100 WHERE id = 3;
```

At the same time, the other transaction will be calculating the interest for all customer accounts with the total balance of $1,000 or more:

```
=> UPDATE accounts SET amount = amount * 1.01
WHERE client IN (
  SELECT client
  FROM accounts
  GROUP BY client
  HAVING sum(amount) >= 1000
);
```

The UPDATE operator execution virtually consists of two stages. First, the rows to be updated are selected based on the provided condition. Since the first transaction is not committed yet, the second transaction cannot see its result, so the selection of rows picked for interest accrual is not affected. Thus, Bob's accounts satisfy the condition, and his balance must be increased by $10 once the UPDATE operation completes.

At the second stage, the selected rows are updated one by one. The second transaction has to wait because the row with id = 3 is locked: it is being updated by the first transaction.

Meanwhile, the first transaction commits its changes:

```
=> COMMIT;
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client |  amount
----+--------+----------
  2 | bob    | 202.0000
  3 | bob    | 707.0000
(2 rows)
```

On the one hand, the UPDATE command must not see any changes made by the first transaction. But on the other hand, it must not lose any committed changes.

Once the lock is released, the UPDATE operator *re-reads* the row to be updated (but only this row!). As a result, Bob gets $9 of interest, based on the total of $900. But if he had $900, his accounts should not have been included into the query results in the first place. *p. 249*

Thus, our transaction has returned incorrect data: different rows have been read from different snapshots. Instead of a lost update, we observe the read skew anomaly again.

Lost updates. However, the trick of re-reading the locked row will not help against lost updates if the data is modified by different SQL operators.

p. 46 Here is an example that we have already seen. The application reads and registers (outside of the database) the current balance of Alice's account:

```
=> BEGIN;
=> SELECT amount FROM accounts WHERE id = 1;
 amount
--------
 800.00
(1 row)
```

Meanwhile, the other transaction does the same:

```
=> BEGIN;
=> SELECT amount FROM accounts WHERE id = 1;
 amount
--------
 800.00
(1 row)
```

The first transaction increases the previously registered value by $100 and commits this change:

```
=> UPDATE accounts SET amount = 800.00 + 100 WHERE id = 1
RETURNING amount;
 amount
--------
 900.00
(1 row)
UPDATE 1
=> COMMIT;
```

The second transaction does the same:

```
=> UPDATE accounts SET amount = 800.00 + 100 WHERE id = 1
RETURNING amount;
 amount
--------
 900.00
(1 row)
UPDATE 1
```

```
=> COMMIT;
```

Unfortunately, Alice has lost $100. The database system does not know that the registered value of $800 is somehow related to accounts.amount, so it cannot prevent the lost update anomaly. At the Read Committed isolation level, this code is incorrect.

Repeatable Read

No non-repeatable and phantom reads. As its name suggests, the Repeatable Read[1] isolation level must guarantee repeatable reading. Let's check it and make sure that phantom reads cannot occur either. For this purpose, we are going to start a transaction that will revert Bob's accounts to their previous state and create a new account for Charlie:

```
=> BEGIN;

=> UPDATE accounts SET amount = 200.00 WHERE id = 2;
=> UPDATE accounts SET amount = 800.00 WHERE id = 3;
=> INSERT INTO accounts VALUES
  (4, 'charlie', 100.00);

=> SELECT * FROM accounts ORDER BY id;
 id | client  | amount
----+---------+--------
  1 | alice   | 900.00
  2 | bob     | 200.00
  3 | bob     | 800.00
  4 | charlie | 100.00
(4 rows)
```

In the second session, let's start another transaction, with the Repeatable Read level explicitly specified in the BEGIN command (the level of the first transaction is not important):

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT * FROM accounts ORDER BY id;
```

[1] postgresql.org/docs/14/transaction-iso.html#XACT-REPEATABLE-READ

```
 id | client |  amount
----+--------+----------
  1 | alice  |   900.00
  2 | bob    | 202.0000
  3 | bob    | 707.0000
(3 rows)
```

Now the first transaction commits its changes, and the second transaction repeats the same query:

```
=> COMMIT;
```

```
=> SELECT * FROM accounts ORDER BY id;
 id | client |  amount
----+--------+----------
  1 | alice  |   900.00
  2 | bob    | 202.0000
  3 | bob    | 707.0000
(3 rows)
=> COMMIT;
```

The second transaction still sees the same data as before: neither new rows nor row updates are visible. At this isolation level, you do not have to worry that something will change between operators.

p. 56 **Serialization failures instead of lost updates.** As we have already seen, if two transactions update one and the same row at the Read Committed level, it can cause the read skew anomaly: the waiting transaction has to re-read the locked row, so it sees the state of this row at a different point in time as compared to other rows.

Such an anomaly is not allowed at the Repeatable Read isolation level, and if it does happen, the transaction can only be aborted with a serialization failure. Let's check it out by repeating the scenario with interest accrual:

```
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client | amount
----+--------+--------
  2 | bob    | 200.00
  3 | bob    | 800.00
(2 rows)
=> BEGIN;
```

```
=> UPDATE accounts SET amount = amount - 100.00 WHERE id = 3;
```

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> UPDATE accounts SET amount = amount * 1.01
WHERE client IN (
  SELECT client
  FROM accounts
  GROUP BY client
  HAVING sum(amount) >= 1000
);
```

```
=> COMMIT;
```

```
ERROR:  could not serialize access due to concurrent update
=> ROLLBACK;
```

The data remains consistent:

```
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client | amount
----+--------+--------
  2 | bob    | 200.00
  3 | bob    | 700.00
(2 rows)
```

The same error will be raised by any concurrent row updates, even if they affect different columns.

We will also get this error if we try to update the balance based on the previously stored value:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT amount FROM accounts WHERE id = 1;
 amount
--------
 900.00
(1 row)
```

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
=> SELECT amount FROM accounts WHERE id = 1;
 amount
--------
 900.00
(1 row)
```

```
=> UPDATE accounts SET amount = 900.00 + 100.00 WHERE id = 1
RETURNING amount;
 amount
---------
 1000.00
(1 row)
UPDATE 1
=> COMMIT;
```

```
=> UPDATE accounts SET amount = 900.00 + 100.00 WHERE id = 1
RETURNING amount;
ERROR:  could not serialize access due to concurrent update
=> ROLLBACK;
```

A practical insight: if your application is using the Repeatable Read isolation level for write transactions, it must be ready to retry transactions that have been completed with a serialization failure. For read-only transactions, such an outcome is impossible.

Write skew. As we have seen, the PostgreSQL implementation of the Repeatable Read isolation level prevents all the anomalies described in the standard. But not all possible ones: no one knows how many of them exist. However, one important fact is proved for sure: snapshot isolation does not prevent *only two* anomalies, no matter how many other anomalies are out there.

The first one is *write skew*.

Let's define the following consistency rule: *it is allowed to have a negative balance in some of the customer's accounts as long as the total balance is non-negative*.

The first transaction gets the total balance of Bob's accounts:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
=> SELECT sum(amount) FROM accounts WHERE client = 'bob';
  sum
--------
 900.00
(1 row)
```

The second transaction gets the same sum:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT sum(amount) FROM accounts WHERE client = 'bob';
  sum
--------
 900.00
(1 row)
```

The first transaction fairly assumes that it can debit one of the accounts by $600:

```
=> UPDATE accounts SET amount = amount - 600.00 WHERE id = 2;
```

The second transaction comes to the same conclusion, but debits the other account:

```
=> UPDATE accounts SET amount = amount - 600.00 WHERE id = 3;
=> COMMIT;
```

```
=> COMMIT;
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client | amount
----+--------+---------
  2 | bob    | -400.00
  3 | bob    |  100.00
(2 rows)
```

Bob's total balance is now negative, although both transactions would have been correct if run separately.

Read-only transaction anomaly. The *read-only transaction* anomaly is the second and the last one allowed at the Repeatable Read isolation level. To observe this anomaly, we have to run three transactions: two of them are going to update the data, while the third one will be read-only.

But first let's restore Bob's balance:

```
=> UPDATE accounts SET amount = 900.00 WHERE id = 2;
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client | amount
----+--------+--------
  3 | bob    | 100.00
  2 | bob    | 900.00
(2 rows)
```

The first transaction calculates the interest to be accrued on Bob's total balance and adds this sum to one of his accounts:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ; -- 1
=> UPDATE accounts SET amount = amount + (
  SELECT sum(amount) FROM accounts WHERE client = 'bob'
) * 0.01
WHERE id = 2;
```

Then the second transaction withdraws some money from Bob's other account and commits this change:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ; -- 2
=> UPDATE accounts SET amount = amount - 100.00 WHERE id = 3;
=> COMMIT;
```

If the first transaction gets committed at this point, there will be no anomalies: we could assume that the first transaction is committed before the second one (but not vice versa—the first transaction had seen the state of account with id = 3 before any updates were made by the second transaction).

But let's imagine that at this very moment we start a ready-only transaction to query an account that is not affected by the first two transactions:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ; -- 3
=> SELECT * FROM accounts WHERE client = 'alice';
 id | client | amount
----+--------+---------
  1 | alice  | 1000.00
(1 row)
```

And only now will the first transaction get committed:

```
=> COMMIT;
```

Which state should the third transaction see at this point? Having started, it could see the changes made by the second transaction (which had already been committed), but not by the first one (which had not been committed yet). But as we have already established, the second transaction should be treated as if it were started after the first one. Any state seen by the third transaction will be inconsistent—this is exactly what is meant by the read-only transaction anomaly:

```
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client | amount
----+--------+--------
  2 | bob    | 900.00
  3 | bob    |   0.00
(2 rows)
=> COMMIT;
```

Serializable

The Serializable[1] isolation level prevents all possible anomalies. This level is virtually built on top of snapshot isolation. Those anomalies that do not occur at the Repeatable Read isolation level (such as dirty, non-repeatable, or phantom reads) cannot occur at the Serializable level either. And those two anomalies that do occur (write skew and read-only transaction anomalies) get detected in a special way to abort the transaction, causing an already familiar serialization failure.

No anomalies. Let's make sure that our write skew scenario will eventually end with a serialization failure: *p. 62*

```
=> BEGIN ISOLATION LEVEL SERIALIZABLE;
=> SELECT sum(amount) FROM accounts WHERE client = 'bob';
   sum
----------
 910.0000
(1 row)
```

[1] postgresql.org/docs/14/transaction-iso.html#XACT-SERIALIZABLE

```
=> BEGIN ISOLATION LEVEL SERIALIZABLE;
=> SELECT sum(amount) FROM accounts WHERE client = 'bob';
   sum
----------
 910.0000
(1 row)
```

```
=> UPDATE accounts SET amount = amount - 600.00 WHERE id = 2;
```

```
=> UPDATE accounts SET amount = amount - 600.00 WHERE id = 3;
=> COMMIT;
COMMIT
```

```
=> COMMIT;
ERROR:  could not serialize access due to read/write dependencies
among transactions
DETAIL:  Reason code: Canceled on identification as a pivot, during
commit attempt.
HINT:  The transaction might succeed if retried.
```

The scenario with the read-only transaction anomaly will lead to the same error.

Deferring a read-only transaction. To avoid situations when a read-only transaction can cause an anomaly that compromises data consistency, PostgreSQL offers an interesting solution: this transaction can be deferred until its execution becomes safe. It is the only case when a SELECT statement can be blocked by row updates.

We are going to check it out by repeating the scenario that demonstrated the read-only transaction anomaly:

```
=> UPDATE accounts SET amount = 900.00 WHERE id = 2;
=> UPDATE accounts SET amount = 100.00 WHERE id = 3;
=> SELECT * FROM accounts WHERE client = 'bob' ORDER BY id;
 id | client | amount
----+--------+--------
  2 | bob    | 900.00
  3 | bob    | 100.00
(2 rows)
=> BEGIN ISOLATION LEVEL SERIALIZABLE; -- 1
```

```
=> UPDATE accounts SET amount = amount + (
  SELECT sum(amount) FROM accounts WHERE client = 'bob'
) * 0.01
WHERE id = 2;
```

```
=> BEGIN ISOLATION LEVEL SERIALIZABLE; -- 2
=> UPDATE accounts SET amount = amount - 100.00 WHERE id = 3;
=> COMMIT;
```

Let's explicitly declare the third transaction as READ ONLY and DEFERRABLE:

```
=> BEGIN ISOLATION LEVEL SERIALIZABLE READ ONLY DEFERRABLE; -- 3
=> SELECT * FROM accounts WHERE client = 'alice';
```

An attempt to run the query blocks the transaction—otherwise, it would have caused an anomaly.

And only when the first transaction is committed, the third one can continue its execution:

```
=> COMMIT;
```

```
 id | client | amount
----+--------+---------
  1 | alice  | 1000.00
(1 row)
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client |  amount
----+--------+----------
  2 | bob    | 910.0000
  3 | bob    |     0.00
(2 rows)
=> COMMIT;
```

Thus, if an application uses the Serializable isolation level, it must be ready to retry transactions that have ended with a serialization failure. (The Repeatable Read level requires the same approach unless the application is limited to read-only transactions.)

The Serializable isolation level brings ease of programming, but the price you pay is the overhead incurred by anomaly detection and forced termination of a certain

fraction of transactions. You can lower this impact by explicitly using the READ ONLY clause when declaring read-only transactions. But the main questions is, of course, how big the fraction of aborted transactions is—since these transactions will have to be retried. It would have been not so bad if PostgreSQL aborted only those transactions that result in data conflicts and are really incompatible. But such an approach would inevitably be too resource-intensive, as it would involve tracking operations on each row.

p. 268 The current implementation allows false positives: PostgreSQL can abort some absolutely safe transactions that are simply out of luck. Their "luck" depends on many factors, such as the presence of appropriate indexes or the amount of RAM available, so the actual behavior is hard to predict in advance.

If you use the Serializable level, it must be observed by all transactions of the application. When combined with other levels, Serializable behaves as Repeatable Read without any notice. So if you decide to use the Serializable level, it makes sense to modify the *default_transaction_isolation* parameter value accordingly—even though someone can still overwrite it by explicitly setting a different level.

read committed

v. 12 There are also other restrictions; for example, queries run at the Serializable level cannot be executed on replicas. And although the functionality of this level is constantly being improved, the current limitations and overhead make it less attractive.

2.4 Which Isolation Level to Use?

Read Committed is the default isolation level in PostgreSQL, and apparently it is this level that is used in the vast majority of applications. This level can be convenient because it allows aborting transactions only in case of a failure; it does not abort any transactions to preserve data consistency. In other words, serialization failures cannot occur, so you do not have to take care of transaction retries.

The downside of this level is a large number of possible anomalies, which have been discussed in detail above. A developer has to keep them in mind all the time and write the code in a way that prevents their occurrence. If it is impossible to define all the needed actions in a single SQL statement, then you have to resort to explicit locking. The toughest part is that the code is hard to test for errors

related to data inconsistency; such errors can appear in unpredictable and barely reproducible ways, so they are very hard to fix too.

The Repeatable Read isolation level eliminates some of the inconsistency problems, but alas, not all of them. Therefore, you must not only remember about the remaining anomalies, but also modify the application to correctly handle serialization failures, which is certainly inconvenient. However, for read-only transactions this level is a perfect complement to the Read Committed level; it can be very useful for cases like building reports that involve multiple SQL queries.

And finally, the Serializable isolation level allows you not to worry about data consistency at all, which simplifies writing the code to a great extent. The only thing required from the application is the ability to retry any transaction that is aborted with a serialization failure. However, the number of aborted transactions and associated overhead can significantly reduce system throughput. You should also keep in mind that the Serializable level is not supported on replicas and cannot be combined with other isolation levels.

3

Pages and Tuples

3.1 Page Structure

Each page has a certain inner layout that usually consists of the following parts:[1]

- page header
- an array of item pointers
- free space
- items (row versions)
- special space

Page Header

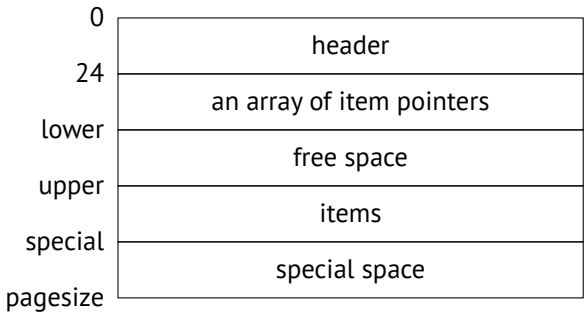
The page *header* is located in the lowest addresses and has a fixed size. It stores various information about the page, such as its checksum and the sizes of all the other parts of the page.

p. 120

These sizes can be easily displayed using the pageinspect extension.[2] Let's take a look at the first page of the table (page numbering is zero-based):

[1] postgresql.org/docs/14/storage-page-layout.html
include/storage/bufpage.h
[2] postgresql.org/docs/14/pageinspect.html

```
=> CREATE EXTENSION pageinspect;
=> SELECT lower, upper, special, pagesize
FROM page_header(get_raw_page('accounts',0));
 lower | upper | special | pagesize
-------+-------+---------+----------
   152 |  6904 |    8192 |     8192
(1 row)
```



Special Space

The *special space* is located in the opposite part of the page, taking its highest addresses. It is used by some indexes to store auxiliary information; in other indexes and table pages this space is zero-sized.

In general, the layout of index pages is quite diverse; their content largely depends on a particular index type. Even one and the same index can have different kinds of pages: for example, B-trees have a metadata page of a special structure (page zero) and regular pages that are very similar to table pages.

Tuples

Rows contain the actual data stored in the database, together with some additional information. They are located just before the special space.

In the case of tables, we have to deal with *row versions* rather than rows because multiversion concurrency control implies having several versions of one and the same row. Indexes do not use this MVCC mechanism; instead, they have to reference all the available row versions, falling back on visibility rules to select the appropriate ones.

Both table row versions and index entries are often referred to as *tuples*. This term is borrowed from the relational theory—it is yet another legacy of PostgreSQL's academic past.

Item Pointers

The *array of pointers* to tuples serves as the page's table of contents. It is located right after the header.

Index entries have to refer to particular heap tuples somehow. PostgreSQL employs six-byte *tuple identifiers* (TIDs) for this purpose. Each TID consists of the page number of the main fork and a reference to a particular row version located in this page.

p. 27

In theory, tuples could be referred to by their offset from the start of the page. But then it would be impossible to move tuples within pages without breaking these references, which in turn would lead to page fragmentation and other unpleasant consequences.

For this reason, PostgreSQL uses indirect addressing: a tuple identifier refers to the corresponding pointer number, and this pointer specifies the current offset of the tuple. If the tuple is moved within the page, its TID still remains the same; it is enough to modify the pointer, which is also located in this page.

Each pointer takes exactly four bytes and contains the following data:

- tuple offset from the start of the page
- tuple length
- several bits defining the tuple status

Free Space

Pages can have some *free space* left between pointers and tuples (which is reflected in the free space map). There is no page fragmentation: all the free space available is always aggregated into one chunk.[1] *p. 28*

3.2 Row Version Layout

Each row version contains a header followed by actual data. The header consists of multiple fields, including the following:

xmin, xmax represent transaction IDs; they are used to differentiate between this and other versions of one and the same row.

infomask provides a set of information bits that define version properties.

ctid is a pointer to the next updated version of the same row.

null bitmap is an array of bits marking the columns that can contain NULL values.

As a result, the header turns out quite big: it requires at least 23 bytes for each tuple, and this value is often exceeded because of the null bitmap and the mandatory padding used for data alignment. In a "narrow" table, the size of various metadata can easily beat the size of the actual data stored.

Data layout on disk fully coincides with data representation in RAM. The page along with its tuples is read into the buffer cache as is, without any transformations. That's why data files are incompatible between different platforms.[2]

One of the sources of incompatibility is the byte order. For example, the x86 architecture is little-endian, z/Architecture is big-endian, and ARM has configurable byte order.

Another reason is data alignment by machine word boundaries, which is required by many architectures. For example, in a 32-bit x86 system, integer numbers (the integer type, takes four bytes) are aligned by the boundary of four-byte words,

[1] backend/storage/page/bufpage.c, PageRepairFragmentation function
[2] include/access/htup_details.h

just like double-precision floating-point numbers (the double precision type, eight bytes). But in a 64-bit system, double values are aligned by the boundary of eight-byte words.

Data alignment makes the size of a tuple dependent on the order of fields in the table. This effect is usually negligible, but in some cases it can lead to a significant size increase. Here is an example:

```
=> CREATE TABLE padding(
  b1 boolean,
  i1 integer,
  b2 boolean,
  i2 integer
);
=> INSERT INTO padding VALUES (true,1,false,2);
=> SELECT lp_len FROM heap_page_items(get_raw_page('padding', 0));
 lp_len
--------
     40
(1 row)
```

I have used the heap_page_items function of the pageinspect extension to display some details about pointers and tuples.

> In PostgreSQL, tables are often referred to as *heap*. This is yet another obscure term that hints at the similarity between space allocation for tuples and dynamic memory allocation. Some analogy can certainly be seen, but tables are managed by completely different algorithms. We can interpret this term in the sense that "everything is piled up into a heap," by contrast with ordered indexes.

The size of the row is 40 bytes. Its header takes 24 bytes, a column of the integer type takes 4 bytes, and boolean columns take 1 byte each. It makes 34 bytes, and 6 bytes are wasted on four-byte alignment of integer columns.

If we rebuild the table, the space will be used more efficiently:

```
=> DROP TABLE padding;
=> CREATE TABLE padding(
  i1 integer,
  i2 integer,
  b1 boolean,
  b2 boolean
);
```

```
=> INSERT INTO padding VALUES (1,2,true,false);
=> SELECT lp_len FROM heap_page_items(get_raw_page('padding', 0));
 lp_len
--------
     34
(1 row)
```

Another possible micro-optimization is to start the table with the fixed-length columns that cannot contain NULL values. Access to such columns will be more efficient because it is possible to cache their offset within the tuple.[1]

3.3 Operations on Tuples

To identify different versions of one and the same row, PostgreSQL marks each of them with two values: xmin and xmax. These values define "validity time" of each row version, but instead of the actual time, they rely on ever-increasing transaction IDs. p. 143

When a row is created, its xmin value is set to the transaction ID of the INSERT command.

When a row is deleted, the xmax value of its current version is set to the transaction ID of the DELETE command.

With a certain degree of abstraction, the UPDATE command can be regarded as two separate operations: DELETE and INSERT. First, the xmax value of the current row version is set to the transaction ID of the UPDATE command. Then a new version of this row is created; its xmin value will be the same as the xmax value of the previous version.

Now let's get down to some low-level details of different operations on tuples.[2]

For these experiments, we will need a two-column table with an index created on one of the columns:

[1] backend/access/common/heaptuple.c, heap_deform_tuple function
[2] backend/access/transam/README

```
=> CREATE TABLE t(
  id integer GENERATED ALWAYS AS IDENTITY,
  s text
);
=> CREATE INDEX ON t(s);
```

Insert

Start a transaction and insert one row:

```
=> BEGIN;
=> INSERT INTO t(s) VALUES ('FOO');
```

Here is the current transaction ID:

```
=> --  txid_current() before v.13
SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                 776
(1 row)
```

> To denote the concept of a transaction, PostgreSQL uses the term xact, which can be found both in SQL function names and in the source code. Consequently, a transaction ID can be called xact ID, TXID, or simply XID. We are going to come across these abbreviations over and over again.

Let's take a look at the page contents. The heap_page_items function can give us all the required information, but it shows the data "as is," so the output format is a bit hard to comprehend:

```
=> SELECT *
FROM heap_page_items(get_raw_page('t',0)) \gx
-[ RECORD 1 ]-------------------
lp          | 1
lp_off      | 8160
lp_flags    | 1
lp_len      | 32
t_xmin      | 776
t_xmax      | 0
t_field3    | 0
t_ctid      | (0,1)
```

```
t_infomask2 | 2
t_infomask  | 2050
t_hoff      | 24
t_bits      |
t_oid       |
t_data      | \x0100000009464f4f
```

To make it more readable, we can leave out some information and expand a few columns:

```
=> SELECT '(0,'||lp||')' AS ctid,
     CASE lp_flags
       WHEN 0 THEN 'unused'
       WHEN 1 THEN 'normal'
       WHEN 2 THEN 'redirect to '||lp_off
       WHEN 3 THEN 'dead'
     END AS state,
     t_xmin as xmin,
     t_xmax as xmax,
     (t_infomask & 256) > 0  AS xmin_committed,
     (t_infomask & 512) > 0  AS xmin_aborted,
     (t_infomask & 1024) > 0 AS xmax_committed,
     (t_infomask & 2048) > 0 AS xmax_aborted
FROM heap_page_items(get_raw_page('t',0)) \gx

-[ RECORD 1 ]--+-------
ctid           | (0,1)
state          | normal
xmin           | 776
xmax           | 0
xmin_committed | f
xmin_aborted   | f
xmax_committed | f
xmax_aborted   | t
```

This is what has been done here:

- The lp pointer is converted to the standard format of a tuple ID: (page number, pointer number).
- The lp_flags state is spelled out. Here it is set to the normal value, which means that it really points to a tuple.
- Of all the information bits, we have singled out just two pairs so far. The xmin_committed and xmin_aborted bits show whether the xmin transaction is

committed or aborted. The xmax_committed and xmax_aborted bits give similar information about the xmax transaction.

v. 13 The pageinspect extension provides the heap_tuple_infomask_flags function that explains all the information bits, but I am going to retrieve only those that are required at the moment, showing them in a more concise form.

Let's get back to our experiment. The INSERT command has added pointer 1 to the heap page; it refers to the first tuple, which is currently the only one.

The xmin field of the tuple is set to the current transaction ID. This transaction is still active, so the xmin_committed and xmin_aborted bits are not set yet.

The xmax field contains 0, which is a dummy number showing that this tuple has not been deleted and represents the current version of the row. Transactions will ignore this number because the xmax_aborted bit is set.

It may seem strange that the bit corresponding to an aborted transaction is set for the transaction that has not happened yet. But there is no difference between such transactions from the isolation standpoint: an aborted transaction leaves no trace, hence it has never existed.

We will use this query more than once, so I am going to wrap it into a function. And while being at it, I will also make the output more concise by hiding the information bit columns and displaying the status of transactions together with their IDs.

```
=> CREATE FUNCTION heap_page(relname text, pageno integer)
RETURNS TABLE(ctid tid, state text, xmin text, xmax text)
AS $$
SELECT (pageno,lp)::text::tid AS ctid,
     CASE lp_flags
       WHEN 0 THEN 'unused'
       WHEN 1 THEN 'normal'
       WHEN 2 THEN 'redirect to '||lp_off
       WHEN 3 THEN 'dead'
     END AS state,
     t_xmin || CASE
       WHEN (t_infomask & 256) > 0 THEN ' c'
       WHEN (t_infomask & 512) > 0 THEN ' a'
       ELSE ''
     END AS xmin,
     t_xmax || CASE
```

```
        WHEN (t_infomask & 1024) > 0 THEN ' c'
        WHEN (t_infomask & 2048) > 0 THEN ' a'
        ELSE ''
      END AS xmax
FROM heap_page_items(get_raw_page(relname,pageno))
ORDER BY lp;
$$ LANGUAGE sql;
```

Now it is much clearer what is happening in the tuple header:

```
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin | xmax
-------+--------+------+------
 (0,1) | normal | 776  | 0 a
(1 row)
```

You can get similar but less detailed information from the table itself by querying the xmin and xmax pseudocolumns:

```
=> SELECT xmin, xmax, * FROM t;
 xmin | xmax | id |  s
------+------+----+-----
  776 |    0 |  1 | FOO
(1 row)
```

Commit

Once a transaction has been completed successfully, its status has to be stored somehow—it must be registered that the transaction is *committed*. For this purpose, PostgreSQL employs a special CLOG (commit log) structure.[1] It is stored as files in the PGDATA/pg_xact directory rather than as a system catalog table.

> Previously, these files were located in PGDATA/pg_clog, but in version 10 this directory got renamed:[2] it was not uncommon for database administrators unfamiliar with PostgreSQL to delete it in search of free disk space, thinking that a "log" is something unnecessary.

[1] include/access/clog.h
backend/access/transam/clog.c

[2] commitfest.postgresql.org/13/750

p. 153 CLOG is split into several files solely for convenience. These files are accessed page by page via buffers in the server's shared memory.[1]

Just like a tuple header, CLOG contains two bits for each transaction: committed and aborted.

Once committed, a transaction is marked in CLOG with the committed bit. When any other transaction accesses a heap page, it has to answer the question: has the xmin transaction already finished?

- If not, then the created tuple must not be visible.

  To check whether the transaction is still active, PostgreSQL uses yet another structure located in the shared memory of the instance; it is called ProcArray. This structure contains the list of all the active processes, with the corresponding current (active) transaction specified for each process.

- If yes, was it committed or aborted? In the latter case, the corresponding tuple cannot be visible either.

  It is this check that requires CLOG. But even though the most recent CLOG pages are stored in memory buffers, it is still expensive to perform this check every time. Once determined, the transaction status is written into the tuple header—more specifically, into xmin_committed and xmin_aborted information bits, which are also called *hint bits*. If one of these bits is set, then the xmin transaction status is considered to be already known, and the next transaction will have to access neither CLOG nor ProcArray.

Why aren't these bits set by the transaction that performs row insertion? The problem is that it is not known yet at that time whether this transaction will complete successfully. And when it is committed, it is already unclear which tuples and pages have been changed. If a transaction affects many pages, it may be too expensive to track them. Besides, some of these pages may be not in the cache anymore; reading them again to simply update the hint bits would seriously slow down the commit.

[1] backend/access/transam/clog.c

The flip side of this cost reduction is that any transaction (even a read-only SELECT command) can start setting hint bits, thus leaving a trail of dirtied pages in the buffer cache.

Finally, let's commit the transaction started with the INSERT statement:

```
=> COMMIT;
```

Nothing has changed in the page (but we know that the transaction status has already been written into CLOG):

```
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin | xmax
-------+--------+------+------
 (0,1) | normal | 776  | 0 a
(1 row)
```

Now the first transaction that accesses the page (in a "standard" way, without using pageinspect) has to determine the status of the xmin transaction and update the hint bits:

```
=> SELECT * FROM t;
 id |  s
----+-----
  1 | FOO
(1 row)
```

```
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | normal | 776 c | 0 a
(1 row)
```

Delete

When a row is deleted, the xmax field of its current version is set to the transaction ID that performs the deletion, and the xmax_aborted bit is unset.

p. 239

While this transaction is active, the xmax value serves as a row lock. If another transaction is going to update or delete this row, it will have to wait until the xmax transaction is complete.

Let's delete a row:

```
=> BEGIN;
=> DELETE FROM t;
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                777
(1 row)
```

The transaction ID has already been written into the xmax field, but the information bits have not been set yet:

```
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | normal | 776 c | 777
(1 row)
```

Abort

The mechanism of aborting a transaction is similar to that of commit and happens just as fast, but instead of committed it sets the aborted bit in CLOG. Although the corresponding command is called ROLLBACK, no actual data rollback is happening: all the changes made by the aborted transaction in data pages remain in place.

```
=> ROLLBACK;
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | normal | 776 c | 777
(1 row)
```

When the page is accessed, the transaction status is checked, and the tuple receives the xmax_aborted hint bit. The xmax number itself still remains in the page, but no one is going to pay attention to it anymore:

```
=> SELECT * FROM t;
 id |  s
----+-----
  1 | FOO
(1 row)

=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+-------
 (0,1) | normal | 776 c | 777 a
(1 row)
```

Update

An update is performed in such a way as if the current tuple is deleted, and then a new one is inserted:

```
=> BEGIN;

=> UPDATE t SET s = 'BAR';

=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                 778
(1 row)
```

The query returns a single row (its new version):

```
=> SELECT * FROM t;
 id |  s
----+-----
  1 | BAR
(1 row)
```

But the page keeps both versions:

```
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | normal | 776 c | 778
 (0,2) | normal | 778   | 0 a
(2 rows)
```

The xmax field of the previously deleted version contains the current transaction ID. This value is written on top of the old one because the previous transaction was aborted. The xmax_aborted bit is unset since the status of the current transaction is still unknown.

To complete this experiment, let's commit the transaction.

```
=> COMMIT;
```

3.4 Indexes

Regardless of their type, indexes do not use row versioning; each row is represented by exactly one tuple. In other words, index row headers do not contain xmin and xmax fields. Index entries point to all the versions of the corresponding table row. To figure out which row version is visible, transactions have to access the table (unless the required page appears in the visibility map).

p. 110

For convenience, let's create a simple function that will use pageinspect to display all the index entries in the page (B-tree index pages store them as a flat list):

```
=> CREATE FUNCTION index_page(relname text, pageno integer)
RETURNS TABLE(itemoffset smallint, htid tid)
AS $$
SELECT itemoffset,
       htid -- ctid before v.13
FROM bt_page_items(relname,pageno);
$$ LANGUAGE sql;
```

The page references both heap tuples, the current and the previous one:

```
=> SELECT * FROM index_page('t_s_idx',1);
 itemoffset | htid
------------+-------
          1 | (0,2)
          2 | (0,1)
(2 rows)
```

Since BAR < FOO, the pointer to the second tuple comes first in the index.

3.5 TOAST

A TOAST table is virtually a regular table, and it has its own versioning that does not depend on row versions of the main table. However, rows of TOAST tables are handled in such a way that they are never updated; they can be either added or deleted, so their versioning is somewhat artificial. p. 30

Each data modification results in creation of a new tuple in the main table. But if an update does not affect any long values stored in TOAST, the new tuple will reference an existing toasted value. Only when a long value gets updated will PostgreSQL create both a new tuple in the main table and new "toasts."

3.6 Virtual Transactions

To consume transaction IDs sparingly, PostgreSQL offers a special optimization.

If a transaction is read-only, it does not affect row visibility in any way. That's why such a transaction is given a *virtual XID*[1] at first, which consists of the backend process ID and a sequential number. Assigning a virtual XID does not require any synchronization between different processes, so it happens very fast. At this point, the transaction has no real ID yet: p. 231

```
=> BEGIN;
```

[1] backend/access/transam/xact.c

```
=> --  txid_current_if_assigned() before v.13
SELECT pg_current_xact_id_if_assigned();
 pg_current_xact_id_if_assigned
--------------------------------

(1 row)
```

At different points in time, the system can contain some virtual XIDs that have already been used. And it is perfectly normal: virtual XIDs exist only in RAM, and only while the corresponding transactions are active; they are never written into data pages and never get to disk.

Once the transaction starts modifying data, it receives an actual unique ID:

```
=> UPDATE accounts
SET amount = amount - 1.00;

=> SELECT pg_current_xact_id_if_assigned();
 pg_current_xact_id_if_assigned
--------------------------------
                            780
(1 row)

=> COMMIT;
```

3.7 Subtransactions

Savepoints

SQL supports *savepoints*, which enable canceling some of the operations within a transaction without aborting this transaction as a whole. But such a scenario does not fit the course of action described above: the status of a transaction applies to all its operations, and no physical data rollback is performed.

To implement this functionality, a transaction containing a savepoint is split into several *subtransactions*,[1] so their status can be managed separately.

[1] backend/access/transam/subtrans.c

Subtransactions have their own IDs (which are bigger than the ID of the main transaction). The status of a subtransaction is written into CLOG in the usual manner; however, committed subtransactions receive both the committed and the aborted bits at once. The final decision depends on the status of the main transaction: if it is aborted, all its subtransactions will be considered aborted too.

The information about subtransactions is stored under the PGDATA/pg_subtrans directory. File access is arranged via buffers that are located in the instance's shared memory and have the same structure as CLOG buffers.[1]

> Do not confuse subtransactions with autonomous ones. Unlike subtransactions, the latter do not depend on each other in any way. Vanilla PostgreSQL does not support autonomous transactions, and it is probably for the best: they are required in very rare cases, but their availability in other database systems often provokes misuse, which can cause a lot of trouble.

Let's truncate the table, start a new transaction, and insert a row:

```
=> TRUNCATE TABLE t;
=> BEGIN;
=> INSERT INTO t(s) VALUES ('FOO');
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                782
(1 row)
```

Now create a savepoint and insert another row:

```
=> SAVEPOINT sp;
=> INSERT INTO t(s) VALUES ('XYZ');
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                782
(1 row)
```

Note that the pg_current_xact_id function returns the ID of the main transaction, not that of a subtransaction.

[1] backend/access/transam/slru.c

```
=> SELECT *
FROM heap_page('t',0) p
  LEFT JOIN t ON p.ctid = t.ctid;
 ctid  | state  | xmin | xmax | id |  s
-------+--------+------+------+----+-----
 (0,1) | normal | 782  | 0 a  |  2 | FOO
 (0,2) | normal | 783  | 0 a  |  3 | XYZ
(2 rows)
```

Let's roll back to the savepoint and insert the third row:

```
=> ROLLBACK TO sp;

=> INSERT INTO t(s) VALUES ('BAR');
=> SELECT *
FROM heap_page('t',0) p
  LEFT JOIN t ON p.ctid = t.ctid;
 ctid  | state  | xmin | xmax | id |  s
-------+--------+------+------+----+-----
 (0,1) | normal | 782  | 0 a  |  2 | FOO
 (0,2) | normal | 783  | 0 a  |    |
 (0,3) | normal | 784  | 0 a  |  4 | BAR
(3 rows)
```

The page still contains the row added by the aborted subtransaction.

Commit the changes:

```
=> COMMIT;

=> SELECT * FROM t;
 id |  s
----+-----
  2 | FOO
  4 | BAR
(2 rows)
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | normal | 782 c | 0 a
 (0,2) | normal | 783 a | 0 a
 (0,3) | normal | 784 c | 0 a
(3 rows)
```

Now we can clearly see that each subtransaction has its own status.

SQL does not allow using subtransactions directly, that is, you cannot start a new transaction before completing the current one:

```
=> BEGIN;
BEGIN
=> BEGIN;
WARNING:  there is already a transaction in progress
BEGIN
=> COMMIT;
COMMIT
=> COMMIT;
WARNING:  there is no transaction in progress
COMMIT
```

Subtransactions are employed implicitly: to implement savepoints, handle exceptions in PL/pgSQL, and in some other, more exotic cases.

Errors and Atomicity

What happens if an error occurs during execution of a statement?

```
=> BEGIN;
=> SELECT * FROM t;
 id |  s
----+-----
  2 | FOO
  4 | BAR
(2 rows)
=> UPDATE t SET s = repeat('X', 1/(id-4));
ERROR:  division by zero
```

After a failure, the whole transaction is considered aborted and cannot perform any further operations:

```
=> SELECT * FROM t;
ERROR:  current transaction is aborted, commands ignored until end
of transaction block
```

And even if you try to commit the changes, PostgreSQL will report that the transaction is rolled back:

```
=> COMMIT;
ROLLBACK
```

Why is it forbidden to continue transaction execution after a failure? Since the already executed operations are never rolled back, we would get access to some changes made before the error—it would break the atomicity of the statement, and hence that of the transaction itself.

For example, in our experiment the operator has managed to update one of the two rows before the failure:

```
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | normal | 782 c | 785
 (0,2) | normal | 783 a | 0 a
 (0,3) | normal | 784 c | 0 a
 (0,4) | normal | 785   | 0 a
(4 rows)
```

On a side note, psql provides a special mode that allows you to continue a transaction after a failure as if the erroneous statement were rolled back:

```
=> \set ON_ERROR_ROLLBACK on

=> BEGIN;

=> UPDATE t SET s = repeat('X', 1/(id-4));
ERROR:  division by zero

=> SELECT * FROM t;
 id |  s
----+-----
  2 | FOO
  4 | BAR
(2 rows)

=> COMMIT;
COMMIT
```

As you can guess, psql simply adds an implicit savepoint before each command when run in this mode; in case of a failure, a rollback is initiated. This mode is not used by default because issuing savepoints (even if they are not rolled back to) incurs significant overhead.

4

Snapshots

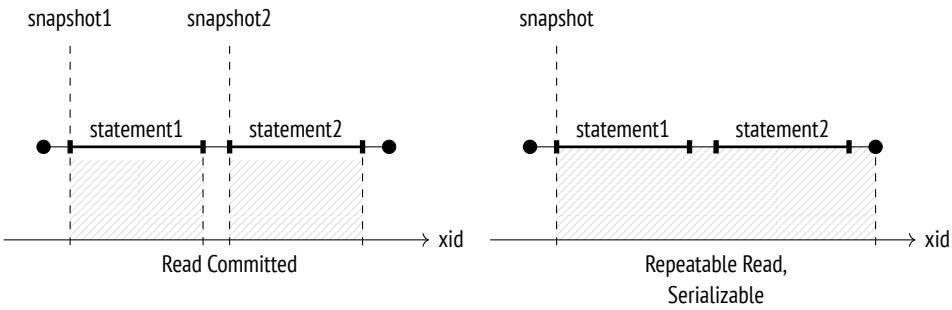
4.1 What is a Snapshot?

A data page can contain several versions of one and the same row, although each transaction must see only one of them at the most. Together, visible versions of
p. 49 all the different rows constitute a *snapshot*. A snapshot includes only the current data committed by the time it was taken, thus providing a consistent (in the ACID sense) view of the data for this particular moment.

To ensure isolation, each transaction uses its own snapshot. It means that different transactions can see different snapshots taken at different points in time, which are nevertheless consistent.

At the Read Committed isolation level, a snapshot is taken at the beginning of each statement, and it remains active only for the duration of this statement.

At the Repeatable Read and Serializable levels, a snapshot is taken at the beginning of the first statement of a transaction, and it remains active until the whole transaction is complete.



4.2 Row Version Visibility

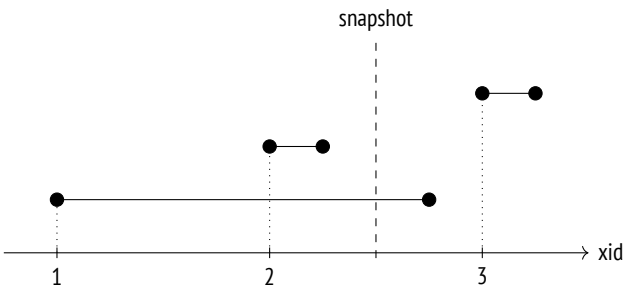
A snapshot is not a physical copy of all the required tuples. Instead, it is defined by several numbers, while tuple visibility is determined by certain rules.

Tuple visibility is defined by xmin and xmax fields of the tuple header (that is, IDs of transactions that perform insertion and deletion) and the corresponding hint bits. Since xmin–xmax intervals do not intersect, each row is represented in any snapshot by only one of its versions.

The exact visibility rules are quite complex,[1] as they take into account a variety of different scenarios and corner cases. Very roughly, we can describe them as follows: a tuple is visible in a snapshot that includes xmin transaction changes but excludes xmax transaction changes (in other words, the tuple has already appeared and has not been deleted yet).

In their turn, transaction changes are visible in a snapshot if this transaction was committed before the snapshot creation. As an exception, transactions can see their own uncommitted changes. If a transaction is aborted, its changes will not be visible in any snapshot.

Let's take a look at a simple example. In this illustration line segments represent transactions (from their start time till commit time):



Here visibility rules are applied to transactions as follows:

- Transaction 2 was committed before the snapshot creation, so its changes are visible.

[1] backend/access/heap/heapam_visibility.c

- Transaction 1 was active at the time of the snapshot creation, so its changes are not visible.
- Transaction 3 was started after the snapshot creation, so its changes are not visible either (it makes no difference whether this transaction is completed or not).

4.3 Snapshot Structure

Unfortunately, the previous illustration has nothing to do with the way PostgreSQL actually sees this picture.[1] The problem is that the system does not know when transactions got committed. It is only known when they were started (this moment is defined by the transaction ID), while their completion is not registered anywhere.

off

Commit times can be tracked[2] if you enable the *track_commit_timestamp* parameter, but they do not participate in visibility checks in any way (although it can still be useful to track them for other purposes, for example, to apply in external replication solutions).

Besides, PostgreSQL always logs commit and rollback times in the corresponding WAL entries, but this information is used only for point-in-time recovery.

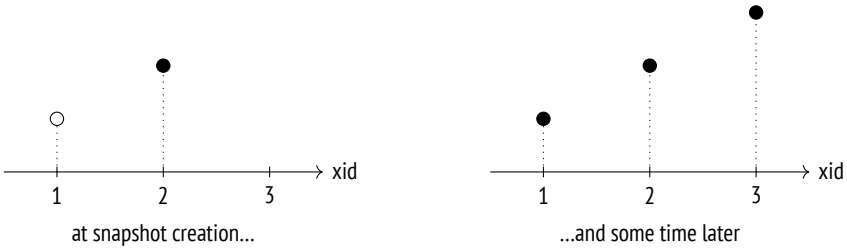
p. 189

It is only the *current* status of a transaction that we can learn. This information is available in the server's shared memory: the ProcArray structure contains the list of all the active sessions and their transactions. Once a transaction is complete, it is impossible to find out whether it was active at the time of the snapshot creation.

So to create a snapshot, it is not enough to register the moment when it was taken: it is also necessary to collect the status of all the transactions at that moment. Otherwise, later it will be impossible to understand which tuples must be visible in the snapshot, and which must be excluded.

Take a look at the information available to the system when the snapshot was taken and some time afterwards (the white circle denotes an active transaction, whereas the black circles stand for completed ones):

[1] include/utils/snapshot.h
backend/utils/time/snapmgr.c
[2] backend/access/transam/commit_ts.c



Suppose we did not know that at the time the snapshot was taken the first transaction was still being executed and the third transaction had not started yet. Then it would seem that they were just like the second transaction (which was committed at that time), and it would be impossible to filter them out.

For this reason, PostgreSQL cannot create a snapshot that shows a consistent state of data at some arbitrary point in the past, even if all the required tuples are present in heap pages. Consequently, it is impossible to implement retrospective queries (which are sometimes also called temporal or flashback queries).

> Intriguingly, such functionality was declared as one of the objectives of Postgres and was implemented at the very start, but it was removed from the database system when the project support was passed on to the community.[1]

Thus, a snapshot consists of several values saved at the time of its creation:[2]

xmin is the snapshot's lower boundary, which is represented by the ID of the oldest active transaction.

All the transactions with smaller IDs are either committed (so their changes are included into the snapshot) or aborted (so their changes are ignored). *p. 143*

xmax is the snapshot's upper boundary, which is represented by the value that exceeds the ID of the latest committed transaction by one. The upper boundary defines the moment when the snapshot was taken.

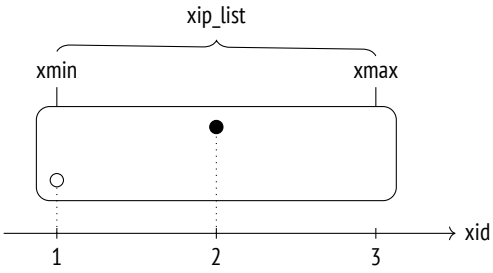
All the transactions whose IDs are equal to or greater than xmax are either still running or do not exist, so their changes cannot be visible.

xip_list is the list of IDs of all the active transactions except for virtual ones, which do not affect visibility in any way. *p. 85*

[1] *Joseph M. Hellerstein*, Looking Back at Postgres. https://arxiv.org/pdf/1901.01973.pdf
[2] backend/storage/ipc/procarray.c, GetSnapshotData function

Snapshots also include several other parameters, but we will ignore them for now.

In a graphical form, a snapshot can be represented as a rectangle that comprises transactions from xmin to xmax:



To understand how visibility rules are defined by the snapshot, we are going to reproduce the above scenario on the accounts table.

```
=> TRUNCATE TABLE accounts;
```

The first transaction inserts the first row into the table and remains open:

```
=> BEGIN;
=> INSERT INTO accounts VALUES (1, 'alice', 1000.00);
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                 790
(1 row)
```

The second transaction inserts the second row and commits this change immediately:

```
=> BEGIN;
=> INSERT INTO accounts VALUES (2, 'bob', 100.00);
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                 791
(1 row)
=> COMMIT;
```

At this point, let's create a new snapshot in another session. We could simply run any query for this purpose, but we will use a special function to take a look at this snapshot right away:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> --  txid_current_snapshot() before v.13
SELECT pg_current_snapshot();
 pg_current_snapshot
---------------------
 790:792:790
(1 row)
```

This function displays the following snapshot components, separated by colons: xmin, xmax, and xip_list (the list of active transactions; in this particular case it consists of a single item).

Once the snapshot is taken, commit the first transaction:

```
=> COMMIT;
```

The third transaction is started after the snapshot creation. It modifies the second row, so a new tuple appears:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100 WHERE id = 2;
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                792
(1 row)
=> COMMIT;
```

Our snapshot sees only one tuple:

```
=> SELECT ctid, * FROM accounts;
 ctid  | id | client | amount
-------+----+--------+--------
 (0,2) |  2 | bob    | 100.00
(1 row)
```

But the table contains three of them:

```
=> SELECT * FROM heap_page('accounts',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+-------
 (0,1) | normal | 790 c | 0 a
 (0,2) | normal | 791 c | 792 c
 (0,3) | normal | 792 c | 0 a
(3 rows)
```

So how does PostgreSQL choose which versions to show? By the above rules, changes are included into a snapshot only if they are made by committed transactions that satisfy the following criteria:

- If xid < xmin, changes are shown unconditionally (like in the case of the transaction that created the accounts table).
- If xmin ⩽ xid < xmax, changes are shown only if the corresponding transaction IDs are not in xip_list.

The first row (0,1) is invisible because it is inserted by a transaction that appears in xip_list (even though this transaction falls into the snapshot range).

The latest version of the second row (0,3) is invisible because the corresponding transaction ID is above the upper boundary of the snapshot.

But the first version of the second row (0,2) is visible: row insertion was performed by a transaction that falls into the snapshot range and does not appear in xip_list (the insertion is visible), while row deletion was performed by a transaction whose ID is above the upper boundary of the snapshot (the deletion is invisible).

```
=> COMMIT;
```

4.4 Visibility of Transactions' Own Changes

Things get a bit more complicated when it comes to defining visibility rules for transactions' own changes: in some cases, only part of such changes must be visible. For example, a cursor that was opened at a particular point in time must not see any changes that happened later, regardless of the isolation level.

To address such situations, tuple headers provide a special field (displayed as cmin and cmax pseudocolumns) that shows the sequence number of the operation within the transaction. The cmin column identifies insertion, while cmax is used for deletion operations. To save space, these values are stored in a single field of the tuple header rather than in two different ones. It is assumed that one and the same row almost never gets both inserted and deleted within a single transaction. (If it does happen, PostgreSQL writes a special combo identifier into this field, and the actual cmin and cmax values are stored by the backend in this case.[1])

As an illustration, let's start a transaction and insert a row into the table:

```
=> BEGIN;
=> INSERT INTO accounts VALUES (3, 'charlie', 100.00);
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                793
(1 row)
```

Open a cursor to run the query that returns the number of rows in this table:

```
=> DECLARE c CURSOR FOR SELECT count(*) FROM accounts;
```

Insert one more row:

```
=> INSERT INTO accounts VALUES (4, 'charlie', 200.00);
```

Now extend the output by another column to display the cmin value for the rows inserted by our transaction (it makes no sense for other rows):

```
=> SELECT xmin, CASE WHEN xmin = 793 THEN cmin END cmin, *
FROM accounts;
 xmin | cmin | id | client  | amount
------+------+----+---------+---------
  790 |      |  1 | alice   | 1000.00
  792 |      |  2 | bob     |  200.00
  793 |    0 |  3 | charlie |  100.00
  793 |    1 |  4 | charlie |  200.00
(4 rows)
```

[1] backend/utils/time/combocid.c

The cursor query gets only three rows; the row inserted when the cursor was already open does not make it into the snapshot because the cmin < 1 condition is not satisfied:

```
=> FETCH c;
 count
-------
     3
(1 row)
```

Naturally, this cmin number is also stored in the snapshot, but it is impossible to display it using any SQL means.

4.5 Transaction Horizon

As mentioned earlier, the lower boundary of the snapshot is represented by xmin, which is the ID of the oldest transaction that was active at the moment of the snapshot creation. This value is very important because it defines the *horizon* of the transaction that uses this snapshot.

If a transaction has no active snapshot (for example, at the Read Committed isolation level between statement execution), its horizon is defined by its own ID if it is assigned.

All the transactions that are beyond the horizon (those with xid < xmin) are guaranteed to be committed. It means that a transaction can see only the current row versions beyond its horizon.

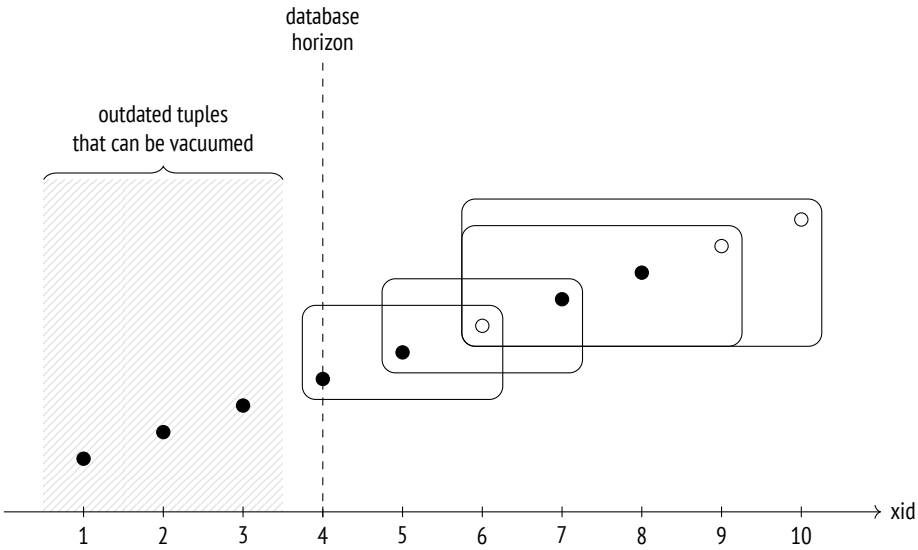
As you can guess, this term is inspired by the concept of *event horizon* in physics.

PostgreSQL tracks the current horizons of all its processes; transactions can see their own horizons in the pg_stat_activity table:

```
=> BEGIN;
=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
 backend_xmin
--------------
          793
(1 row)
```

Virtual transactions have no real IDs, but they still use snapshots just like regular transactions, so they have their own horizons. The only exception is virtual transactions without an active snapshot: the concept of the horizon makes no sense for them, and they are fully "transparent" to the system when it comes to snapshots and visibility (even though pg_stat_activity.backend_xmin may still contain an xmin of an old snapshot).

We can also define the *database horizon* in a similar manner. For this purpose, we should take the horizons of all the transactions in this database and select the most remote one, which has the oldest xmin.[1] Beyond this horizon, outdated heap tuples will never be visible to any transaction in this database. *Such tuples can be safely cleaned up by vacuum*—this is exactly why the concept of the horizon is so important from a practical standpoint.



Let's draw some conclusions:

- If a transaction (no matter whether it is real or virtual) at the Repeatable Read or Serializable isolation level is running for a long time, it thereby holds the database horizon and defers vacuuming.

[1] backend/storage/ipc/procarray.c, ComputeXidHorizons function

- A real transaction at the Read Committed isolation level holds the database horizon in the same way, even if it is not executing any operators (being in the "idle in transaction" state).
- A virtual transaction at the Read Committed isolation level holds the horizon only while executing operators.

There is only one horizon for the whole database, so if it is being held by a transaction, it is impossible to vacuum any data within this horizon—even if this data has not been accessed by this transaction.

> Cluster-wide tables of the system catalog have a separate horizon that takes into account all transactions in all databases. Temporary tables, on the contrary, do not have to pay attention to any transactions except those that are being executed by the current process.

Let's get back to our current experiment. The active transaction of the first session still holds the database horizon; we can see it by incrementing the transaction counter:

```
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                 794
(1 row)
```

```
=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
 backend_xmin
--------------
          793
(1 row)
```

And only when this transaction is complete, the horizon moves forward, and outdated tuples can be vacuumed:

```
=> COMMIT;
=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
 backend_xmin
--------------
          795
(1 row)
```

In a perfect world, you should avoid combining long transactions with frequent updates (that spawn new row versions), as it will lead to table and index bloating. p. 163

4.6 System Catalog Snapshots

Although the system catalog consists of regular tables, they cannot be accessed via a snapshot used by a transaction or an operator. The snapshot must be "fresh" enough to include all the latest changes, otherwise transactions could see outdated definitions of table columns or miss newly added integrity constraints.

Here is a simple example:

```
=> BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

=> SELECT 1; -- a snapshot for the transaction is taken

    => ALTER TABLE accounts
      ALTER amount SET NOT NULL;

=> INSERT INTO accounts(client, amount)
  VALUES ('alice', NULL);
ERROR:  null value in column "amount" of relation "accounts"
violates not-null constraint
DETAIL:  Failing row contains (1, alice, null).

=> ROLLBACK;
```

The integrity constraint that appeared after the snapshot creation was visible to the INSERT command. It may seem that such behavior breaks isolation, but if the inserting transaction had managed to reach the accounts table before the ALTER TABLE command, the latter would have been blocked until this transaction completed. p. 232

In general, the server behaves as if a separate snapshot is created for each system catalog query. But the implementation is, of course, much more complex[1] since frequent snapshot creation would negatively affect performance; besides, many system catalog objects get cached, and it must also be taken into account.

[1] backend/utils/time/snapmgr.c, GetCatalogSnapshot function

4.7 Exporting Snapshots

In some situations, concurrent transactions must see one and the same snapshot by all means. For example, if the pg_dump utility is run in the parallel mode, all its processes must see the same database state to produce a consistent backup.

We cannot assume that snapshots will be identical simply because transactions were started "simultaneously." To ensure that all the transactions see the same data, we must employ the snapshot export mechanism.

The pg_export_snapshot function returns a snapshot ID, which can be passed to another transaction (outside of the database system):

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT count(*) FROM accounts;
 count
-------
     4
(1 row)

=> SELECT pg_export_snapshot();
 pg_export_snapshot
---------------------
 00000004-0000006E-1
(1 row)
```

Before executing the first statement, the other transaction can import the snapshot by running the SET TRANSACTION SNAPSHOT command. The isolation level must be set to Repeatable Read or Serializable because operators use their own snapshots at the Read Committed level:

```
=> DELETE FROM accounts;
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SET TRANSACTION SNAPSHOT '00000004-0000006E-1';
```

Now the second transaction is going to use the snapshot of the first transaction, and consequently, it will see four rows (instead of zero):

```
=> SELECT count(*) FROM accounts;
 count
-------
     4
(1 row)
```

Clearly, the second transaction will not see any changes made by the first transaction after the snapshot export (and vice versa): regular visibility rules still apply.

The exported snapshot's lifetime is the same as that of the exporting transaction.

```
=> COMMIT;
```

```
=> COMMIT;
```

5

Page Pruning and HOT Updates

5.1 Page Pruning

While a heap page is being read or updated, PostgreSQL can perform some quick page cleanup, or *pruning*.[1] It happens in the following cases:

- The previous UPDATE operation did not find enough space to place a new tuple into the same page. This event is reflected in the page header.
- 100 The heap page contains more data than allowed by the *fillfactor* storage parameter.

  An INSERT operation can add a new row into the page only if this page is filled for less than *fillfactor* percent. The rest of the space is kept for UPDATE operations (no such space is reserved by default).

Page pruning removes the tuples that cannot be visible in any snapshot anymore
p. 101
(that is, that are beyond the database horizon). It never goes beyond a single heap page, but in return it is performed very fast. Pointers to pruned tuples remain in place since they may be referenced from an index—which is already a different page.

For the same reason, neither the visibility map nor the free space map is refreshed (so the recovered space is set aside for updates, not for insertions).

Since a page can be pruned during reads, any SELECT statement can cause page modifications. This is yet another such case in addition to deferred setting of in-
p. 80
formation bits.

[1] backend/access/heap/pruneheap.c, heap_page_prune_opt function

Let's take a look at how page pruning actually works. We are going to create a two-column table and build an index on each of the columns:

```
=> CREATE TABLE hot(id integer, s char(2000)) WITH (fillfactor = 75);
=> CREATE INDEX hot_id ON hot(id);
=> CREATE INDEX hot_s ON hot(s);
```

If the s column contains only Latin letters, each heap tuple will have a fixed size of 2004 bytes, plus 24 bytes of the header. The *fillfactor* storage parameter is set to 75 %. It means that the page has enough free space for four tuples, but we can insert only three.

Let's insert a new row and update it several times:

```
=> INSERT INTO hot VALUES (1, 'A');
=> UPDATE hot SET s = 'B';
=> UPDATE hot SET s = 'C';
=> UPDATE hot SET s = 'D';
```

Now the page contains four tuples:

```
=> SELECT * FROM heap_page('hot',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+-------
 (0,1) | normal | 801 c | 802 c
 (0,2) | normal | 802 c | 803 c
 (0,3) | normal | 803 c | 804
 (0,4) | normal | 804   | 0 a
(4 rows)
```

Expectedly, we have just exceeded the *fillfactor* threshold. You can tell it by the difference between the pagesize and upper values—it is bigger than 75 % of the page size, which is 6144 bytes: p. 70

```
=> SELECT upper, pagesize FROM page_header(get_raw_page('hot',0));
 upper | pagesize
-------+----------
    64 |     8192
(1 row)
```

The next page access triggers page pruning that removes all the outdated tuples. Then a new tuple (0,5) is added into the freed space:

```
=> UPDATE hot SET s = 'E';
=> SELECT * FROM heap_page('hot',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | dead   |       |
 (0,2) | dead   |       |
 (0,3) | dead   |       |
 (0,4) | normal | 804 c | 805
 (0,5) | normal | 805   | 0 a
(5 rows)
```

The remaining heap tuples are physically moved towards the highest addresses so that all the free space is aggregated into a single continuous chunk. The tuple pointers are also modified accordingly. As a result, there is no free space fragmentation in the page.

The pointers to the pruned tuples cannot be removed yet because they are still referenced from the indexes; PostgreSQL changes their status from normal to dead. Let's take a look at the first page of the hot_s index (the zero page is used for metadata):

```
=> SELECT * FROM index_page('hot_s',1);
 itemoffset | htid
------------+-------
          1 | (0,1)
          2 | (0,2)
          3 | (0,3)
          4 | (0,4)
          5 | (0,5)
(5 rows)
```

We can see the same picture in the other index too:

```
=> SELECT * FROM index_page('hot_id',1);
 itemoffset | htid
------------+-------
          1 | (0,1)
          2 | (0,2)
          3 | (0,3)
          4 | (0,4)
          5 | (0,5)
(5 rows)
```

An index scan can return (0,1), (0,2), and (0,3) as tuple identifiers. The server tries to read the corresponding heap tuple but sees that the pointer has the dead status; it means that this tuple does not exist anymore and should be ignored. And while being at it, the server also changes the pointer status in the index page to avoid repeated heap page access.[1]

Let's extend the function displaying index pages so that it also shows whether the pointer is dead: v. 13

```
=> DROP FUNCTION index_page(text, integer);

=> CREATE FUNCTION index_page(relname text, pageno integer)
RETURNS TABLE(itemoffset smallint, htid tid, dead boolean)
AS $$
SELECT itemoffset,
       htid,
       dead -- starting from v.13
FROM bt_page_items(relname,pageno);
$$ LANGUAGE sql;

=> SELECT * FROM index_page('hot_id',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,1) | f
          2 | (0,2) | f
          3 | (0,3) | f
          4 | (0,4) | f
          5 | (0,5) | f
(5 rows)
```

All the pointers in the index page are active so far. But as soon as the first index scan occurs, their status changes:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM hot WHERE id = 1;
                        QUERY PLAN
-------------------------------------------------------
 Index Scan using hot_id on hot (actual rows=1 loops=1)
   Index Cond: (id = 1)
(2 rows)
```

[1] backend/access/index/indexam.c, index_fetch_heap function

```
=> SELECT * FROM index_page('hot_id',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,1) | t
          2 | (0,2) | t
          3 | (0,3) | t
          4 | (0,4) | t
          5 | (0,5) | f
(5 rows)
```

Although the heap tuple referenced by the fourth pointer is still unpruned and has the normal status, it is already beyond the database horizon. That's why this pointer is also marked as dead in the index.

5.2 HOT Updates

It would be very inefficient to keep references to all heap tuples in an index.

To begin with, each row modification triggers updates of *all* the indexes created on the table: once a new heap tuple appears, each index must include a reference to this tuple, even if the modified fields are not indexed.

Furthermore, indexes accumulate references to historic heap tuples, so they have
p. 118 to be pruned together with these tuples.

Things get worse as you create more indexes on a table.

But if the updated column is not a part of *any* index, there is no point in creating another index entry that contains the same key value. To avoid such redundancies, PostgreSQL provides an optimization called *Heap-Only Tuple updates*.[1]

If such an update is performed, an index page contains only one entry for each row. This entry points to the very first row version; all the subsequent versions located in the same page are bound into a chain by ctid pointers in the tuple headers.

Row versions that are not referenced from any index are tagged with the Heap-Only Tuple bit. If a version is included into the HOT chain, it is tagged with the Heap Hot Updated bit.

[1] backend/access/heap/README.HOT

If an index scan accesses a heap page and finds a row version marked as Heap Hot Updated, it means that the scan should continue, so it goes further along the chain of HOT updates. Obviously, all the fetched row versions are checked for visibility before the result is returned to the client.

To take a look at how HOT updates are performed, let's delete one of the indexes and truncate the table.

```
=> DROP INDEX hot_s;
=> TRUNCATE TABLE hot;
```

For convenience, we will redefine the heap_page function so that its output includes three more fields: ctid and the two bits related to HOT updates:

```
=> DROP FUNCTION heap_page(text,integer);
=> CREATE FUNCTION heap_page(relname text, pageno integer)
RETURNS TABLE(
  ctid tid, state text,
  xmin text, xmax text,
  hhu text, hot text, t_ctid tid
) AS $$
SELECT (pageno,lp)::text::tid AS ctid,
       CASE lp_flags
         WHEN 0 THEN 'unused'
         WHEN 1 THEN 'normal'
         WHEN 2 THEN 'redirect to '||lp_off
         WHEN 3 THEN 'dead'
       END AS state,
       t_xmin || CASE
         WHEN (t_infomask & 256) > 0 THEN ' c'
         WHEN (t_infomask & 512) > 0 THEN ' a'
         ELSE ''
       END AS xmin,
       t_xmax || CASE
         WHEN (t_infomask & 1024) > 0 THEN ' c'
         WHEN (t_infomask & 2048) > 0 THEN ' a'
         ELSE ''
       END AS xmax,
       CASE WHEN (t_infomask2 & 16384) > 0 THEN 't' END AS hhu,
       CASE WHEN (t_infomask2 & 32768) > 0 THEN 't' END AS hot,
       t_ctid
FROM heap_page_items(get_raw_page(relname,pageno))
ORDER BY lp;
$$ LANGUAGE sql;
```

Let's repeat the insert and update operations:

```
=> INSERT INTO hot VALUES (1, 'A');
=> UPDATE hot SET s = 'B';
```

The page now contains a chain of HOT updates:

- The Heap Hot Updated bit shows that the executor should follow the CTID chain.
- The Heap Only Tuple bit indicates that this tuple is not referenced from any indexes.

```
=> SELECT * FROM heap_page('hot',0);
 ctid  | state  | xmin  | xmax | hhu | hot | t_ctid
-------+--------+-------+------+-----+-----+--------
 (0,1) | normal | 812 c | 813  | t   |     | (0,2)
 (0,2) | normal | 813   | 0 a  |     | t   | (0,2)
(2 rows)
```

As we make further updates, the chain will grow—but only within the page limits:

```
=> UPDATE hot SET s = 'C';
=> UPDATE hot SET s = 'D';
=> SELECT * FROM heap_page('hot',0);
 ctid  | state  | xmin  | xmax  | hhu | hot | t_ctid
-------+--------+-------+-------+-----+-----+--------
 (0,1) | normal | 812 c | 813 c | t   |     | (0,2)
 (0,2) | normal | 813 c | 814 c | t   | t   | (0,3)
 (0,3) | normal | 814 c | 815   | t   | t   | (0,4)
 (0,4) | normal | 815   | 0 a   |     | t   | (0,4)
(4 rows)
```

The index still contains only one reference, which points to the head of this chain:

```
=> SELECT * FROM index_page('hot_id',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,1) | f
(1 row)
```

A HOT update is possible if the modified fields are not a part of *any* index. Otherwise, some of the indexes would contain a reference to a heap tuple that appears in the middle of the chain, which contradicts the idea of this optimization. Since a HOT chain can grow only within a single page, traversing the whole chain never requires access to other pages and thus does not hamper performance.

5.3 Page Pruning for HOT Updates

A special case of page pruning—which is nevertheless important—is pruning of HOT update chains.

In the example above, the *fillfactor* threshold is already exceeded, so the next update should trigger page pruning. But this time the page contains a chain of HOT updates. The head of this chain must always remain in its place since it is referenced from the index, but other pointers can be released because they are sure to have no external references.

To avoid moving the head, PostgreSQL uses dual addressing: the pointer referenced from the index (which is (0,1) in this case) receives the redirect status since it points to the tuple that currently starts the chain:

```
=> UPDATE hot SET s = 'E';

=> SELECT * FROM heap_page('hot',0);
 ctid  |     state     | xmin  | xmax | hhu | hot | t_ctid
-------+---------------+-------+------+-----+-----+--------
 (0,1) | redirect to 4 |       |      |     |     |
 (0,2) | normal        | 816   | 0 a  |     | t   | (0,2)
 (0,3) | unused        |       |      |     |     |
 (0,4) | normal        | 815 c | 816  | t   | t   | (0,2)
(4 rows)
```

The tuples (0,1), (0,2), and (0,3) have been pruned; the head pointer 1 remains for redirection purposes, while pointers 2 and 3 have been deallocated (received the unused status) since they are guaranteed to have no references from indexes. The new tuple is written into the freed space as tuple (0,2).

Let's perform some more updates:

```
=> UPDATE hot SET s = 'F';

=> UPDATE hot SET s = 'G';

=> SELECT * FROM heap_page('hot',0);
 ctid  |     state     | xmin  | xmax  | hhu | hot | t_ctid
-------+---------------+-------+-------+-----+-----+--------
 (0,1) | redirect to 4 |       |       |     |     |
 (0,2) | normal        | 816 c | 817 c | t   | t   | (0,3)
 (0,3) | normal        | 817 c | 818   | t   | t   | (0,5)
 (0,4) | normal        | 815 c | 816 c | t   | t   | (0,2)
 (0,5) | normal        | 818   | 0 a   |     | t   | (0,5)
(5 rows)
```

The next update is going to trigger page pruning:

```
=> UPDATE hot SET s = 'H';

=> SELECT * FROM heap_page('hot',0);
 ctid  |     state     | xmin  | xmax | hhu | hot | t_ctid
-------+---------------+-------+------+-----+-----+--------
 (0,1) | redirect to 5 |       |      |     |     |
 (0,2) | normal        | 819   | 0 a  |     | t   | (0,2)
 (0,3) | unused        |       |      |     |     |
 (0,4) | unused        |       |      |     |     |
 (0,5) | normal        | 818 c | 819  | t   | t   | (0,2)
(5 rows)
```

Again, some of the tuples are pruned, and the pointer to the head of the chain is shifted accordingly.

If unindexed columns are modified frequently, it makes sense to reduce the *fillfactor* value, thus reserving some space in the page for updates. Obviously, you have to keep in mind that the lower the *fillfactor* value is, the more free space is left in the page, so the physical size of the table grows.

5.4 HOT Chain Splits

If the page has no more space to accommodate a new tuple, the chain will be cut off. PostgreSQL will have to add a separate index entry to refer to the tuple located in another page.

To observe this situation, let's start a concurrent transaction with a snapshot that blocks page pruning:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT 1;
```

Now we are going to perform some updates in the first session:

```
=> UPDATE hot SET s = 'I';

=> UPDATE hot SET s = 'J';

=> UPDATE hot SET s = 'K';

=> SELECT * FROM heap_page('hot',0);
 ctid  |     state     | xmin  | xmax  | hhu | hot | t_ctid
-------+---------------+-------+-------+-----+-----+--------
 (0,1) | redirect to 2 |       |       |     |     |
 (0,2) | normal        | 819 c | 820 c | t   | t   | (0,3)
 (0,3) | normal        | 820 c | 821 c | t   | t   | (0,4)
 (0,4) | normal        | 821 c | 822   | t   | t   | (0,5)
 (0,5) | normal        | 822   | 0 a   |     | t   | (0,5)
(5 rows)
```

When the next update happens, this page will not be able to accommodate another tuple, and page pruning will not manage to free any space:

```
=> UPDATE hot SET s = 'L';
```

```
=> COMMIT; -- the snapshot is not required anymore
```

```
=> SELECT * FROM heap_page('hot',0);
 ctid  |     state     | xmin  | xmax  | hhu | hot | t_ctid
-------+---------------+-------+-------+-----+-----+--------
 (0,1) | redirect to 2 |       |       |     |     |
 (0,2) | normal        | 819 c | 820 c | t   | t   | (0,3)
 (0,3) | normal        | 820 c | 821 c | t   | t   | (0,4)
 (0,4) | normal        | 821 c | 822 c | t   | t   | (0,5)
 (0,5) | normal        | 822 c | 823   |     | t   | (1,1)
(5 rows)
```

Tuple (0,5) contains the (1,1) reference that goes to page 1:

```
=> SELECT * FROM heap_page('hot',1);
 ctid  | state  | xmin | xmax | hhu | hot | t_ctid
-------+--------+------+------+-----+-----+--------
 (1,1) | normal | 823  | 0 a  |     |     | (1,1)
(1 row)
```

However, this reference is not used: the Heap Hot Updated bit is not set for tuple (0,5). As for tuple (1,1), it can be accessed from the index that now has two entries. Each of them points to the head of their own HOT chain:

```
=> SELECT * FROM index_page('hot_id',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,1) | f
          2 | (1,1) | f
(2 rows)
```

5.5 Page Pruning for Indexes

I have declared that page pruning is confined to a single heap page and does not affect indexes. However, indexes have their own pruning,[1] which also cleans up a single page—an index one in this case.

Index pruning happens when an insertion into a B-tree is about to split the page into two, as the original page does not have enough space anymore. The problem is that even if some index entries are deleted later, two separate index pages will not

[1] postgresql.org/docs/14/btree-implementation.html#BTREE-DELETION

be merged into one. It leads to index bloating, and once bloated, the index cannot shrink even if a large part of the data is deleted. But if pruning can remove some of the tuples, a page split may be deferred.

There are two types of tuples that can be pruned from an index.

First of all, PostgreSQL prunes those tuples that have been tagged as dead.[1] As I have already said, PostgreSQL sets such a tag during an index scan if it detects an index entry pointing to a tuple that is not visible in any snapshot anymore or simply does not exist.

If no tuples are known to be dead, PostgreSQL checks those index entries that reference different versions of one and the same table row.[2] Because of MVCC, update operations may generate a large number of row versions, and many of them are soon likely to disappear behind the database horizon. HOT updates cushion this effect, but they are not always applicable: if the column to update is a part of an index, the corresponding references are propagated to all the indexes. Before splitting the page, it makes sense to search for the rows that are not tagged as dead yet but can already be pruned. To achieve this, PostgreSQL has to check visibility of heap tuples. Such checks require table access, so they are performed only for "promising" index tuples, which have been created as copies of the existing ones for MVCC purposes. It is cheaper to perform such a check than to allow an extra page split. v. 14

[1] backend/access/nbtree/README, Simple deletion section
[2] backend/access/nbtree/README, Bottom-Up deletion section
include/access/tableam.h

6

Vacuum and Autovacuum

6.1 Vacuum

Page pruning happens very fast, but it frees only part of the space that can be potentially reclaimed. Operating within a single heap page, it does not touch upon indexes (or vice versa, it cleans up an index page without affecting the table).

Routine vacuuming,[1] which is the main vacuuming procedure, is performed by the VACUUM command.[2] It processes the whole table and eliminates both outdated heap tuples and all the corresponding index entries.

Vacuuming is performed in parallel with other processes in the database system. While being vacuumed, tables and indexes can be used in the usual manner, both for read and write operations (but concurrent execution of such commands as CREATE INDEX, ALTER TABLE, and some others is not allowed).

p. 232

p. 29

To avoid scanning extra pages, PostgreSQL uses a visibility map. Pages tracked in this map are skipped since they are sure to contain only the current tuples, so a page will only be vacuumed if it does not appear in this map. If all the tuples remaining in a page after vacuuming are beyond the database horizon, the visibility map is refreshed to include this page.

The free space map also gets updated to reflect the space that has been cleared.

Let's create a table with an index on it:

[1] postgresql.org/docs/14/routine-vacuuming.html
[2] postgresql.org/docs/14/sql-vacuum.html
backend/commands/vacuum.c

```
=> CREATE TABLE vac(
  id integer,
  s char(100)
)
WITH (autovacuum_enabled = off);

=> CREATE INDEX vac_s ON vac(s);
```

The *autovacuum_enabled* storage parameter turns off autovacuum; we are doing it here solely for the purpose of experimentation to precisely control vacuuming start time.

Let's insert a row and make a couple of updates:

```
=> INSERT INTO vac(id,s) VALUES (1,'A');

=> UPDATE vac SET s = 'B';

=> UPDATE vac SET s = 'C';
```

Now the table contains three tuples:

```
=> SELECT * FROM heap_page('vac',0);
 ctid  | state  | xmin  | xmax  | hhu | hot | t_ctid
-------+--------+-------+-------+-----+-----+--------
 (0,1) | normal | 826 c | 827 c |     |     | (0,2)
 (0,2) | normal | 827 c | 828   |     |     | (0,3)
 (0,3) | normal | 828   | 0 a   |     |     | (0,3)
(3 rows)
```

Each tuple is referenced from the index:

```
=> SELECT * FROM index_page('vac_s',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,1) | f
          2 | (0,2) | f
          3 | (0,3) | f
(3 rows)
```

Vacuuming has removed all the dead tuples, leaving only the current one:

```
=> VACUUM vac;
```

```
=> SELECT * FROM heap_page('vac',0);
 ctid  | state  | xmin  | xmax | hhu | hot | t_ctid
-------+--------+-------+------+-----+-----+--------
 (0,1) | unused |       |      |     |     |
 (0,2) | unused |       |      |     |     |
 (0,3) | normal | 828 c | 0 a  |     |     | (0,3)
(3 rows)
```

In the case of page pruning, the first two pointers would be considered dead, but here they have the unused status since no index entries are referring to them now:

```
=> SELECT * FROM index_page('vac_s',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,3) | f
(1 row)
```

Pointers with the unused status are treated as free and can be reused by new row versions.

Now the heap page appears in the visibility map; we can check it using the pg_visibility extension:

```
=> CREATE EXTENSION pg_visibility;
=> SELECT all_visible
FROM pg_visibility_map('vac',0);
 all_visible
-------------
 t
(1 row)
```

The page header has also received an attribute showing that all its tuples are visible in all snapshots:

```
=> SELECT flags & 4 > 0 AS all_visible
FROM page_header(get_raw_page('vac',0));
 all_visible
-------------
 t
(1 row)
```

6.2 Database Horizon Revisited

Vacuuming detects dead tuples based on the database horizon. This concept is so fundamental that it makes sense to get back to it once again.

Let's restart our experiment from the very beginning:

```
=> TRUNCATE vac;
=> INSERT INTO vac(id,s) VALUES (1,'A');
=> UPDATE vac SET s = 'B';
```

But this time, before updating the row, we are going to open another transaction that will hold the database horizon (it can be almost any transaction, except for a virtual one executed at the Read Committed isolation level). For example, this transaction can modify some rows in *another* table. *p. 101*

```
    => BEGIN;
    => UPDATE accounts SET amount = 0;
```

```
=> UPDATE vac SET s = 'C';
```

Now our table contains three tuples, and the index contains three references. Let's vacuum the table and see what changes:

```
=> VACUUM vac;
=> SELECT * FROM heap_page('vac',0);
 ctid  | state  | xmin  | xmax  | hhu | hot | t_ctid
-------+--------+-------+-------+-----+-----+--------
 (0,1) | unused |       |       |     |     |
 (0,2) | normal | 833 c | 835 c |     |     | (0,3)
 (0,3) | normal | 835 c | 0 a   |     |     | (0,3)
(3 rows)
=> SELECT * FROM index_page('vac_s',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,2) | f
          2 | (0,3) | f
(2 rows)
```

While the previous run left only one tuple in the page, now we have two of them: VACUUM has decided that version (0,2) cannot be removed yet. The reason is the database horizon, which is defined by an unfinished transaction in this case:

```
=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
 backend_xmin
--------------
          834
(1 row)
```

We can use the VERBOSE clause when calling VACUUM to observe what is going on:

```
=> VACUUM VERBOSE vac;
INFO:  vacuuming "public.vac"
INFO:  table "vac": found 0 removable, 2 nonremovable row versions
in 1 out of 1 pages
DETAIL:  1 dead row versions cannot be removed yet, oldest xmin: 834
Skipped 0 pages due to buffer pins, 0 frozen pages.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

The output shows the following information:

- VACUUM has detected no tuples that can be removed (0 REMOVABLE).
- Two tuples must not be removed (2 NONREMOVABLE).
- One of the nonremovable tuples is dead (1 DEAD), the other is in use.
- The current horizon respected by VACUUM (OLDEST XMIN) is the horizon of the active transaction.

Once the active transaction completes, the database horizon moves forward, and vacuuming can continue:

```
=> COMMIT;
```

```
=> VACUUM VERBOSE vac;
INFO:  vacuuming "public.vac"
INFO:  scanned index "vac_s" to remove 1 row versions
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO:  table "vac": removed 1 dead item identifiers in 1 pages
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO:  index "vac_s" now contains 1 row versions in 2 pages
DETAIL:  1 index row versions were removed.
0 index pages were newly deleted.
0 index pages are currently deleted, of which 0 are currently
reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO:  table "vac": found 1 removable, 1 nonremovable row versions
in 1 out of 1 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 836
Skipped 0 pages due to buffer pins, 0 frozen pages.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

VACUUM has detected and removed a dead tuple beyond the new database horizon.

Now the page contains no outdated row versions; the only version remaining is the current one:

```
=> SELECT * FROM heap_page('vac',0);
 ctid  | state  | xmin  | xmax | hhu | hot | t_ctid
-------+--------+-------+------+-----+-----+--------
 (0,1) | unused |       |      |     |     |
 (0,2) | unused |       |      |     |     |
 (0,3) | normal | 835 c | 0 a  |     |     | (0,3)
(3 rows)
```

The index also contains only one entry:

```
=> SELECT * FROM index_page('vac_s',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,3) | f
(1 row)
```

6.3 Vacuum Phases

The mechanism of vacuuming seems quite simple, but this impression is misleading. After all, both tables and indexes have to be processed concurrently, without blocking other processes. To enable such operation, vacuuming of each table is carried out in several phases.[1]

It all starts with scanning a table in search of dead tuples; if found, they are first removed from indexes and then from the table itself. If too many dead tuples have to be vacuumed in one go, this process is repeated. Eventually, heap truncation may be performed.

Heap Scan

In the first phase, a *heap scan* is performed.[2] The scanning process takes the visibility map into account: all pages tracked in this map are skipped because they are sure to contain no outdated tuples. If a tuple is beyond the horizon and is not required anymore, its ID is added to a special tid array. Such tuples cannot be removed yet because they may still be referenced from indexes.

The tid array resides in the local memory of the VACUUM process; the size of the allocated memory chunk is defined by the *maintenance_work_mem* parameter. The whole chunk is allocated at once rather than on demand. However, the allocated memory never exceeds the volume required in the worst-case scenario, so if the table is small, vacuuming may use less memory than specified in this parameter.

64MB

Index Vacuuming

The first phase can have two outcomes: either the table is scanned in full, or the memory allocated for the tid array is filled up before this operation completes. In any case, *index vacuuming* begins.[3] In this phase, *each* of the indexes created on

[1] backend/access/heap/vacuumlazy.c, heap_vacuum_rel function
[2] backend/access/heap/vacuumlazy.c, lazy_scan_heap function
[3] backend/access/heap/vacuumlazy.c, lazy_vacuum_all_indexes function

the table is *fully scanned* to find all the entries that refer to the tuples registered in the tid array. These entries are removed from index pages.

> An index can help you quickly get to a heap tuple by its index key, but there is no way to quickly find an index entry by the corresponding tuple ID. This functionality is currently being implemented for B-trees,[1] but this work is not completed yet.

If there are several indexes bigger than the *min_parallel_index_scan_size* value, they can be vacuumed by background workers running in parallel. Unless the level of parallelism is explicitly defined by the parallel *N* clause, VACUUM launches one worker per suitable index (within the general limits imposed on the number of background workers).[2] One index cannot be processed by several workers. 512kB v. 13

During the index vacuuming phase, PostgreSQL updates the free space map and calculates statistics on vacuuming. However, this phase is skipped if rows are only inserted (and are neither deleted nor updated) because the table contains no dead tuples in this case. Then an index scan will be forced only once at the very end, as part of a separate phase of *index cleanup*.[3]

The index vacuuming phase leaves no references to outdated heap tuples in indexes, but the tuples themselves are still present in the table. It is perfectly normal: index scans cannot find any dead tuples, while sequential scans of the table rely on visibility rules to filter them out.

Heap Vacuuming

Then the *heap vacuuming* phase begins.[4] The table is scanned again to remove the tuples registered in the tid array and free the corresponding pointers. Now that all the related index references have been removed, it can be done safely.

The space recovered by VACUUM is reflected in the free space map, while the pages that now contain only the current tuples visible in all snapshots are tagged in the visibility map.

[1] commitfest.postgresql.org/21/1802
[2] postgresql.org/docs/14/bgworker.html
[3] backend/access/heap/vacuumlazy.c, lazy_cleanup_all_indexes function
backend/access/nbtree/nbtree.c, btvacuumcleanup function
[4] backend/access/heap/vacuumlazy.c, lazy_vacuum_heap function

If the table was not read in full during the heap scan phase, the tid array is cleared, and the heap scan is resumed from where it left off last time.

Heap Truncation

Vacuumed heap pages contain some free space; occasionally, you may be lucky to clear the whole page. If you get several empty pages at the end of the file, vacuuming can "bite off" this tail and return the reclaimed space to the operating system. It happens during *heap truncation*,[1] which is the final vacuum phase.

p. 232 Heap truncation requires a short exclusive lock on the table. To avoid holding other processes for too long, attempts to acquire a lock do not exceed five seconds.

Since the table has to be locked, truncation is only performed if the empty tail takes at least $\frac{1}{16}$ of the table or has reached the length of 1,000 pages. These thresholds are hardcoded and cannot be configured.

v. 12 If, despite all these precautions, table locks still cause any issues, truncation can be disabled altogether using the *vacuum_truncate* and *toast.vacuum_truncate* storage parameters.

6.4 Analysis

When talking about vacuuming, we have to mention yet another task that is closely related to it, even though there is no formal connection between them. p. 308 It is *analysis*,[2] or gathering statistical information for the query planner. The collected statistics include the number of rows (pg_class.reltuples) and pages (pg_class.relpages) in relations, data distribution within columns, and some other information.

You can run the analysis manually using the ANALYZE command,[3] or combine it with vacuuming by calling VACUUM ANALYZE. However, these two tasks are still performed sequentially, so there is no difference in terms of performance.

[1] backend/access/heap/vacuumlazy.c, lazy_truncate_heap function
[2] postgresql.org/docs/14/routine-vacuuming.html#VACUUM-FOR-STATISTICS
[3] backend/commands/analyze.c

Historically, VACUUM ANALYZE appeared first, in version 6.1, while a separate ANALYZE command was not implemented until version 7.2. In earlier versions, statistics were collected by a TCL script.

Automatic vacuum and analysis are set up in a similar way, so it makes sense to discuss them together.

6.5 Automatic Vacuum and Analysis

Unless the database horizon is held up for a long time, routine vacuuming should cope with its work. But how often do we need to call the VACUUM command?

If a frequently updated table is vacuumed too seldom, it will grow bigger than desired. Besides, it may accumulate too many changes, and then the next VACUUM run will have to make several passes over the indexes.

If the table is vacuumed too often, the server will be busy with maintenance instead of useful work.

Furthermore, typical workloads may change over time, so having a fixed vacuuming schedule will not help anyway: the more often the table is updated, the more often it has to be vacuumed.

This problem is solved by *autovacuum*,[1] which launches vacuum and analysis processes based on the intensity of table updates.

About the Autovacuum Mechanism

When autovacuum is enabled (*autovacuum* configuration parameter is on), the autovacuum launcher process is always running in the system. This process defines the autovacuum schedule and maintains the list of "active" databases based on usage statistics. Such statistics are collected if the *track_counts* parameter is enabled. Do not switch off these parameters, otherwise autovacuum will not work.

on

on

[1] postgresql.org/docs/14/routine-vacuuming.html#AUTOVACUUM

1min Once in *autovacuum_naptime*, the autovacuum launcher starts an autovacuum worker[1] for each active database in the list (these workers are spawned by postmaster, as usual). Consequently, if there are N active databases in the cluster, N workers are spawned within the *autovacuum_naptime* interval. But the total number of autovacuum workers running in parallel cannot exceed the threshold defined
3 by the *autovacuum_max_workers* parameter.

Autovacuum workers are very similar to regular background workers, but they appeared much earlier than this general mechanism of task management. It was decided to leave the autovacuum implementation unchanged, so autovacuum workers do not use *max_worker_processes* slots.

Once started, the background worker connects to the specified database and builds two lists:

- the list of all tables, materialized views, and TOAST tables to be vacuumed
- the list of all tables and materialized views to be analyzed (TOAST tables are not analyzed because they are always accessed via an index)

Then the selected objects are vacuumed or analyzed one by one (or undergo both operations), and once the job is complete, the worker is terminated.

Automatic vacuuming works similar to the manual one initiated by the VACUUM command, but there are some nuances:

- Manual vacuuming accumulates tuple IDs in a memory chunk of the *maintenance_work_mem* size. However, using the same limit for autovacuum is undesirable, as it can result in excessive memory consumption: there may be several autovacuum workers running in parallel, and each of them will get *maintenance_work_mem* of memory at once. Instead, PostgreSQL provides a separate memory limit for autovacuum processes, which is defined by the *autovacuum_work_mem* parameter.

-1 By default, the *autovacuum_work_mem* parameter falls back on the regular *maintenance_work_mem* limit, so if the *autovacuum_max_workers* value is high, you may have to adjust the *autovacuum_work_mem* value accordingly.

[1] backend/postmaster/autovacuum.c

- Concurrent processing of several indexes created on one table can be performed only by manual vacuuming; using autovacuum for this purpose would result in a large number of parallel processes, so it is not allowed.

If a worker fails to complete all the scheduled tasks within the *autovacuum_naptime* interval, the autovacuum launcher spawns another worker to be run in parallel in that database. The second worker will build its own lists of objects to be vacuumed and analyzed and will start processing them. There is no parallelism at the table level; only *different* tables can be processed concurrently.

Which Tables Need to be Vacuumed?

You can disable autovacuum at the table level—although it is hard to imagine why it could be necessary. There are two storage parameters provided for this purpose, one for regular tables and the other for TOAST tables:

- *autovacuum_enabled*
- *toast.autovacuum_enabled*

In usual circumstances, autovacuum is triggered either by accumulation of dead tuples or by insertion of new rows. *p. 149*

Dead tuple accumulation. Dead tuples are constantly being counted by the statistics collector; their current number is shown in the system catalog table called pg_stat_all_tables.

It is assumed that dead tuples have to be vacuumed if they exceed the threshold defined by the following two parameters:

- *autovacuum_vacuum_threshold*, which specifies the number of dead tuples (an absolute value) 50
- *autovacuum_vacuum_scale_factor*, which sets the fraction of dead tuples in a table 0.2

Vacuuming is required if the following condition is satisfied:

pg_stat_all_tables.n_dead_tup >
autovacuum_vacuum_threshold +
autovacuum_vacuum_scale_factor × pg_class.reltuples

The main parameter here is of course *autovacuum_vacuum_scale_factor*: its value is important for large tables (and it is large tables that are likely to cause the majority of issues). The default value of 20 % seems too big and may have to be significantly reduced.

For different tables, optimal parameter values may vary: they largely depend on the table size and workload type. It makes sense to set more or less adequate initial values and then override them for particular tables using storage parameters:

- *autovacuum_vacuum_threshold* and *toast.autovacuum_vacuum_threshold*
- *autovacuum_vacuum_scale_factor* and *toast.autovacuum_vacuum_scale_factor*

v. 13 **Row insertions.** If rows are only inserted and are neither deleted nor updated, the table contains no dead tuples. But such tables should also be vacuumed to freeze heap tuples in advance and update the visibility map (thus enabling index-only scans).

p. 143
p. 381

A table will be vacuumed if the number of rows inserted since the previous vacuuming exceeds the threshold defined by another similar pair of parameters:

1000 - *autovacuum_vacuum_insert_threshold*
0.2 - *autovacuum_vacuum_insert_scale_factor*

The formula is as follows:

pg_stat_all_tables.n_ins_since_vacuum >
autovacuum_vacuum_insert_threshold +
autovacuum_vacuum_insert_scale_factor × pg_class.reltuples

Like in the previous example, you can override these values at the table level using storage parameters:

- *autovacuum_vacuum_insert_threshold* and its TOAST counterpart
- *autovacuum_vacuum_insert_scale_factor* and its TOAST counterpart

Which Tables Need to Be Analyzed?

Automatic analysis needs to process only modified rows, so the calculations are a bit simpler than those for autovacuum.

It is assumed that a table has to be analyzed if the number of rows modified since the previous analysis exceeds the threshold defined by the following two configuration parameters:

- *autovacuum_analyze_threshold* 50
- *autovacuum_analyze_scale_factor* 0.1

Autoanalysis is triggered if the following condition is met:

$$\text{pg_stat_all_tables.n_mod_since_analyze} > \textit{autovacuum_analyze_threshold} + \textit{autovacuum_analyze_scale_factor} \times \text{pg_class.reltuples}$$

To override autoanalysis settings for particular tables, you can use the same-name storage parameters:

- *autovacuum_analyze_threshold*
- *autovacuum_analyze_scale_factor*

Since TOAST tables are not analyzed, they have no corresponding parameters.

Autovacuum in Action

To formalize everything said in this section, let's create two views that show which tables currently need to be vacuumed and analyzed.[1] The function used in these views returns the current value of the passed parameter, taking into account that this value can be redefined at the table level:

```
=> CREATE FUNCTION p(param text, c pg_class) RETURNS float
AS $$
  SELECT coalesce(
    -- use storage parameter if set
    (SELECT option_value
     FROM   pg_options_to_table(c.reloptions)
     WHERE  option_name = CASE
              -- for TOAST tables the parameter name is different
              WHEN c.relkind = 't' THEN 'toast.' ELSE ''
            END || param
    ),
    -- else take the configuration parameter value
    current_setting(param)
  )::float;
$$ LANGUAGE sql;
```

This is how a vacuum-related view can look like:

```
=> CREATE VIEW need_vacuum AS
WITH c AS (
  SELECT c.oid,
    greatest(c.reltuples, 0) reltuples,
    p('autovacuum_vacuum_threshold', c) threshold,
    p('autovacuum_vacuum_scale_factor', c) scale_factor,
    p('autovacuum_vacuum_insert_threshold', c) ins_threshold,
    p('autovacuum_vacuum_insert_scale_factor', c) ins_scale_factor
  FROM pg_class c
  WHERE c.relkind IN ('r','m','t')
)
SELECT st.schemaname || '.' || st.relname AS tablename,
  st.n_dead_tup AS dead_tup,
  c.threshold + c.scale_factor * c.reltuples AS max_dead_tup,
  st.n_ins_since_vacuum AS ins_tup,
  c.ins_threshold + c.ins_scale_factor * c.reltuples AS max_ins_tup,
  st.last_autovacuum
FROM pg_stat_all_tables st
  JOIN c ON c.oid = st.relid;
```

[1] backend/postmaster/autovacuum.c, relation_needs_vacanalyze function

The max_dead_tup column shows the number of dead tuples that will trigger autovacuum, whereas the max_ins_tup column shows the threshold value related to insertion.

Here is a similar view for analysis:

```
=> CREATE VIEW need_analyze AS
WITH c AS (
  SELECT c.oid,
    greatest(c.reltuples, 0) reltuples,
    p('autovacuum_analyze_threshold', c) threshold,
    p('autovacuum_analyze_scale_factor', c) scale_factor
  FROM pg_class c
  WHERE c.relkind IN ('r','m')
)
SELECT st.schemaname || '.' || st.relname AS tablename,
  st.n_mod_since_analyze AS mod_tup,
  c.threshold + c.scale_factor * c.reltuples AS max_mod_tup,
  st.last_autoanalyze
FROM pg_stat_all_tables st
  JOIN c ON c.oid = st.relid;
```

The max_mod_tup column shows the threshold value for autoanalysis.

To speed up the experiment, we will be starting autovacuum every second:

```
=> ALTER SYSTEM SET autovacuum_naptime = '1s';
=> SELECT pg_reload_conf();
```

Let's truncate the vac table and then insert 1,000 rows. Note that autovacuum is turned off at the table level.

```
=> TRUNCATE TABLE vac;
=> INSERT INTO vac(id,s)
  SELECT id, 'A' FROM generate_series(1,1000) id;
```

Here is what our vacuum-related view will show:

```
=> SELECT * FROM need_vacuum WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]---+-----------
tablename       | public.vac
dead_tup        | 0
max_dead_tup    | 50
ins_tup         | 1000
max_ins_tup     | 1000
last_autovacuum |
```

The actual threshold value is max_dead_tup = 50, although the formula listed above suggests that it should be $50 + 0.2 \times 1000 = 250$. The thing is that statistics on this table are not available yet since the INSERT command does not update it:

```
=> SELECT reltuples FROM pg_class WHERE relname = 'vac';
 reltuples
-----------
        -1
(1 row)
```

v. 14 The pg_class.reltuples value is set to -1; this special constant is used instead of zero to differentiate between a table without any statistics and a really empty table that has already been analyzed. For the purpose of calculation, the negative value is taken as zero, which gives us $50 + 0.2 \times 0 = 50$.

The value of max_ins_tup = 1000 differs from the projected value of 1,200 for the same reason.

Let's have a look at the analysis view:

```
=> SELECT * FROM need_analyze WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]----+-----------
tablename        | public.vac
mod_tup          | 1006
max_mod_tup      | 50
last_autoanalyze |
```

We have updated (inserted in this case) 1,000 rows; as a result, the threshold is exceeded: since the table size is unknown, it is currently set to 50. It means that autoanalysis will be triggered immediately when we turn it on:

```
=> ALTER TABLE vac SET (autovacuum_enabled = on);
```

Once the table analysis completes, the threshold is reset to an adequate value of 150 rows.

```
=> SELECT reltuples FROM pg_class WHERE relname = 'vac';
 reltuples
-----------
      1000
(1 row)
```

```
=> SELECT * FROM need_analyze WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]----+-----------------------------
tablename        | public.vac
mod_tup          | 0
max_mod_tup      | 150
last_autoanalyze | 2022-11-25 22:57:44.714517+03
```

Let's get back to autovacuum:

```
=> SELECT * FROM need_vacuum WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]---+-----------
tablename       | public.vac
dead_tup        | 0
max_dead_tup    | 250
ins_tup         | 1000
max_ins_tup     | 1200
last_autovacuum |
```

The max_dead_tup and max_ins_tup values have also been updated based on the actual table size discovered by the analysis.

Vacuuming will be started if at least one of the following conditions is met:

- More than 250 dead tuples are accumulated.
- More than 200 rows are inserted into the table. v. 13

Let's turn off autovacuum again and update 251 rows so that the threshold value is exceeded by one:

```
=> ALTER TABLE vac SET (autovacuum_enabled = off);
=> UPDATE vac SET s = 'B' WHERE id <= 251;
=> SELECT * FROM need_vacuum WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]---+-----------
tablename       | public.vac
dead_tup        | 251
max_dead_tup    | 250
ins_tup         | 1000
max_ins_tup     | 1200
last_autovacuum |
```

Now the trigger condition is satisfied. Let's enable autovacuum; after a while, we will see that the table has been processed, and its usage statistics has been reset:

```
=> ALTER TABLE vac SET (autovacuum_enabled = on);
=> SELECT * FROM need_vacuum WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]---+-----------------------------
tablename       | public.vac
dead_tup        | 0
max_dead_tup    | 250
ins_tup         | 0
max_ins_tup     | 1200
last_autovacuum | 2022-11-25 22:57:51.025012+03
```

6.6 Managing the Load

Operating at the page level, vacuuming does not block other processes; but nevertheless, it increases the system load and can have a noticeable impact on performance.

Vacuum Throttling

To control vacuuming intensity, PostgreSQL makes regular pauses in table pro-
200 cessing. After completing about *vacuum_cost_limit* units of work, the process falls
0 asleep and remains idle for the *vacuum_cost_delay* time interval.

The default zero value of *vacuum_cost_delay* means that routine vacuuming actually never sleeps, so the exact *vacuum_cost_limit* value makes no difference. It is assumed that if administrators have to resort to manual vacuuming, they are likely to expect its completion as soon as possible.

If the sleep time is set, then the process will pause each time it has spent *vac-*
p. 169 *uum_cost_limit* units of work on page processing in the buffer cache. The cost of
1 each page read is estimated at *vacuum_cost_page_hit* units if the page is found in
2 the buffer cache, or *vacuum_cost_page_miss* units otherwise.[1] If a clean page is dirt-
20 ied by vacuum, it adds another *vacuum_cost_page_dirty* units.[2]

[1] backend/storage/buffer/bufmgr.c, ReadBuffer_common function
[2] backend/storage/buffer/bufmgr.c, MarkBufferDirty function

If you keep the default value of the *vacuum_cost_limit* parameter, VACUUM can process up to 200 pages per cycle in the best-case scenario (if all the pages are cached, and no pages are dirtied by VACUUM) and only nine pages in the worst case (if all the pages are read from disk and become dirty).

Autovacuum Throttling

Throttling for autovacuum[1] is quite similar to VACUUM throttling. However, autovacuum can be run with a different intensity as it has its own set of parameters:

- *autovacuum_vacuum_cost_limit* -1
- *autovacuum_vacuum_cost_delay* 2ms

If any of these parameters is set to –1, it falls back on the corresponding parameter for regular VACUUM. Thus, the *autovacuum_vacuum_cost_limit* parameter relies on the *vacuum_cost_limit* value by default.

> Prior to version 12, the default value of *autovacuum_vacuum_cost_delay* was 20 ms, and it led to very poor performance on modern hardware.

Autovacuum work units are limited to *autovacuum_vacuum_cost_limit* per cycle, and since they are shared between all the workers, the overall impact on the system remains roughly the same, regardless of their number. So if you need to speed up autovacuum, both the *autovacuum_max_workers* and *autovacuum_vacuum_cost_limit* values should be increased proportionally.

If required, you can override these settings for particular tables by setting the following storage parameters:

- *autovacuum_vacuum_cost_delay* and *toast.autovacuum_vacuum_cost_delay*
- *autovacuum_vacuum_cost_limit* and *toast.autovacuum_vacuum_cost_limit*

[1] backend/postmaster/autovacuum.c, autovac_balance_cost function

6.7 Monitoring

If vacuuming is monitored, you can detect situations when dead tuples cannot be removed in one go, as references to them do not fit the *maintenance_work_mem* memory chunk. In this case, all the indexes will have to be fully scanned several times. It can take a substantial amount of time for large tables, thus creating a significant load on the system. Even though queries will not be blocked, extra I/O operations can seriously limit system throughput.

Such issues can be corrected either by vacuuming the table more often (so that each run cleans up fewer tuples) or by allocating more memory.

Monitoring Vacuum

When run with the VERBOSE clause, the VACUUM command performs the cleanup and
v. 9.6 displays the status report, whereas the pg_stat_progress_vacuum view shows the current state of the started process.

v. 13 There is also a similar view for analysis (pg_stat_progress_analyze), even though it is usually performed very fast and is unlikely to cause any issues.

Let's insert more rows into the table and update them all so that VACUUM has to run for a noticeable period of time:

```
=> TRUNCATE vac;
=> INSERT INTO vac(id,s)
  SELECT id, 'A' FROM generate_series(1,500000) id;
=> UPDATE vac SET s  = 'B';
```

For the purpose of this demonstration, we will limit the amount of memory allocated for the tid array by 1 MB:

```
=> ALTER SYSTEM SET maintenance_work_mem = '1MB';
=> SELECT pg_reload_conf();
```

Launch the VACUUM command and query the pg_stat_progress_vacuum view several times while it is running:

```
=> VACUUM VERBOSE vac;
```

```
=> SELECT * FROM pg_stat_progress_vacuum \gx
-[ RECORD 1 ]------+------------------
pid                | 14654
datid              | 16391
datname            | internals
relid              | 16479
phase              | vacuuming indexes
heap_blks_total    | 17242
heap_blks_scanned  | 3009
heap_blks_vacuumed | 0
index_vacuum_count | 0
max_dead_tuples    | 174761
num_dead_tuples    | 174522
=> SELECT * FROM pg_stat_progress_vacuum \gx
-[ RECORD 1 ]------+------------------
pid                | 14654
datid              | 16391
datname            | internals
relid              | 16479
phase              | vacuuming indexes
heap_blks_total    | 17242
heap_blks_scanned  | 17242
heap_blks_vacuumed | 6017
index_vacuum_count | 2
max_dead_tuples    | 174761
num_dead_tuples    | 150956
```

In particular, this view shows:

- phase—the name of the current vacuum phase (I have described the main ones, but there are actually more of them[1])
- heap_blks_total—the total number of pages in a table
- heap_blks_scanned—the number of scanned pages
- heap_blks_vacuumed—the number of vacuumed pages
- index_vacuum_count—the number of index scans

[1] postgresql.org/docs/14/progress-reporting.html#VACUUM-PHASES

The overall vacuuming progress is defined by the ratio of heap_blks_vacuumed to heap_blks_total, but you have to keep in mind that it changes in spurts because of index scans. In fact, it is more important to pay attention to the number of vacuum cycles: if this value is greater than one, it means that the allocated memory was not enough to complete vacuuming in one go.

You can see the whole picture in the output of the VACUUM VERBOSE command, which has already finished by this time:

```
INFO:  vacuuming "public.vac"
INFO:  scanned index "vac_s" to remove 174522 row versions
DETAIL:  CPU: user: 0.02 s, system: 0.00 s, elapsed: 0.05 s
INFO:  table "vac": removed 174522 dead item identifiers in
3009 pages
DETAIL:  CPU: user: 0.01 s, system: 0.00 s, elapsed: 0.06 s
INFO:  scanned index "vac_s" to remove 174522 row versions
DETAIL:  CPU: user: 0.02 s, system: 0.00 s, elapsed: 0.05 s
INFO:  table "vac": removed 174522 dead item identifiers in
3009 pages
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO:  scanned index "vac_s" to remove 150956 row versions
DETAIL:  CPU: user: 0.02 s, system: 0.00 s, elapsed: 0.07 s
INFO:  table "vac": removed 150956 dead item identifiers in
2603 pages
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO:  index "vac_s" now contains 500000 row versions in
932 pages
DETAIL:  500000 index row versions were removed.
433 index pages were newly deleted.
433 index pages are currently deleted, of which 0 are
currently reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO:  table "vac": found 500000 removable, 500000
nonremovable row versions in 17242 out of 17242 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest
xmin: 851
Skipped 0 pages due to buffer pins, 0 frozen pages.
CPU: user: 0.20 s, system: 0.01 s, elapsed: 0.47 s.
VACUUM
```

index vacuum
table vacuum
index vacuum
table vacuum
index vacuum
table vacuum

All in all, there have been three index scans; each scan has removed 174,522 pointers to dead tuples at the most. This value is defined by the number of tid pointers (each of them takes 6 bytes) that can fit into an array of the *main-*

tenance_work_mem size. The maximum size possible is shown by pg_stat_progress_vacuum.max_dead_tuples, but the actually used space is always a bit smaller. It guarantees that when the next page is read, all its pointers to dead tuples, no matter how many of them are located in this page, will fit into the remaining memory.

Monitoring Autovacuum

The main approach to monitoring autovacuum is to print its status information (which is similar to the output of the VACUUM VERBOSE command) into the server log for further analysis. If the *log_autovacuum_min_duration* parameter is set to zero, all autovacuum runs are logged: -1

```
=> ALTER SYSTEM SET log_autovacuum_min_duration = 0;

=> SELECT pg_reload_conf();

=> UPDATE vac SET s = 'C';
UPDATE 500000

postgres$ tail -n 13 /home/postgres/logfile
2022-11-25 22:58:12.754 MSK [17505] LOG:  automatic vacuum of table
"internals.public.vac": index scans: 3
pages: 0 removed, 17242 remain, 0 skipped due to pins, 0
skipped frozen
tuples: 500000 removed, 500000 remain, 0 are dead but not
yet removable, oldest xmin: 853
index scan needed: 8622 pages from table (50.01% of total)
had 500000 dead item identifiers removed
index "vac_s": pages: 1428 in total, 496 newly deleted, 929
currently deleted, 433 reusable
avg read rate: 13.152 MB/s, avg write rate: 17.475 MB/s
buffer usage: 45852 hits, 5855 misses, 7780 dirtied
WAL usage: 41366 records, 15059 full page images, 96970827
bytes
system usage: CPU: user: 0.27 s, system: 0.20 s, elapsed:
3.47 s
2022-11-25 22:58:13.146 MSK [17505] LOG:  automatic analyze of table
"internals.public.vac"
avg read rate: 40.765 MB/s, avg write rate: 0.020 MB/s
buffer usage: 15353 hits, 2035 misses, 1 dirtied
system usage: CPU: user: 0.10 s, system: 0.00 s, elapsed:
0.38 s
```

To track the list of tables that have to be vacuumed and analyzed, you can use the need_vacuum and need_analyze views, which we have already reviewed. If this list grows, it means that autovacuum does not cope with the load and has to be sped up by either reducing the gap (*autovacuum_vacuum_cost_delay*) or increasing the amount of work done between the gaps (*autovacuum_vacuum_cost_limit*). It is not unlikely that the degree of parallelism will also have to be increased (*autovacuum_max_workers*).

7

Freezing

7.1 Transaction ID Wraparound

In PostgreSQL, a transaction ID takes 32 bits. Four billions seems to be quite a big number, but it can be exhausted very fast if the system is being actively used. For example, for an average load of 1,000 transactions per second (excluding virtual ones), it will happen in about six weeks of continuous operation.

Once all the numbers are used up, the counter has to be reset to start the next round (this situation is called a "wraparound"). But a transaction with a smaller ID can only be considered older than another transaction with a bigger ID if the assigned numbers are always increasing. So the counter cannot simply start using the same numbers anew after being reset.

Allocating 64 bits for transaction IDs would have eliminated this problem altogether, so why doesn't PostgreSQL take advantage of it? The thing is that each tuple header has to store IDs for two transactions: xmin and xmax. The header is quite big already (at least 24 bytes if data alignment is taken into account), and adding more bits would have given another 8 bytes. *p. 70*

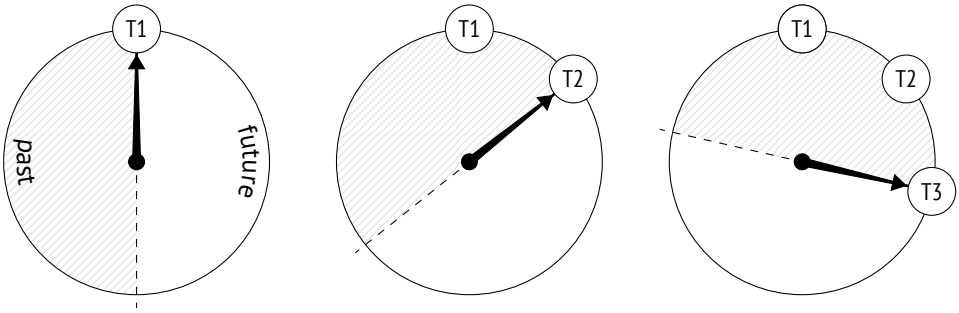
> PostgreSQL does implement 64-bit transaction IDs[1] that extend a regular ID by a 32-bit epoch, but they are used only internally and never get into data pages.

To correctly handle wraparound, PostgreSQL has to compare the age of transactions (defined as the number of subsequent transactions that have appeared since the start of this transaction) rather than transaction IDs. Thus, instead of the terms *less than* and *greater than* we should use the concepts of *older* (precedes) and *younger* (follows).

[1] include/access/transam.h, FullTransactionId type

In the code, this comparison is implemented by simply using the 32-bit arithmetic: first the difference between 32-bit transaction IDs is found, and then this result is compared to zero.[1]

To better visualize this idea, you can imagine a sequence of transaction IDs as a clock face. For each transaction, half of the circle in the clockwise direction will be in the future, while the other half will be in the past.



However, this visualization has an unpleasant catch. An old transaction (T1) is in the remote past as compared to more recent transactions. But sooner or later a new transaction will see it in the half of the circle related to the future. If it were really so, it would have a catastrophic impact: from now on, all newer transactions would not see the changes made by transaction T1.

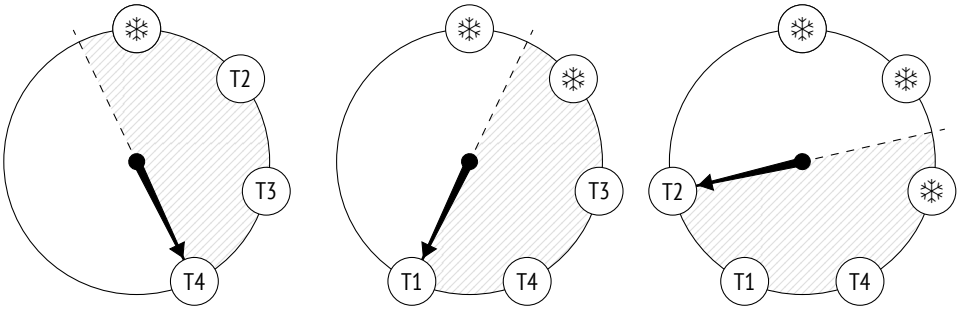
7.2 Tuple Freezing and Visibility Rules

To prevent such "time travel," vacuuming performs one more task (in addition to page cleanup):[2] it searches for tuples that are beyond the database horizon (so they
p. 244 are visible in all snapshots) and tags them in a special way, that is, *freezes* them.

For frozen tuples, visibility rules do not have to take xmin into account since such tuples are known to be visible in all snapshots, so this transaction ID can be safely reused.

[1] backend/access/transam/transam.c, TransactionIdPrecedes function
[2] postgresql.org/docs/14/routine-vacuuming.html#VACUUM-FOR-WRAPAROUND

You can imagine that the xmin transaction ID is replaced in frozen tuples by a hypothetical "minus infinity" (pictured as a snowflake below); it is a sign that this tuple is created by a transaction that is so far in the past that its actual ID does not matter anymore. Yet in reality xmin remains unchanged, whereas the freezing attribute is defined by a combination of two hint bits: committed and aborted.



Many sources (including the documentation) mention FrozenTransactionId = 2. It is the "minus infinity" that I have already referred to—this value used to replace xmin in versions prior to 9.4, but now hint bits are employed instead. As a result, the original transaction ID remains in the tuple, which is convenient for both debugging and support. Old systems can still contain the obsolete FrozenTransactionId, even if they have been upgraded to higher versions.

The xmax transaction ID does not participate in freezing in any way. It is only present in outdated tuples, and once such tuples stop being visible in all snapshots (which means that the xmax ID is beyond the database horizon), they will be vacuumed away.

Let's create a new table for our experiments. The *fillfactor* parameter should be set to the lowest value so that each page can accommodate only two tuples—it will be easier to track the progress this way. We will also disable autovacuum to make sure that the table is only cleaned up on demand.

```
=> CREATE TABLE tfreeze(
  id integer,
  s char(300)
)
WITH (fillfactor = 10, autovacuum_enabled = off);
```

We are going to create yet another flavor of the function that displays heap pages using pageinspect. Dealing with a range of pages, it will show the values of the freezing attribute (f) and the xmin transaction age for each tuple (it will have to call the age system function—the age itself is not stored in heap pages, of course):

```
=> CREATE FUNCTION heap_page(
  relname text, pageno_from integer, pageno_to integer
)
RETURNS TABLE(
  ctid tid, state text,
  xmin text, xmin_age integer, xmax text
) AS $$
SELECT (pageno,lp)::text::tid AS ctid,
       CASE lp_flags
         WHEN 0 THEN 'unused'
         WHEN 1 THEN 'normal'
         WHEN 2 THEN 'redirect to '||lp_off
         WHEN 3 THEN 'dead'
       END AS state,
       t_xmin || CASE
         WHEN (t_infomask & 256+512) = 256+512 THEN ' f'
         WHEN (t_infomask & 256) > 0 THEN ' c'
         WHEN (t_infomask & 512) > 0 THEN ' a'
         ELSE ''
       END AS xmin,
       age(t_xmin) AS xmin_age,
       t_xmax || CASE
         WHEN (t_infomask & 1024) > 0 THEN ' c'
         WHEN (t_infomask & 2048) > 0 THEN ' a'
         ELSE ''
       END AS xmax
FROM generate_series(pageno_from, pageno_to) p(pageno),
     heap_page_items(get_raw_page(relname, pageno))
ORDER BY pageno, lp;
$$ LANGUAGE sql;
```

Now let's insert some rows into the table and run the VACUUM command that will immediately create the visibility map.

```
=> CREATE EXTENSION IF NOT EXISTS pg_visibility;
```

```
=> INSERT INTO tfreeze(id, s)
  SELECT id, 'FOO'||id FROM generate_series(1,100) id;
INSERT 0 100
```

We are going to observe the first two heap pages using the pg_visibility extension. When vacuuming completes, both pages get tagged in the visibility map (all_visible) but not in the freeze map (all_frozen), as they still contain some unfrozen tuples: v. 9.6

```
=> VACUUM tfreeze;
=> SELECT *
FROM generate_series(0,1) g(blkno),
     pg_visibility_map('tfreeze',g.blkno)
ORDER BY g.blkno;
 blkno | all_visible | all_frozen
-------+-------------+------------
     0 | t           | f
     1 | t           | f
(2 rows)
```

The xmin_age of the transaction that has created the rows equals 1 because it is the latest transaction performed in the system:

```
=> SELECT * FROM heap_page('tfreeze',0,1);
 ctid  | state  | xmin  | xmin_age | xmax
-------+--------+-------+----------+------
 (0,1) | normal | 856 c |        1 | 0 a
 (0,2) | normal | 856 c |        1 | 0 a
 (1,1) | normal | 856 c |        1 | 0 a
 (1,2) | normal | 856 c |        1 | 0 a
(4 rows)
```

7.3 Managing Freezing

There are four main parameters that control freezing. All of them represent transaction age and define when the following events happen:

- Freezing starts (*vacuum_freeze_min_age*).
- Aggressive freezing is performed (*vacuum_freeze_table_age*).
- Freezing is forced (*autovacuum_freeze_max_age*).
- Freezing receives priority (*vacuum_failsafe_age*). v. 14

Minimal Freezing Age

50 million

The *vacuum_freeze_min_age* parameter defines the minimal freezing age of xmin transactions. The lower its value, the higher the overhead: if a row is "hot" and is actively being changed, then freezing all its newly created versions will be a wasted effort. Setting this parameter to a relatively high value allows you to wait for a while.

To observe the freezing process, let's reduce this parameter value to one:

```
=> ALTER SYSTEM SET vacuum_freeze_min_age = 1;
=> SELECT pg_reload_conf();
```

Now update one row in the zero page. The new row version will get into the same page because the *fillfactor* value is quite small:

```
=> UPDATE tfreeze SET s = 'BAR' WHERE id = 1;
```

The age of all transactions has been increased by one, and the heap pages now look as follows:

```
=> SELECT * FROM heap_page('tfreeze',0,1);
 ctid  | state  | xmin  | xmin_age | xmax
-------+--------+-------+----------+------
 (0,1) | normal | 856 c |        2 | 857
 (0,2) | normal | 856 c |        2 | 0 a
 (0,3) | normal | 857   |        1 | 0 a
 (1,1) | normal | 856 c |        2 | 0 a
 (1,2) | normal | 856 c |        2 | 0 a
(5 rows)
```

At this point, the tuples that are older than *vacuum_freeze_min_age* = 1 are subject
p. 124
to freezing. But vacuum will not process any pages tagged in the visibility map:

```
=> SELECT * FROM generate_series(0,1) g(blkno),
     pg_visibility_map('tfreeze',g.blkno)
ORDER BY g.blkno;
 blkno | all_visible | all_frozen
-------+-------------+------------
     0 | f           | f
     1 | t           | f
(2 rows)
```

The previous UPDATE command has removed the visibility bit of the zero page, so the tuple that has an appropriate xmin age in this page will be frozen. But the first page will be skipped altogether:

```
=> VACUUM tfreeze;
```

```
=> SELECT * FROM heap_page('tfreeze',0,1);
 ctid  |     state     | xmin  | xmin_age | xmax
-------+---------------+-------+----------+------
 (0,1) | redirect to 3 |       |          |
 (0,2) | normal        | 856 f |        2 | 0 a
 (0,3) | normal        | 857 c |        1 | 0 a
 (1,1) | normal        | 856 c |        2 | 0 a
 (1,2) | normal        | 856 c |        2 | 0 a
(5 rows)
```

Now the zero page appears in the visibility map again, and if nothing changes in it, vacuuming will not return to this page anymore:

```
=> SELECT * FROM generate_series(0,1) g(blkno),
     pg_visibility_map('tfreeze',g.blkno)
ORDER BY g.blkno;
 blkno | all_visible | all_frozen
-------+-------------+------------
     0 | t           | f
     1 | t           | f
(2 rows)
```

Age for Aggressive Freezing

As we have just seen, if a page contains only the current tuples that are visible in all snapshots, vacuuming will not freeze them. To overcome this constraint, PostgreSQL provides the *vacuum_freeze_table_age* parameter. It defines the transaction age that allows vacuuming to ignore the visibility map, so any heap page can be frozen.

150 million

For each table, the system catalog keeps a transaction ID for which it is known that all the older transactions are sure to be frozen. It is stored as relfrozenxid:

```
=> SELECT relfrozenxid, age(relfrozenxid)
FROM pg_class
WHERE relname = 'tfreeze';
 relfrozenxid | age
--------------+-----
          854 |   4
(1 row)
```

It is the age of this transaction that is compared to the *vacuum_freeze_table_age* value to decide whether the time has come for aggressive freezing.

v. 9.6 Thanks to the freeze map, there is no need to perform a full table scan during vacuuming: it is enough to check only those pages that do not appear in the map. Apart from this important optimization, the freeze map also brings fault tolerance: if vacuuming is interrupted, its next run will not have to get back to the pages that have already been processed and are tagged in the map.

PostgreSQL performs aggressive freezing of all pages in a table each time when the number of transactions in the system reaches the *vacuum_freeze_table_age* – *vacuum_freeze_min_age* limit (if the default values are used, it happens after each 100 million transactions). Thus, if the *vacuum_freeze_min_age* value is too big, it can lead to excessive freezing and increased overhead.

To freeze the whole table, let's reduce the *vacuum_freeze_table_age* value to four; then the condition for aggressive freezing will be satisfied:

```
=> ALTER SYSTEM SET vacuum_freeze_table_age = 4;

=> SELECT pg_reload_conf();
```

Run the VACUUM command:

```
=> VACUUM VERBOSE tfreeze;
INFO:  aggressively vacuuming "public.tfreeze"
INFO:  table "tfreeze": found 0 removable, 100 nonremovable row
versions in 50 out of 50 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 858
Skipped 0 pages due to buffer pins, 0 frozen pages.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

Now that the whole table has been analyzed, the relfrozenxid value can be advanced—heap pages are guaranteed to have no older unfrozen xmin transactions:

```
=> SELECT relfrozenxid, age(relfrozenxid)
FROM pg_class
WHERE relname = 'tfreeze';
 relfrozenxid | age
--------------+-----
          857 |   1
(1 row)
```

The first page now contains only frozen tuples:

```
=> SELECT * FROM heap_page('tfreeze',0,1);
 ctid  |     state     | xmin  | xmin_age | xmax
-------+---------------+-------+----------+------
 (0,1) | redirect to 3 |       |          |
 (0,2) | normal        | 856 f |        2 | 0 a
 (0,3) | normal        | 857 c |        1 | 0 a
 (1,1) | normal        | 856 f |        2 | 0 a
 (1,2) | normal        | 856 f |        2 | 0 a
(5 rows)
```

Besides, this page is tagged in the freeze map:

```
=> SELECT * FROM generate_series(0,1) g(blkno),
  pg_visibility_map('tfreeze',g.blkno)
ORDER BY g.blkno;
 blkno | all_visible | all_frozen
-------+-------------+------------
     0 | t           | f
     1 | t           | t
(2 rows)
```

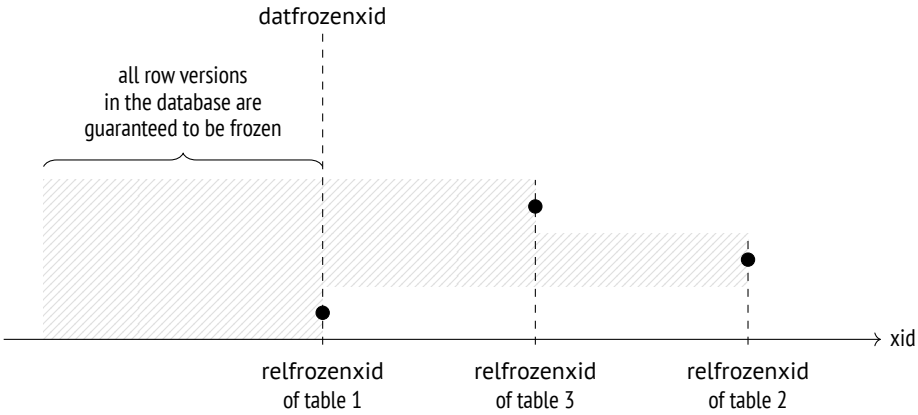
Age for Forced Autovacuum

Sometimes it is not enough to configure the two parameters discussed above to timely freeze tuples. Autovacuum might be switched off, while regular VACUUM is not being called at all (it is a very bad idea, but technically it is possible). Besides,

p. 127 some inactive databases (like template0) may not be vacuumed. PostgreSQL can handle such situations by *forcing* autovacuum in the aggressive mode.

Autovacuum is forced[1] (even if it is switched off) when there is a risk that the
200 age of some unfrozen transaction IDs in the database will exceed the *autovacu-*
million *um_freeze_max_age* value. The decision is taken based on the age of the oldest pg_class.relfrozenxid transaction in all the tables, as all the older transactions are guaranteed to be frozen. The ID of this transaction is stored in the system catalog:

```
=> SELECT datname, datfrozenxid, age(datfrozenxid) FROM pg_database;
  datname  | datfrozenxid | age
-----------+--------------+-----
 postgres  |          726 | 132
 template1 |          726 | 132
 template0 |          726 | 132
 internals |          726 | 132
(4 rows)
```



The *autovacuum_freeze_max_age* limit is set to 2 billion transactions (a bit less than half of the circle), while the default value is 10 times smaller. It is done for good reason: a big value increases the risk of transaction ID wraparound, as PostgreSQL may fail to timely freeze all the required tuples. In this case, the server must stop immediately to prevent possible issues and will have to be restarted by an administrator.

[1] backend/access/transam/varsup.c, SetTransactionIdLimit function

The *autovacuum_freeze_max_age* value also affects the size of CLOG. There is no need to keep the status of frozen transactions, and all the transactions that precede the one with the oldest datfrozenxid in the cluster are sure to be frozen. Those CLOG files that are not required anymore are removed by autovacuum.[1] p. 79

Changing the *autovacuum_freeze_max_age* parameter requires a server restart. However, all the freezing settings discussed above can also be adjusted at the table level via the corresponding storage parameters. Note that the names of all these parameters start with "auto":

- *autovacuum_freeze_min_age* and *toast.autovacuum_freeze_min_age*
- *autovacuum_freeze_table_age* and *toast.autovacuum_freeze_table_age*
- *autovacuum_freeze_max_age* and *toast.autovacuum_freeze_max_age*

Age for Failsafe Freezing

v. 14

If autovacuum is already struggling to prevent transaction ID wraparound and it is clearly a race against time, a safety switch is pulled: autovacuum will ignore the *autovacuum_vacuum_cost_delay* (*vacuum_cost_delay*) value and will stop vacuuming indexes to freeze heap tuples as soon as possible.

A failsafe freezing mode[2] is enabled if there is a risk that the age of an unfrozen transaction in the database will exceed the *vacuum_failsafe_age* value. It is assumed that this value must be higher than *autovacuum_freeze_max_age*. 1.6 billion

7.4 Manual Freezing

It is sometimes more convenient to manage freezing manually rather than rely on autovacuum.

[1] backend/commands/vacuum.c, vac_truncate_clog function
[2] backend/access/heap/vacuumlazy.c, lazy_check_wraparound_failsafe function

Freezing by Vacuum

You can initiate freezing by calling the VACUUM command with the FREEZE option. It will freeze all the heap tuples regardless of their transaction age, as if *vacuum_freeze_min_age* = 0.

v. 12 If the purpose of such a call is to freeze heap tuples as soon as possible, it makes sense to disable index vacuuming, like it is done in the failsafe mode. You can do it either explicitly, by running the VACUUM (freeze, index_cleanup false) command, or via the *vacuum_index_cleanup* storage parameter. It is rather obvious that it should not be done on a regular basis since in this case VACUUM will not be coping well with its main task of page cleanup.

Freezing Data at the Initial Loading

The data that is not expected to change can be frozen at once, while it is being loaded into the database. It is done by running the COPY command with the FREEZE option.

Tuples can be frozen during the initial loading only if the resulting table has been created or truncated within the same transaction, as both these operations acquire
p. 232 an exclusive lock on the table. This restriction is necessary because frozen tuples are expected to be visible in all snapshots, regardless of the isolation level; otherwise, transactions would suddenly see freshly-frozen tuples right as they are being uploaded. But if the lock is acquired, other transactions will not be able to get access to this table.

Nevertheless, it is still technically possible to break isolation. Let's start a new transaction at the Repeatable Read isolation level in a separate session:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT 1; -- the shapshot is built
```

Truncate the tfreeze table and insert new rows into this table within the same transaction. (If the read-only transaction had already accessed the tfreeze table, the TRUNCATE command will be blocked.)

```
=> BEGIN;
=> TRUNCATE tfreeze;
=> COPY tfreeze FROM stdin WITH FREEZE;
1 FOO
2 BAR
3 BAZ
\.
=> COMMIT;
```

Now the reading transaction sees the new data as well:

```
=> SELECT count(*) FROM tfreeze;
 count
-------
     3
(1 row)
=> COMMIT;
```

It does break isolation, but since data loading is unlikely to happen regularly, in most cases it will not cause any issues.

If you load data with freezing, the visibility map is created at once, and page headers receive the visibility attribute: v. 14 p. 120

```
=> SELECT * FROM pg_visibility_map('tfreeze',0);
 all_visible | all_frozen
-------------+------------
 t           | t
(1 row)
=> SELECT flags & 4 > 0 AS all_visible
FROM page_header(get_raw_page('tfreeze',0));
 all_visible
-------------
 t
(1 row)
```

Thus, if the data has been loaded with freezing, the table will not be processed by vacuum (as long as the data remains unchanged). Unfortunately, this functionality is not supported for TOAST tables yet: if an oversized value is loaded, vacuum will have to rewrite the whole TOAST table to set visibility attributes in all page headers. v. 14

8

Rebuilding Tables and Indexes

8.1 Full Vacuuming

Why is Routine Vacuuming not Enough?

Routine vacuuming can free more space than page pruning, but sometimes it may still be not enough.

If table or index files have grown in size, VACUUM can clean up some space within pages, but it can rarely reduce the number of pages. The reclaimed space can only be returned to the operating system if several empty pages appear at the very end of the file, which does not happen too often.

An excessive size can lead to unpleasant consequences:

- Full table (or index) scan will take longer.
- A bigger buffer cache may be required (pages are cached as a whole, so data density decreases).
- B-trees can get an extra level, which slows down index access.
- Files take up extra space on disk and in backups.

If the fraction of useful data in a file has dropped below some reasonable level, an administrator can perform *full vacuuming* by running the VACUUM FULL command.[1] In this case, the table and all its indexes are rebuilt from scratch, and the data is packed as densely as possible (taking the *fillfactor* parameter into account).

p. 106

[1] postgresql.org/docs/14/routine-vacuuming.html#VACUUM-FOR-SPACE-RECOVERY

When full vacuuming is performed, PostgreSQL first fully rebuilds the table and then each of its indexes. While an object is being rebuilt, both old and new files have to be stored on disk,[1] so this process may require a lot of free space.

You should also keep in mind that this operation fully blocks access to the table, both for reads and writes.

Estimating Data Density

For the purpose of illustration, let's insert some rows into the table:

```
=> TRUNCATE vac;
=> INSERT INTO vac(id,s)
  SELECT id, id::text FROM generate_series(1,500000) id;
```

Storage density can be estimated using the pgstattuple extension:

```
=> CREATE EXTENSION pgstattuple;
=> SELECT * FROM pgstattuple('vac') \gx
-[ RECORD 1 ]------+---------
table_len          | 70623232
tuple_count        | 500000
tuple_len          | 64500000
tuple_percent      | 91.33
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 381844
free_percent       | 0.54
```

The function reads the whole table and displays statistics on space distribution in its files. The tuple_percent field shows the percentage of space taken up by useful data (heap tuples). This value is inevitably less than 100 % because of various metadata within pages, but in this example it is still quite high.

For indexes, the displayed information differs a bit, but the avg_leaf_density field has the same meaning: it shows the percentage of useful data (in B-tree leaf pages).

[1] backend/commands/cluster.c

```
=> SELECT * FROM pgstatindex('vac_s') \gx
-[ RECORD 1 ]------+----------
version            | 4
tree_level         | 3
index_size         | 114302976
root_block_no      | 2825
internal_pages     | 376
leaf_pages         | 13576
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 53.88
leaf_fragmentation | 10.59
```

The previously used pgstattuple functions read the table or index in full to get the precise statistics. For large objects, it can turn out to be too expensive, so the extension also provides another function called pgstattuple_approx, which skips the pages tracked in the visibility map to show approximate figures.

A faster but even less accurate method is to roughly estimate the ratio between the data volume and the file size using the system catalog.[1]

Here are the current sizes of the table and its index:

```
=> SELECT pg_size_pretty(pg_table_size('vac')) AS table_size,
        pg_size_pretty(pg_indexes_size('vac')) AS index_size;
 table_size | index_size
------------+------------
 67 MB      | 109 MB
(1 row)
```

Now let's delete 90 % of all the rows:

```
=> DELETE FROM vac WHERE id % 10 != 0;
DELETE 450000
```

Routine vacuuming does not affect the file size because there are no empty pages at the end of the file:

```
=> VACUUM vac;
```

[1] wiki.postgresql.org/wiki/Show_database_bloat

```
=> SELECT pg_size_pretty(pg_table_size('vac')) AS table_size,
        pg_size_pretty(pg_indexes_size('vac')) AS index_size;
 table_size | index_size
------------+------------
 67 MB      | 109 MB
(1 row)
```

However, data density has dropped about 10 times:

```
=> SELECT vac.tuple_percent, vac_s.avg_leaf_density
FROM pgstattuple('vac') vac, pgstatindex('vac_s') vac_s;
 tuple_percent | avg_leaf_density
---------------+------------------
          9.13 |             6.71
(1 row)
```

The table and the index are currently located in the following files:

```
=> SELECT pg_relation_filepath('vac') AS vac_filepath,
        pg_relation_filepath('vac_s') AS vac_s_filepath \gx
-[ RECORD 1 ]--+-----------------
vac_filepath   | base/16391/16514
vac_s_filepath | base/16391/16515
```

Let's check what we will get after VACUUM FULL. While the command is running, its progress can be tracked in the pg_stat_progress_cluster view (which is similar to the pg_stat_progress_vacuum view provided for VACUUM): v. 12

```
=> VACUUM FULL vac;
```

```
=> SELECT * FROM pg_stat_progress_cluster \gx
-[ RECORD 1 ]-------+-----------------
pid                 | 19631
datid               | 16391
datname             | internals
relid               | 16479
command             | VACUUM FULL
phase               | rebuilding index
cluster_index_relid | 0
heap_tuples_scanned | 50000
heap_tuples_written | 50000
heap_blks_total     | 8621
heap_blks_scanned   | 8621
index_rebuild_count | 0
```

Expectedly, VACUUM FULL phases[1] differ from those of routine vacuuming.

Full vacuuming has replaced old files with new ones:

```
=> SELECT pg_relation_filepath('vac') AS vac_filepath,
        pg_relation_filepath('vac_s') AS vac_s_filepath \gx
-[ RECORD 1 ]--+-----------------
vac_filepath   | base/16391/16526
vac_s_filepath | base/16391/16529
```

Both index and table sizes are much smaller now:

```
=> SELECT pg_size_pretty(pg_table_size('vac')) AS table_size,
        pg_size_pretty(pg_indexes_size('vac')) AS index_size;
 table_size | index_size
------------+------------
 6904 kB    | 6504 kB
(1 row)
```

As a result, data density has increased. For the index, it is even higher than the original one: it is more efficient to create a B-tree from scratch based on the available data than to insert entries row by row into an already existing index:

```
=> SELECT vac.tuple_percent,
          vac_s.avg_leaf_density
FROM pgstattuple('vac') vac,
     pgstatindex('vac_s') vac_s;
 tuple_percent | avg_leaf_density
---------------+------------------
         91.23 |            91.08
(1 row)
```

Freezing

When the table is being rebuilt, PostgreSQL freezes its tuples because this operation costs almost nothing compared to the rest of the work:

[1] postgresql.org/docs/14/progress-reporting.html#CLUSTER-PHASES

```
=> SELECT * FROM heap_page('vac',0,0) LIMIT 5;
 ctid  | state  | xmin  | xmin_age | xmax
-------+--------+-------+----------+------
 (0,1) | normal | 861 f |        5 | 0 a
 (0,2) | normal | 861 f |        5 | 0 a
 (0,3) | normal | 861 f |        5 | 0 a
 (0,4) | normal | 861 f |        5 | 0 a
 (0,5) | normal | 861 f |        5 | 0 a
(5 rows)
```

But pages are registered neither in the visibility map nor in the freeze map, and the page header does not receive the visibility attribute (as it happens when the COPY command is executed with the FREEZE option): *p. 154*

```
=> SELECT * FROM pg_visibility_map('vac',0);
 all_visible | all_frozen
-------------+------------
 f           | f
(1 row)
=> SELECT flags & 4 > 0 all_visible
FROM page_header(get_raw_page('vac',0));
 all_visible
-------------
 f
(1 row)
```

The situation improves only after VACUUM is called (or autovacuum is triggered):

```
=> VACUUM vac;
=> SELECT * FROM pg_visibility_map('vac',0);
 all_visible | all_frozen
-------------+------------
 t           | t
(1 row)
=> SELECT flags & 4 > 0 AS all_visible
FROM page_header(get_raw_page('vac',0));
 all_visible
-------------
 t
(1 row)
```

It essentially means that even if all tuples in a page are beyond the database horizon, such a page will still have to be rewritten.

8.2 Other Rebuilding Methods

Alternatives to Full Vacuuming

In addition to VACUUM FULL, there are several other commands that can fully rebuild tables and indexes. All of them exclusively lock the table, all of them delete old data files and recreate them anew.

p. 369 The CLUSTER command is fully analogous to VACUUM FULL, but it also reorders tuples in files based on one of the available indexes. In some cases, it can help the planner p. 374 use index scans more efficiently. But you should bear in mind that clusterization is not supported: all further table updates will be breaking the physical order of tuples.

Programmatically, VACUUM FULL is simply a special instance of the CLUSTER command that does not require tuple reordering.[1]

The REINDEX command rebuilds one or more indexes.[2] In fact, VACUUM FULL and CLUSTER use this command under the hood when rebuilding indexes.

The TRUNCATE command[3] deletes all table rows; it is a logical equivalent of DELETE p. 81 run without the WHERE clause. But while DELETE simply marks heap tuples as deleted (so they still have to be vacuumed), TRUNCATE creates a new empty file, which is usually faster.

Reducing Downtime during Rebuilding

p. 232 VACUUM FULL is not meant to be run regularly, as it exclusively locks the table (even for queries) for the whole duration of its operation. This is usually not an option for highly available systems.

[1] backend/commands/cluster.c
[2] backend/commands/indexcmds.c
[3] backend/commands/tablecmds.c, ExecuteTruncate function

There are several extensions (such as pg_repack[1]) that can rebuild tables and indexes with almost zero downtime. An exclusive lock is still required, but only at the beginning and at the end of this process, and only for a short time. It is achieved by a more complex implementation: all the changes made on the original table while it is being rebuilt are saved by a trigger and then applied to the new table. To complete the operation, the utility replaces one table with the other in the system catalog.

An unconventional solution is offered by the pgcompacttable utility.[2] It performs multiple fake row updates (that do not change any data) so that current row versions gradually move towards the start of the file.

Between these update series, vacuuming removes outdated tuples and truncates the file little by little. This approach takes much more time and resources, but it requires no extra space for rebuilding the table and does not lead to load spikes. Short-time exclusive locks are still acquired while the table is being truncated, but vacuuming handles them rather smoothly. p. 126

8.3 Preventive Measures

Read-Only Queries

One of the reasons for file bloating is executing long-running transactions that hold the database horizon alongside intensive data updates. p. 101

As such, long-running (read-only) transactions do not cause any issues. So a common approach is to split the load between different systems: keep fast OLTP queries on the primary server and direct all OLAP transactions to a replica. Although it makes the solution more expensive and complicated, such measures may turn out to be indispensable.

In some cases, long transactions are the result of application or driver bugs rather than a necessity. If an issue cannot be resolved in a civilized way, the administrator can resort to the following two parameters:

[1] github.com/reorg/pg_repack
[2] github.com/dataegret/pgcompacttable

v. 9.6 • The *old_snapshot_threshold* parameter defines the maximum lifetime of a snapshot. Once this time is up, the server has the right to remove outdated tuples; if a long-running transaction still requires them, it will get an error ("snapshot too old").

v. 9.6 • The *idle_in_transaction_session_timeout* parameter limits the lifetime of an idle transaction. The transaction is aborted upon reaching this threshold.

Data Updates

Another reason for bloating is simultaneous modification of a large number of tuples. If all table rows get updated, the number of tuples can double, and vacuuming will not have enough time to interfere. Page pruning can reduce this problem, but not resolve it entirely.

Let's extend the output with another column to keep track of the processed rows:

```
=> ALTER TABLE vac ADD processed boolean DEFAULT false;
=> SELECT pg_size_pretty(pg_table_size('vac'));
 pg_size_pretty
----------------
 6936 kB
(1 row)
```

Once all the rows are updated, the table gets almost two times bigger:

```
=> UPDATE vac SET processed = true;
UPDATE 50000
=> SELECT pg_size_pretty(pg_table_size('vac'));
 pg_size_pretty
----------------
 14 MB
(1 row)
```

To address this situation, you can reduce the number of changes performed by a single transaction, spreading them out over time; then vacuuming can delete outdated tuples and free some space for new ones within the already existing pages. Assuming that each row update can be committed separately, we can use the following query that selects a batch of rows of the specified size as a template:

```
SELECT ID
FROM table
WHERE filtering the already processed rows
LIMIT batch size
FOR UPDATE SKIP LOCKED
```

This code snippet selects and immediately locks a set of rows that does not exceed the specified size. The rows that are already locked by other transactions are skipped: they will get into another batch next time. It is a rather flexible and convenient solution that allows you to easily change the batch size and restart the operation in case of a failure. Let's unset the processed attribute and perform full vacuuming to restore the original size of the table:

p. 255

```
=> UPDATE vac SET processed = false;
=> VACUUM FULL vac;
```

Once the first batch is updated, the table size grows a bit:

```
=> WITH batch AS (
  SELECT id FROM vac WHERE NOT processed LIMIT 1000
  FOR UPDATE SKIP LOCKED
)
UPDATE vac SET processed = true
WHERE id IN (SELECT id FROM batch);
UPDATE 1000
=> SELECT pg_size_pretty(pg_table_size('vac'));
 pg_size_pretty
----------------
 7064 kB
(1 row)
```

But from now on, the size remains almost the same because new tuples replace the removed ones:

```
=> VACUUM vac;
=> WITH batch AS (
  SELECT id FROM vac WHERE NOT processed LIMIT 1000
  FOR UPDATE SKIP LOCKED
)
UPDATE vac SET processed = true
WHERE id IN (SELECT id FROM batch);
UPDATE 1000
```

```
=> SELECT pg_size_pretty(pg_table_size('vac'));
 pg_size_pretty
----------------
 7072 kB
(1 row)
```

Part II

Buffer Cache and WAL

9

Buffer Cache

9.1 Caching

In modern computing systems, caching is omnipresent—both at the hardware and at the software level. The processor alone can have up to three or four levels of cache. RAID controllers and disks add their own cache too.

Caching is used to even out performance difference between fast and slow types of memory. Fast memory is expensive and has smaller volume, while slow memory is bigger and cheaper. Therefore, fast memory cannot accommodate *all* the data stored in slow memory. But in most cases only a *small* portion of data is being actively used at each particular moment, so allocating some fast memory for *cache* to keep hot data can significantly reduce the overhead incurred by slow memory access.

In PostgreSQL, buffer cache[1] holds relation pages, thus balancing access times to disks (milliseconds) and RAM (nanoseconds).

The operating system has its own cache that serves the same purpose. For this reason, database systems are usually designed to avoid double caching: the data stored on disk is usually queried directly, bypassing the OS cache. But PostgreSQL uses a different approach: it reads and writes all data via buffered file operations.

> Double caching can be avoided if you apply direct I/O. It will reduce the overhead, as PostgreSQL will use direct memory access (DMA) instead of copying buffered pages into the OS address space; besides, you will gain immediate control over physical writes on disk. However, direct I/O does not support data prefetching enabled by bufferization, so you have to implement it separately via asynchronous I/O, which requires massive code *p. 386*

[1] backend/storage/buffer/README

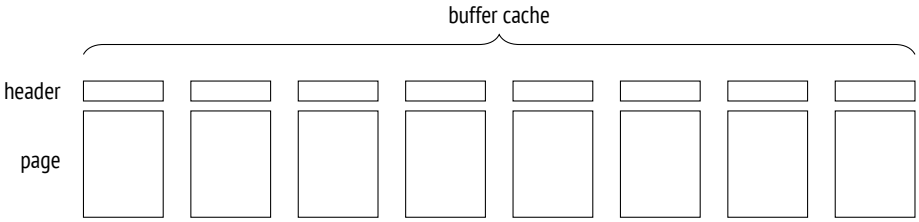
modifications in PostgreSQL core, as well as handling OS incompatibilities when it comes to direct and asynchronous I/O support. But once the asynchronous communication is set up, you can enjoy additional benefits of no-wait disk access.

The PostgreSQL community has already started this major effort,[1] but it will take a long time for the actual results to appear.

9.2 Buffer Cache Design

Buffer cache is located in the server's shared memory and is accessible to all the processes. It takes the major part of the shared memory and is surely one of the most important and complex data structures in PostgreSQL. Understanding how cache works is important in its own right, but even more so as many other structures (such as subtransactions, CLOG transaction status, and WAL entries) use a similar caching mechanism, albeit a simpler one.

The name of this cache is inspired by its inner structure, as it consists of an array of *buffers*. Each buffer reserves a memory chuck that can accommodate a single data page together with its header.[2]



A header contains some information about the buffer and the page in it, such as:

- physical location of the page (file ID, fork, and block number in the fork)
- the attribute showing that the data in the page has been modified and sooner or later has to be written back to disk (such a page is called *dirty*)
- buffer usage count
- pin count (or reference count)

[1] www.postgresql.org/message-id/flat/20210223100344.llw5an2aklengrmn%40alap3.anarazel.de
[2] include/storage/buf_internals.h

To get access to a relation's data page, a process requests it from the buffer manager[1] and receives the ID of the buffer that contains this page. Then it reads the cached data and modifies it right in the cache if needed. While the page is in use, its buffer is *pinned*. Pins forbid eviction of the cached page and can be applied together with other locks. Each pin increments the usage count as well. *p. 275*

As long as the page is cached, its usage does not incur any file operations.

We can explore the buffer cache using the pg_buffercache extension:

```
=> CREATE EXTENSION pg_buffercache;
```

Let's create a table and insert a row:

```
=> CREATE TABLE cacheme(
  id integer
) WITH (autovacuum_enabled = off);
=> INSERT INTO cacheme VALUES (1);
```

Now the buffer cache contains a heap page with the newly inserted row. You can see it for yourself by selecting all the buffers related to a particular table. We will need such a query again, so let's wrap it into a function:

```
=> CREATE FUNCTION buffercache(rel regclass)
RETURNS TABLE(
  bufferid integer, relfork text, relblk bigint,
  isdirty boolean, usagecount smallint, pins integer
) AS $$
SELECT bufferid,
  CASE relforknumber
    WHEN 0 THEN 'main'
    WHEN 1 THEN 'fsm'
    WHEN 2 THEN 'vm'
  END,
  relblocknumber,
  isdirty,
  usagecount,
  pinning_backends
FROM pg_buffercache
WHERE relfilenode = pg_relation_filenode(rel)
ORDER BY relforknumber, relblocknumber;
$$ LANGUAGE sql;
```

[1] backend/storage/buffer/bufmgr.c

```
=> SELECT * FROM buffercache('cacheme');
 bufferid | relfork | relblk | isdirty | usagecount | pins
----------+---------+--------+---------+------------+------
      268 | main    |      0 | t       |          1 |    0
(1 row)
```

The page is dirty: it has been modified, but is not written to disk yet. Its usage count is set to one.

9.3 Cache Hits

When the buffer manager has to read a page,[1] it first checks the buffer cache.

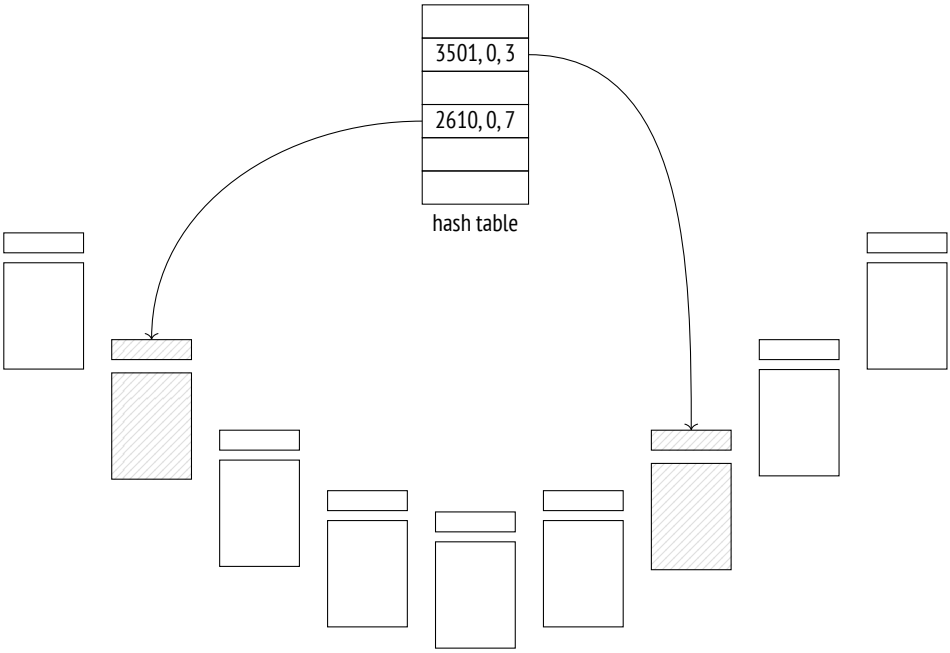
All buffer IDs are stored in a hash table,[2] which is used to speed up their search.

> Many modern programming languages include hash tables as one of the basic data types. Hash tables are often called associative arrays, and indeed, from the user's perspective they do look like an array; however, their index (a *hash key*) can be of any data type, for example, a text string rather than an integer.
>
> While the range of possible key values can be quite large, hash tables never contain that many different values at a time. The idea of hashing is to convert a key value into an integer number using a *hash function*. This number (or some of its bits) is used as an index of a regular array. The elements of this array are called *hash table buckets*.
>
> A good hash function distributes hash keys between buckets more or less uniformly, but it can still assign the same number to different keys, thus placing them into the same bucket; it is called a *collision*. For this reason, values are stored in buckets together with hash keys; to access a hashed value by its key, PostgreSQL has to check all the keys in the bucket.

There are multiple implementations of hash tables; of all the possible options, the buffer cache uses the extendible table that resolves hash collisions by chaining.[3]

A hash key consists of the ID of the relation file, the type of the fork, and the ID of the page within this fork's file. Thus, knowing the page, PostgreSQL can quickly find the buffer containing this page or make sure that the page is not currently cached.

[1] backend/storage/buffer/bufmgr.c, ReadBuffer_common function
[2] backend/storage/buffer/buf_table.c
[3] backend/utils/hash/dynahash.c



The buffer cache implementation has long been criticized for relying on a hash table: this structure is of no use when it comes to finding all the buffers taken by pages of a particular relation, which is required to remove pages from cache when running DROP and TRUNCATE commands or truncating a table during vacuuming.[1] Yet no one has suggested an adequate alternative so far.

If the hash table contains the required buffer ID, the buffer manager pins this buffer and returns its ID to the process. Then this process can start using the cached page without incurring any I/O traffic.

To pin a buffer, PostgreSQL has to increment the pin counter in its header; a buffer can be pinned by several processes at a time. While its pin counter is greater than zero, the buffer is assumed to be in use, and no radical changes in its contents are allowed. For example, a new tuple can appear (it will be invisible following the visibility rules), but the page itself cannot be replaced.

When run with the analyze and buffers options, the EXPLAIN command executes the displayed query plan and shows the number of used buffers:

[1] backend/storage/buffer/bufmgr.c, DropRelFileNodeBuffers function

```
=> EXPLAIN (analyze, buffers, costs off, timing off, summary off)
  SELECT * FROM cacheme;
                 QUERY PLAN
---------------------------------------------
 Seq Scan on cacheme (actual rows=1 loops=1)
   Buffers: shared hit=1
 Planning:
   Buffers: shared hit=12 read=7
(4 rows)
```

Here hit=1 means that the only page that had to be read was found in the cache.

Buffer pinning increases the usage count by one:

```
=> SELECT * FROM buffercache('cacheme');
 bufferid | relfork | relblk | isdirty | usagecount | pins
----------+---------+--------+---------+------------+------
      268 | main    |      0 | t       |          2 |    0
(1 row)
```

To observe pinning in action during query execution, let's open a cursor—it will hold the buffer pin, as it has to provide quick access to the next row in the result set:

```
=> BEGIN;
=> DECLARE c CURSOR FOR SELECT * FROM cacheme;
=> FETCH c;
 id
----
  1
(1 row)
=> SELECT * FROM buffercache('cacheme');
 bufferid | relfork | relblk | isdirty | usagecount | pins
----------+---------+--------+---------+------------+------
      268 | main    |      0 | t       |          3 |    1
(1 row)
```

If a process cannot use a pinned buffer, it usually skips it and simply chooses another one. We can see it during table vacuuming:

```
=> VACUUM VERBOSE cacheme;
INFO:  vacuuming "public.cacheme"
INFO:  table "cacheme": found 0 removable, 0 nonremovable row
versions in 1 out of 1 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin:
878
Skipped 1 page due to buffer pins, 0 frozen pages.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

The page was skipped because its tuples could not be physically removed from the pinned buffer.

But if it is exactly this buffer that is required, the process joins the queue and waits for exclusive access to this buffer. An example of such an operation is vacuuming with freezing.[1] p. 143

Once the cursor closes or moves on to another page, the buffer gets unpinned. In this example, it happens at the end of the transaction:

```
=> COMMIT;
=> SELECT * FROM buffercache('cacheme');
 bufferid | relfork | relblk | isdirty | usagecount | pins
----------+---------+--------+---------+------------+------
      268 | main    |      0 | t       |          3 |    0
      310 | vm      |      0 | f       |          2 |    0
(2 rows)
```

Page modifications are protected by the same pinning mechanism. For example, let's insert another row into the table (it will get into the same page):

```
=> INSERT INTO cacheme VALUES (2);
=> SELECT * FROM buffercache('cacheme');
 bufferid | relfork | relblk | isdirty | usagecount | pins
----------+---------+--------+---------+------------+------
      268 | main    |      0 | t       |          4 |    0
      310 | vm      |      0 | f       |          2 |    0
(2 rows)
```

[1] backend/storage/buffer/bufmgr.c, LockBufferForCleanup function

PostgreSQL does not perform any immediate writes to disk: a page remains dirty in the buffer cache for a while, providing some performance gains for both reads and writes.

9.4 Cache Misses

If the hash table has no entry related to the queried page, it means that this page is not cached. In this case, a new buffer is assigned (and immediately pinned), the page is read into this buffer, and the hash table references are modified accordingly.

Let's restart the instance to clear its buffer cache:

```
postgres$ pg_ctl restart -l /home/postgres/logfile
```

An attempt to read a page will result in a cache miss, and the page will be loaded into a new buffer:

```
=> EXPLAIN (analyze, buffers, costs off, timing off, summary off)
  SELECT * FROM cacheme;
                 QUERY PLAN
---------------------------------------------
 Seq Scan on cacheme (actual rows=2 loops=1)
   Buffers: shared read=1 dirtied=1
 Planning:
   Buffers: shared hit=15 read=7
(4 rows)
```

Instead of hit, the plan now shows the read status, which denotes a cache miss. Besides, this page has become dirty, as the query has modified some hint bits.

p. 80

A buffer cache query shows that the usage count for the newly added page is set to one:

```
=> SELECT * FROM buffercache('cacheme');
 bufferid | relfork | relblk | isdirty | usagecount | pins
----------+---------+--------+---------+------------+------
       98 | main    |      0 | t       |          1 |    0
(1 row)
```

The pg_statio_all_tables view contains the complete statistics on buffer cache usage by tables:

```
=> SELECT heap_blks_read, heap_blks_hit
FROM pg_statio_all_tables
WHERE relname = 'cacheme';
 heap_blks_read | heap_blks_hit
----------------+---------------
              2 |             5
(1 row)
```

PostgreSQL provides similar views for indexes and sequences. They can also display statistics on I/O operations, but only if *track_io_timing* is enabled. off

Buffer Search and Eviction

Choosing a buffer for a page is not so trivial.[1] There are two possible scenarios:

1. Right after the server start all the buffers are empty and are bound into a list.

   While some buffers are still free, the next page read from disk will occupy the first buffer, and it will be removed from the list.

   A buffer can return to the list[2] only if its page disappears, without being replaced by another page. It can happen if you call DROP or TRUNCATE commands, or if the table is truncated during vacuuming.

2. Sooner or later no free buffers will be left (since the size of the database is usually bigger than the memory chunk allocated for cache). Then the buffer manager will have to select one of the buffers that is already in use and evict the cached page from this buffer. It is performed using the clock sweep algorithm, which is well illustrated by the clock metaphor. Pointing to one of the buffers, the clock hand starts going around the buffer cache and reduces the usage count for each cached page by one as it passes. The first unpinned buffer with the zero count found by the clock hand will be cleared.

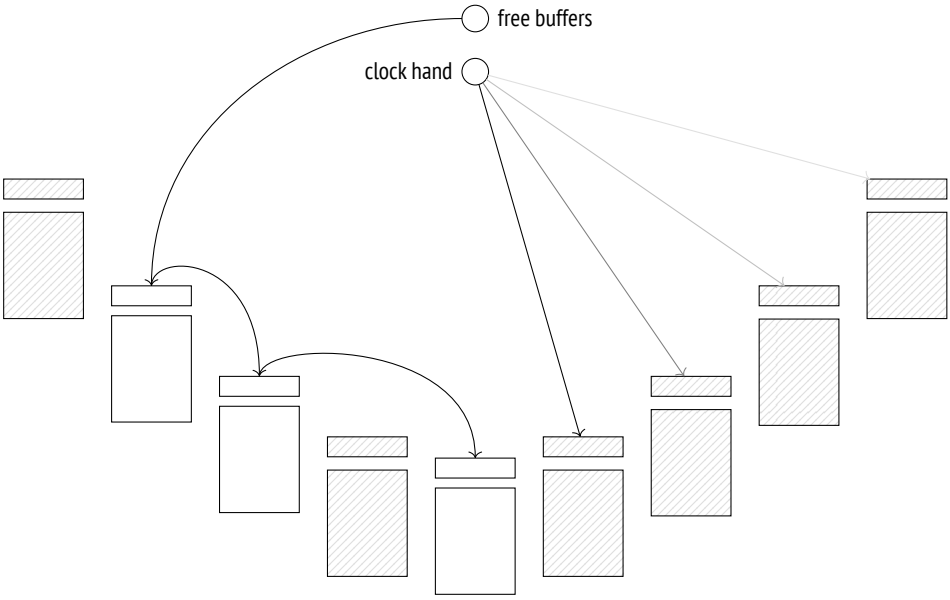
[1] backend/storage/buffer/freelist.c, StrategyGetBuffer function
[2] backend/storage/buffer/freelist.c, StrategyFreeBuffer function

Thus, the usage count is incremented each time the buffer is accessed (that is, pinned), and reduced when the buffer manager is searching for pages to evict. As a result, the least recently used pages are evicted first, while those that have been accessed more often will remain in the cache longer.

As you can guess, if all the buffers have a non-zero usage count, the clock hand has to complete more than one full circle before any of them finally reaches the zero value. To avoid running multiple laps, PostgreSQL limits the usage count by 5.

Once the buffer to evict is found, the reference to the page that is still in this buffer must be removed from the hash table.

p. 202 But if this buffer is dirty, that is, it contains some modified data, the old page cannot be simply thrown away—the buffer manager has to write it to disk first.



Then the buffer manager reads a new page into the found buffer—no matter if it had to be cleared or was still free. It uses buffered I/O for this purpose, so the page will be read from disk only if the operating system cannot find it in its own cache.

Those database systems that use direct I/O and do not depend on the OS cache differentiate between logical reads (from RAM, that is, from the buffer cache) and physical reads (from disk). From the standpoint of PostgreSQL, a page can be either read from the buffer cache or requested from the operating system, but there is no way to tell whether it was found in RAM or read from disk in the latter case.

The hash table is updated to refer to the new page, and the buffer gets pinned. Its usage count is incremented and is now set to one, which gives this buffer some time to increase this value while the clock hand is traversing the buffer cache.

9.5 Bulk Eviction

If bulk reads or writes are performed, there is a risk that one-time data can quickly oust useful pages from the buffer cache.

As a precaution, bulk operations use rather small *buffer rings*, and eviction is performed within their boundaries, without affecting other buffers.

Alongside the "buffer ring," the code also uses the term "ring buffer". However, this synonym is rather ambiguous because the ring buffer itself consists of several buffers (that belong to the buffer cache). The term "buffer ring" is more accurate in this respect.

A buffer ring of a particular size consists of an array of buffers that are used one after another. At first, the buffer ring is empty, and individual buffers join it one by one, after being selected from the buffer cache in the usual manner. Then eviction comes into play, but only within the ring limits.[1]

Buffers added into a ring are not excluded from the buffer cache and can still be used by other operations. So if the buffer to be reused turns out to be pinned, or its usage count is higher than one, it will be simply detached from the ring and replaced by another buffer.

PostgreSQL supports three eviction strategies.

Bulk reads strategy is used for sequential scans of large tables if their size exceeds $\frac{1}{4}$ of the buffer cache. The ring buffer takes 256 kB (32 standard pages). *p. 335*

[1] backend/storage/buffer/freelist.c, GetBufferFromRing function

This strategy does not allow writing dirty pages to disk to free a buffer; instead, the buffer is excluded from the ring and replaced by another one. As a result, reading does not have to wait for writing to complete, so it is performed faster.

If it turns out that the table is already being scanned, the process that starts another scan joins the existing buffer ring and gets access to the currently available data, without incurring extra I/O operations.[1] When the first process completes the scan, the second one gets back to the skipped part of the table.

Bulk writes strategy is applied by COPY FROM, CREATE TABLE AS SELECT, and CREATE MATERIALIZED VIEW commands, as well as by those ALTER TABLE flavors that cause table rewrites. The allocated ring is quite big, its default size being 16 MB (2048 standard pages), but it never exceeds $\frac{1}{8}$ of the total size of the buffer cache.

Vacuuming strategy is used by the process of vacuuming when it performs a full table scan without taking the visibility map into account. The ring buffer is assigned 256 kB of RAM (32 standard pages).

Buffer rings do not always prevent undesired eviction. If UPDATE or DELETE commands affect a lot of rows, the performed table scan applies the bulk reads strategy, but since the pages are constantly being modified, buffer rings virtually become useless.

p. 30 Another example worth mentioning is storing oversized data in TOAST tables. In spite of a potentially large volume of data that has to be read, toasted values are always accessed via an index, so they bypass buffer rings.

Let's take a closer look at the bulk reads strategy. For simplicity, we will create a table in such a way that an inserted row takes the whole page. By default, the buffer cache size is 16,384 pages, 8 kB each. So the table must take more than 4096 pages for the scan to use a buffer ring.

```
=> CREATE TABLE big(
  id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
  s char(1000)
) WITH (fillfactor = 10);
```

[1] backend/access/common/syncscan.c

```
=> INSERT INTO big(s)
  SELECT 'FOO' FROM generate_series(1,4096+1);
```

Let's analyze the table:

```
=> ANALYZE big;
=> SELECT relname, relfilenode, relpages
FROM pg_class
WHERE relname IN ('big', 'big_pkey');
 relname  | relfilenode | relpages
----------+-------------+----------
 big      |       16546 |     4097
 big_pkey |       16551 |       14
(2 rows)
```

Restart the server to clear the cache, as now it contains some heap pages that have been read during analysis.

```
postgres$ pg_ctl restart -l /home/postgres/logfile
```

Once the server is restarted, let's read the whole table:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT id FROM big;
                 QUERY PLAN
--------------------------------------------
 Seq Scan on big (actual rows=4097 loops=1)
(1 row)
```

Heap pages take only 32 buffers, which make up the buffer ring for this operation:

```
=> SELECT count(*)
FROM pg_buffercache
WHERE relfilenode = pg_relation_filenode('big'::regclass);
 count
-------
    32
(1 row)
```

But in the case of an index scan the buffer ring is not used:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM big ORDER BY id;
                          QUERY PLAN
-----------------------------------------------------------------
 Index Scan using big_pkey on big (actual rows=4097 loops=1)
(1 row)
```

As a result, the buffer cache ends up containing the whole table and the whole index:

```
=> SELECT relfilenode, count(*)
FROM pg_buffercache
WHERE relfilenode IN (
  pg_relation_filenode('big'),
  pg_relation_filenode('big_pkey')
)
GROUP BY relfilenode;
 relfilenode | count
-------------+-------
       16546 |  4097
       16551 |    14
(2 rows)
```

9.6 Choosing the Buffer Cache Size

128MB The size of the buffer cache is defined by the *shared_buffers* parameter. Its default value is known to be low, so it makes sense to increase it right after the PostgreSQL installation. You will have to reload the server in this case because shared memory is allocated for cache at the server start.

But how can we determine an appropriate value?

Even a very large database has a limited set of hot data that is being used simultaneously. In the perfect world, it is this set that must fit the buffer cache (with some space being reserved for one-time data). If the cache size is smaller, the actively used pages will be evicting each other all the time, thus leading to excessive I/O operations. But thoughtless increase of the cache size is not a good idea either: RAM is a scarce resource, and besides, larger cache incurs higher maintenance costs.

The optimal buffer cache size differs from system to system: it depends on things like the total size of the available memory, data profiles, and workload types. Unfortunately, there is no magic value or formula to suit everyone equally well.

You should also keep in mind that a cache miss in PostgreSQL does not necessarily trigger a physical I/O operation. If the buffer cache is quite small, the OS cache uses the remaining free memory and can smooth things out to some extent. But unlike the database, the operating system knows nothing about the read data, so it applies a different eviction strategy.

A typical recommendation is to start with $\frac{1}{4}$ of RAM and then adjust this setting as required.

The best approach is experimentation: you can increase or decrease the cache size and compare the system performance. Naturally, it requires having a test system that is fully analogous to the production one, and you must be able to reproduce typical workloads.

You can also run some analysis using the pg_buffercache extension. For example, explore buffer distribution depending on their usage:

```
=> SELECT usagecount, count(*)
FROM pg_buffercache
GROUP BY usagecount
ORDER BY usagecount;
 usagecount | count
------------+-------
          1 |  4128
          2 |    50
          3 |     4
          4 |     4
          5 |    73
            | 12125
(6 rows)
```

NULL usage count values correspond to free buffers. They are quite expected in this case because the server was restarted and remained idle most of the time. The majority of the used buffers contain pages of the system catalog tables read by the backend to fill its system catalog cache and to perform queries.

We can check what fraction of each relation is cached, and whether this data is hot (a page is considered hot here if its usage count is bigger than one):

```
=> SELECT c.relname,
  count(*) blocks,
  round( 100.0 * 8192 * count(*) /
    pg_table_size(c.oid) ) AS "% of rel",
  round( 100.0 * 8192 * count(*) FILTER (WHERE b.usagecount > 1) /
    pg_table_size(c.oid) ) AS "% hot"
FROM pg_buffercache b
  JOIN pg_class c ON pg_relation_filenode(c.oid) = b.relfilenode
WHERE b.reldatabase IN (
  0, -- cluster-wide objects
  (SELECT oid FROM pg_database WHERE datname = current_database())
)
AND b.usagecount IS NOT NULL
GROUP BY c.relname, c.oid
ORDER BY 2 DESC
LIMIT 10;
             relname              | blocks | % of rel | % hot
----------------------------------+--------+----------+-------
 big                              |   4097 |      100 |     1
 pg_attribute                     |     30 |       48 |    47
 big_pkey                         |     14 |      100 |     0
 pg_proc                          |     13 |       12 |     6
 pg_operator                      |     11 |       61 |    50
 pg_class                         |     10 |       59 |    59
 pg_proc_oid_index                |      9 |       82 |    45
 pg_attribute_relid_attnum_index  |      8 |       73 |    64
 pg_proc_proname_args_nsp_index   |      6 |       18 |     6
 pg_amproc                        |      5 |       56 |    56
(10 rows)
```

This example shows that the big table and its index are fully cached, but their pages are not being actively used.

Analyzing data from different angles, you can gain some useful insights. However, make sure to follow these simple rules when running pg_buffercache queries:

- Repeat such queries several times since the returned figures will vary to some extent.
- Do not run such queries non-stop because the pg_buffercache extension locks the viewed buffers, even if only briefly.

9.7 Cache Warming

After a server restart, the cache requires some time to warm up, that is, to accumulate the actively used data. It may be helpful to cache certain tables right away, and the pg_prewarm extension serves exactly this purpose:

```
=> CREATE EXTENSION pg_prewarm;
```

Apart from loading tables into the buffer cache (or into the OS cache only), this extension can write the current cache state to disk and then restore it after the server restart. To enable this functionality, you have to add this extension's library to *shared_preload_libraries* and restart the server: v. 11

```
=> ALTER SYSTEM SET shared_preload_libraries = 'pg_prewarm';

postgres$ pg_ctl restart -l /home/postgres/logfile
```

If the *pg_prewarm.autoprewarm* setting has not changed, a process called autoprewarm leader will be started automatically after the server is reloaded; once in *pg_prewarm.autoprewarm_interval* seconds, this process will flush the list of cached pages to disk (using one of the *max_parallel_processes* slots). on 300s

```
postgres$ ps -o pid,command \
--ppid `head -n 1 /usr/local/pgsql/data/postmaster.pid` | \
grep prewarm
  23279 postgres: autoprewarm leader
```

Now that the server has been restarted, the big table is not cached anymore:

```
=> SELECT count(*)
FROM pg_buffercache
WHERE relfilenode = pg_relation_filenode('big'::regclass);
 count
-------
     0
(1 row)
```

If you have well-grounded assumptions that the whole table is going to be actively used and disk access will make response times unacceptably high, you can load this table into the buffer cache in advance:

```
=> SELECT pg_prewarm('big');
 pg_prewarm
------------
       4097
(1 row)
=> SELECT count(*)
FROM pg_buffercache
WHERE relfilenode = pg_relation_filenode('big'::regclass);
 count
-------
  4097
(1 row)
```

The list of pages is dumped into the PGDATA/autoprewarm.blocks file. You can wait until the autoprewarm leader completes for the first time, but we will initiate the dump manually:

```
=> SELECT autoprewarm_dump_now();
 autoprewarm_dump_now
----------------------
                 4224
(1 row)
```

The number of flushed pages is bigger than 4097 because all the used buffers are taken into account. The file is written in a text format; it contains the IDs of the database, tablespace, and file, as well as the fork and segment numbers:

```
postgres$ head -n 10 /usr/local/pgsql/data/autoprewarm.blocks
<<4224>>
0,1664,1262,0,0
0,1664,1260,0,0
16391,1663,1259,0,0
16391,1663,1259,0,1
16391,1663,1259,0,2
16391,1663,1259,0,3
16391,1663,1249,0,0
16391,1663,1249,0,1
16391,1663,1249,0,2
```

Let's restart the server again.

```
postgres$ pg_ctl restart -l /home/postgres/logfile
```

The table appears in the cache right away:

```
=> SELECT count(*)
FROM pg_buffercache
WHERE relfilenode = pg_relation_filenode('big'::regclass);
 count
-------
  4097
(1 row)
```

It is again the autoprewarm leader that does all the preliminary work: it reads the file, sorts the pages by databases, reorders them (so that disk reads happen sequentially if possible), and then passes them to the autoprewarm worker for processing.

9.8 Local Cache

Temporary tables do not follow the workflow described above. Since temporary data is visible to a single process only, there is no point in loading it into the shared buffer cache. Therefore, temporary data uses the local cache of the process that owns the table.[1]

In general, local buffer cache works similar to the shared one:

- Page search is performed via a hash table.
- Eviction follows the standard algorithm (except that buffer rings are not used).
- Pages can be pinned to avoid eviction.

However, local cache implementation is much simpler because it has to handle neither locks on memory structures (buffers can be accessed by a single process only) nor fault tolerance (temporary data exists till the end of the session at the most).

p. 275
p. 189

[1] backend/storage/buffer/localbuf.c

Since only few sessions typically use temporary tables, local cache memory is assigned on demand. The maximum size of the local cache available to a session is limited by the *temp_buffers* parameter.

8MB

> Despite a similar name, the *temp_file_limit* parameter has nothing to do with temporary tables; it is related to files that may be created during query execution to temporarily store intermediate data.

In the EXPLAIN command output, all calls to the local buffer cache are tagged as local instead of shared:

```
=> CREATE TEMPORARY TABLE tmp AS SELECT 1;
=> EXPLAIN (analyze, buffers, costs off, timing off, summary off)
  SELECT * FROM tmp;
                QUERY PLAN
----------------------------------------
 Seq Scan on tmp (actual rows=1 loops=1)
   Buffers: local hit=1
 Planning:
   Buffers: shared hit=12 read=7
(4 rows)
```

10

Write-Ahead Log

10.1 Logging

In case of a failure, such as a power outage, an OS error, or a database server crash, all the contents of RAM will be lost; only the data written to disk will persist. To start the server after a failure, you have to restore data consistency. If the disk itself has been damaged, the same issue has to be resolved by backup recovery.

In theory, you could maintain data consistency on disk at all times. But in practice it means that the server has to constantly write random pages to disk (even though sequential writing is cheaper), and the order of such writes must guarantee that consistency is not compromised at any particular moment (which is hard to achieve, especially if you deal with complex index structures).

Just like the majority of database systems, PostgreSQL uses a different approach.

While the server is running, some of the current data is available only in RAM, its writing to permanent storage being deferred. Therefore, the data stored on disk is always inconsistent during server operation, as pages are never flushed all at once. But each change that happens in RAM (such as a page update performed in the buffer cache) is *logged*: PostgreSQL creates a log entry that contains all the essential information required to repeat this operation if the need arises.[1]

A log entry related to a page modification must be written to disk *ahead* of the modified page itself. Hence the name of the log: *write-ahead log*, or WAL. This requirement guarantees that in case of a failure PostgreSQL can read WAL entries from disk and *replay* them to repeat the already completed operations whose results were still in RAM and did not make it to disk before the crash.

[1] postgresql.org/docs/14/wal-intro.html

Keeping a write-ahead log is usually more efficient than writing random pages to disk. WAL entries constitute a continuous stream of data, which can be handled even by HDDs. Besides, WAL entries are often smaller than the page size.

It is required to log all operations that can potentially break data consistency in case of a failure. In particular, the following actions are recorded in WAL:

- page modifications performed in the buffer cache—since writes are deferred
- transaction commits and rollbacks—since the status change happens in CLOG buffers and does not make it to disk right away
- file operations (like creation and deletion of files and directories when tables get added or removed)—since such operations must be in sync with data changes

The following actions are not logged:

- operations on UNLOGGED tables
- operations on temporary tables—since their lifetime is anyway limited by the session that spawns them

  Prior to PostgreSQL 10, hash indexes were not logged either. Their only purpose was to match hash functions to different data types.

Apart from crash recovery, WAL can also be used for point-in-time recovery from a backup and replication.

10.2 WAL Structure

Logical Structure

Speaking about its logical structure, we can describe WAL[1] as a stream of log entries of variable length. Each entry contains some *data* about a particular operation

[1] postgresql.org/docs/14/wal-internals.html
backend/access/transam/README

preceded by a standard *header*.[1] Among other things, the header provides the following information:

- transaction ID related to the entry
- the resource manager that interprets the entry[2]
- the checksum to detect data corruption
- entry length
- a reference to the previous WAL entry

  WAL is usually read in the forward direction, but some utilities like pg_rewind may scan it backwards.

WAL data itself can have different formats and meaning. For example, it can be a page fragment that has to replace some part of the page at the specified offset. The corresponding resource manager must know how to interpret and replay a particular entry. There are separate managers for tables, various index types, transaction status, and other entities.

WAL files take up special buffers in the server's shared memory. The size of the cache used by WAL is defined by the *wal_buffers* parameter. By default, this size is -1
chosen automatically as $\frac{1}{32}$ of the total buffer cache size.

WAL cache is quite similar to buffer cache, but it usually operates in the ring buffer mode: new entries are added to its head, while older entries are saved to disk starting at the tail. If WAL cache is too small, disk synchronization will be performed more often than necessary.

Under low load, the insert position (the buffer's head) is almost always the same as the position of the entries that have already been saved to disk (the buffer's tail):

```
=> SELECT pg_current_wal_lsn(), pg_current_wal_insert_lsn();
 pg_current_wal_lsn | pg_current_wal_insert_lsn
--------------------+---------------------------
 0/3E72A000         | 0/3E72B2F0
(1 row)
```

[1] include/access/xlogrecord.h
[2] include/access/rmgrlist.h

Prior to PostgreSQL 10, all function names contained the XLOG acronym instead of WAL.

To refer to a particular entry, PostgreSQL uses a special data type: pg_lsn (log sequence number, LSN). It represents a 64-bit offset in bytes from the start of the WAL to an entry. An LSN is displayed as two 32-bit numbers in the hexadecimal notation separated by a slash.

Let's create a table:

```
=> CREATE TABLE wal(id integer);
=> INSERT INTO wal VALUES (1);
```

Start a transaction and note the LSN of the WAL insert position:

```
=> BEGIN;
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 0/3E744260
(1 row)
```

Now run some arbitrary command, for example, update a row:

```
=> UPDATE wal SET id = id + 1;
```

The page modification is performed in the buffer cache in RAM. This change is logged in a WAL page, also in RAM. As a result, the insert LSN is advanced:

```
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 0/3E7442A8
(1 row)
```

To ensure that the modified data page is flushed to disk strictly after the corresponding WAL entry, the page header stores the LSN of the latest WAL entry related to this page. You can view this LSN using pageinspect:

```
=> SELECT lsn FROM page_header(get_raw_page('wal',0));
    lsn
------------
 0/3E7442A8
(1 row)
```

There is only one WAL for the whole database cluster, and new entries constantly get appended to it. For this reason, the LSN stored in the page may turn out to be smaller than the one returned by the pg_current_wal_insert_lsn function some time ago. But if nothing has happened in the system, these numbers will be the same.

Now let's commit the transaction:

```
=> COMMIT;
```

The commit operation is also logged, and the insert LSN changes again:

```
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 0/3E7442D0
(1 row)
```

A commit updates transaction status in CLOG pages, which are kept in their own cache.[1] The CLOG cache usually takes 128 pages in the shared memory.[2] To make sure that a CLOG page is not flushed to disk before the corresponding WAL entry, the LSN of the latest WAL entry has to be tracked for CLOG pages too. But this information is stored in RAM, not in the page itself. p. 79

At some point WAL entries will make it to disk; then it will be possible to evict CLOG and data pages from the cache. If they had to be evicted earlier, it would have been discovered, and WAL entries would have been forced to disk first.[3] p. 210

If you know two LSN positions, you can calculate the size of WAL entries between them (in bytes) by simply subtracting one position from the other. You just have to cast them to the pg_lsn type:

```
=> SELECT '0/3E7442D0'::pg_lsn - '0/3E744260'::pg_lsn;
 ?column?
----------
      112
(1 row)
```

[1] backend/access/transam/slru.c
[2] backend/access/transam/clog.c, CLOGShmemBuffers function
[3] backend/storage/buffer/bufmgr.c, FlushBuffer function

In this particular case, WAL entries related to UPDATE and COMMIT operations took about a hundred of bytes.

You can use the same approach to estimate the volume of WAL entries generated by a particular workload per unit of time. This information will be required for the checkpoint setup.

Physical Structure

On disk, the WAL is stored in the PGDATA/pg_wal directory as separate files, or seg-
16MB ments. Their size is shown by the read-only *wal_segment_size* parameter.

v. 11 For high-load systems, it makes sense to increase the segment size since it may reduce the overhead, but this setting can be modified only during cluster initialization (initdb --wal-segsize).

WAL entries get into the current file until it runs out of space; then PostgreSQL starts a new file.

We can learn in which file a particular entry is located, and at what offset from the start of the file:

```
=> SELECT file_name, upper(to_hex(file_offset)) file_offset
FROM pg_walfile_name_offset('0/3E744260');
        file_name         | file_offset
--------------------------+-------------
 000000010000000000000003E | 744260
```

timeline — log sequence number

The name of the file consists of two parts. The highest eight hexadecimal digits define the *timeline* used for recovery from a backup, while the rest represent the highest LSN bits (the lowest LSN bits are shown in the file_offset field).

v. 10 To view the current WAL files, you can call the following function:

```
=> SELECT *
FROM pg_ls_waldir()
WHERE name = '000000010000000000000003E';
```

```
           name           |   size   |      modification
--------------------------+----------+------------------------
 000000010000000000000003E | 16777216 | 2022-11-25 22:58:50+03
(1 row)
```

Now let's take a look at the headers of the newly created WAL entries using the pg_waldump utility, which can filter WAL entries both by the LSN range (like in this example) and by a particular transaction ID.

The pg_waldump utility should be started on behalf of the postgres OS user, as it needs access to WAL files on disk.

```
postgres$ /usr/local/pgsql/bin/pg_waldump \
-p /usr/local/pgsql/data/pg_wal -s 0/3E744260 -e 0/3E7442D0#
rmgr: Heap        len (rec/tot):     69/    69, tx:        887, lsn:
0/3E744260, prev 0/3E744238, desc: HOT_UPDATE off 1 xmax 887 flags
0x40 ; new off 2 xmax 0, blkref #0: rel 1663/16391/16562 blk 0
rmgr: Transaction len (rec/tot):     34/    34, tx:        887, lsn:
0/3E7442A8, prev 0/3E744260, desc: COMMIT 2022-11-25 22:58:50.041828
MSK
```

Here we can see the headers of two entries.

The first one is the HOT_UPDATE operation handled by the Heap resource manager. p. 110
The blkref field shows the filename and the page ID of the updated heap page:

```
=> SELECT pg_relation_filepath('wal');
 pg_relation_filepath
----------------------
 base/16391/16562
(1 row)
```

The second entry is the COMMIT operation supervised by the Transaction resource manager.

10.3 Checkpoint

To restore data consistency after a failure (that is, to perform recovery), PostgreSQL has to replay the WAL in the forward direction and apply the entries that represent lost changes to the corresponding pages. To find out what has been lost, the LSN

of the page stored on disk is compared to the LSN of the WAL entry. But at which point should we start the recovery? If we start too late, the pages written to disk before this point will fail to receive all the changes, which will lead to irreversible data corruption. Starting from the very beginning is unrealistic: it is impossible to store such a potentially huge volume of data, and neither is it possible to accept such a long recovery time. We need a *checkpoint* that is gradually moving forward, thus making it safe to start the recovery from this point and remove all the previous WAL entries.

The most straightforward way to create a checkpoint is to periodically suspend all system operations and force all dirty pages to disk. This approach is of course unacceptable, as the system will hang for an indefinite but quite significant time.

For this reason, the checkpoint is spread out over time, virtually constituting an interval. Checkpoint execution is performed by a special background process called checkpointer.[1]

Checkpoint start. The checkpointer process flushes to disk everything that can be written instantaneously: CLOG transaction status, subtransactions' metadata, and a few other structures.

Checkpoint execution. Most of the checkpoint execution time is spent on flushing dirty pages to disk.[2]

First, a special tag is set in the headers of all the buffers that were dirty at the checkpoint start. It happens very fast since no I/O operations are involved.

Then checkpointer traverses all the buffers and writes the tagged ones to disk. Their pages are not evicted from the cache: they are simply written down, so usage and pin counts can be ignored.

v. 9.6 Pages are processed in the order of their IDs to avoid random writing if possible. For better load balancing, PostgreSQL alternates between different tablespaces (as they may be located on different physical devices).

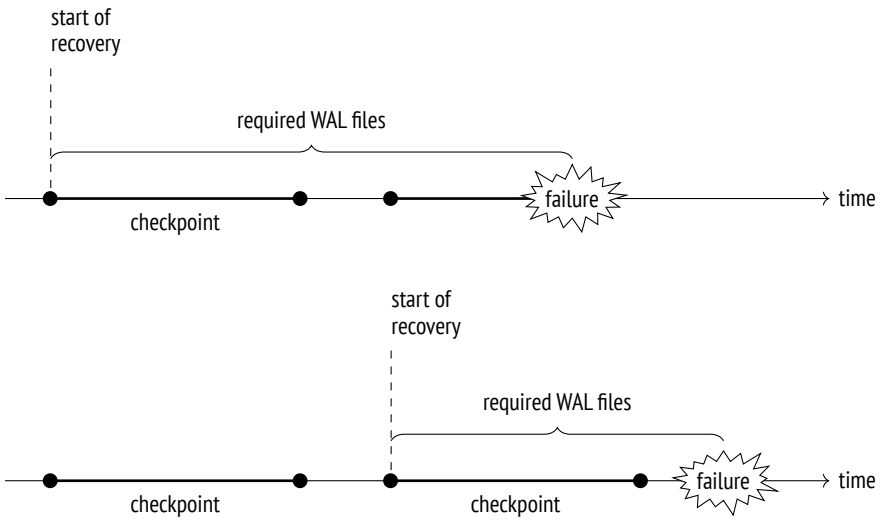
[1] backend/postmaster/checkpointer.c
backend/access/transam/xlog.c, CreateCheckPoint function

[2] backend/storage/buffer/bufmgr.c, BufferSync function

Backends can also write tagged buffers to disk—if they get to them first. In any case, buffer tags are removed at this stage, so for the purpose of the checkpoint each buffer will be written only once.

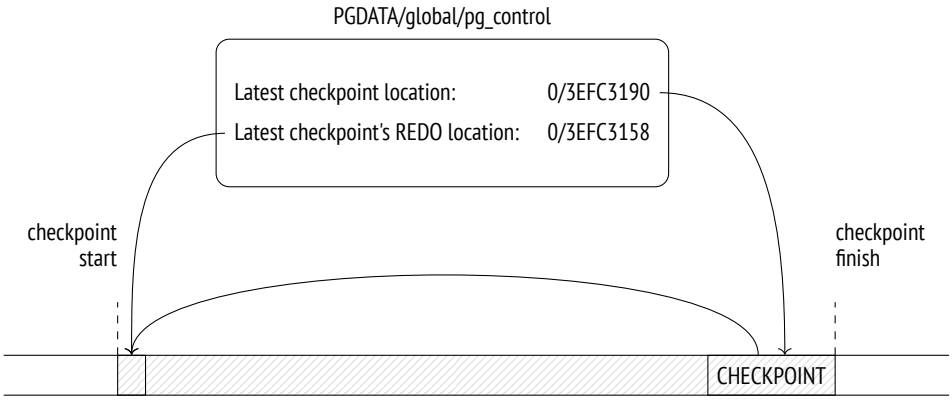
Naturally, pages can still be modified in the buffer cache while the checkpoint is in progress. But since new dirty buffers are not tagged, checkpointer will ignore them.

Checkpoint completion. When all the buffers that were dirty *at the start* of the checkpoint are written to disk, the checkpoint is considered *complete*. From now on (but not earlier!), the *start* of the checkpoint will be used as a new starting point of recovery. All the WAL entries written before this point are not required anymore.



Finally, checkpointer creates a WAL entry that corresponds to the checkpoint completion, specifying the checkpoint's start LSN. Since the checkpoint logs nothing when it starts, this LSN can belong to a WAL entry of any type.

The PGDATA/global/pg_control file also gets updated to refer to the latest completed checkpoint. (Until this process is over, pg_control keeps the previous checkpoint.)



To figure out once and for all what points where, let's take a look at a simple example. We will make several cached pages dirty:

```
=> UPDATE big SET s = 'FOO';
=> SELECT count(*) FROM pg_buffercache WHERE isdirty;
 count
-------
  4119
(1 row)
```

Note the current WAL position:

```
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 0/3EFC3158
(1 row)
```

Now let's complete the checkpoint manually. All the dirty pages will be flushed to disk; since nothing happens in the system, new dirty pages will not appear:

```
=> CHECKPOINT;
=> SELECT count(*) FROM pg_buffercache WHERE isdirty;
 count
-------
     0
(1 row)
```

Let's see how the checkpoint is reflected in the WAL:

```
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 0/3EFC3208
(1 row)

postgres$ /usr/local/pgsql/bin/pg_waldump \
-p /usr/local/pgsql/data/pg_wal -s 0/3EFC3158 -e 0/3EFC3208
rmgr: Standby     len (rec/tot):     50/    50, tx:          0, lsn:
0/3EFC3158, prev 0/3EFC3130, desc: RUNNING_XACTS nextXid 889
latestCompletedXid 888 oldestRunningXid 889
rmgr: XLOG        len (rec/tot):    114/   114, tx:          0, lsn:
0/3EFC3190, prev 0/3EFC3158, desc: CHECKPOINT_ONLINE redo
0/3EFC3158; tli 1; prev tli 1; fpw true; xid 0:889; oid 24754; multi
1; offset 0; oldest xid 726 in DB 1; oldest multi 1 in DB 1;
oldest/newest commit timestamp xid: 0/0; oldest running xid 889;
online
```

The latest WAL entry is related to the checkpoint completion (CHECKPOINT_ONLINE). The start LSN of this checkpoint is specified after the word redo; this position corresponds to the latest inserted WAL entry at the time of the checkpoint start.

The same information can also be found in the pg_control file:

```
postgres$ /usr/local/pgsql/bin/pg_controldata \
-D /usr/local/pgsql/data | egrep 'Latest.*location'
Latest checkpoint location:           0/3EFC3190
Latest checkpoint's REDO location:    0/3EFC3158
```

10.4 Recovery

The first process launched at the server start is postmaster. In its turn, postmaster spawns the startup process,[1] which takes care of data recovery in case of a failure.

To determine whether recovery is needed, the startup process reads the pg_control file and checks the cluster status. The pg_controldata utility enables us to view the content of this file:

[1] backend/postmaster/startup.c
backend/access/transam/xlog.c, StartupXLOG function

```
postgres$ /usr/local/pgsql/bin/pg_controldata \
-D /usr/local/pgsql/data | grep state
Database cluster state:               in production
```

A properly stopped server has the "shut down" status; the "in production" status of a non-running server indicates a failure. In this case, the startup process will automatically initiate recovery from the start LSN of the latest completed checkpoint found in the same pg_control file.

If the PGDATA directory contains a backup_label file related to a backup, the start LSN position is taken from that file.

The startup process reads WAL entries one by one, starting from the defined position, and applies them to data pages if the LSN of the page is smaller than the LSN of the WAL entry. If the page contains a bigger LSN, WAL should not be applied; in fact, it *must not* be applied because its entries are designed to be replayed strictly sequentially.

However, some WAL entries constitute a *full page image*, or FPI. Entries of this type can be applied to any state of the page since all the page contents will be erased anyway. Such modifications are called *idempotent*. Another example of an idempotent operation is registering transaction status changes: each transaction status is defined in CLOG by certain bits that are set regardless of their previous values, so there is no need to keep the LSN of the latest change in CLOG pages.

WAL entries are applied to pages in the buffer cache, just like regular page updates during normal operation.

Files get restored from WAL in a similar manner: for example, if a WAL entry shows that the file must exit, but it is missing for some reason, it will be created anew.

Once the recovery is over, all unlogged relations are overwritten by the correspond-
p. 28 ing initialization forks.

Finally, the checkpoint is executed to secure the recovered state on disk.

The job of the startup process is now complete.

In its classic form, the recovery process consists of two phases. In the roll-forward phase, WAL entries are replayed, repeating the lost operations. In the roll-back phase, the server aborts the transactions that were not yet committed at the time of the failure.

> In PostgreSQL, the second phase is not required. After the recovery, the CLOG will contain neither commit nor abort bits for an unfinished transaction (which technically denotes an active transaction), but since it is known for sure that the transaction is not running anymore, it will be considered aborted.[1]

We can simulate a failure by forcing the server to stop in the immediate mode:

```
postgres$ pg_ctl stop -m immediate
```

Here is the new cluster state:

```
postgres$ /usr/local/pgsql/bin/pg_controldata \
-D /usr/local/pgsql/data | grep 'state'
Database cluster state:               in production
```

When we launch the server, the startup process sees that a failure has occurred and enters the recovery mode:

```
postgres$ pg_ctl start -l /home/postgres/logfile
postgres$ tail -n 6 /home/postgres/logfile
LOG:  database system was interrupted; last known up at 2022-11-25
22:58:50 MSK
LOG:  database system was not properly shut down; automatic recovery
in progress
LOG:  redo starts at 0/3EFC3158
LOG:  invalid record length at 0/3EFC3208: wanted 24, got 0
LOG:  redo done at 0/3EFC3190 system usage: CPU: user: 0.00 s,
system: 0.00 s, elapsed: 0.00 s
LOG:  database system is ready to accept connections
```

If the server is being stopped normally, postmaster disconnects all clients and then executes the final checkpoint to flush all dirty pages to disk.

Note the current WAL position:

```
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 0/3EFC3280
(1 row)
```

[1] backend/access/heap/heapam_visibility.c, HeapTupleSatisfiesMVCC function

Now let's stop the server properly:

```
postgres$ pg_ctl stop
```

Here is the new cluster state:

```
postgres$ /usr/local/pgsql/bin/pg_controldata \
-D /usr/local/pgsql/data | grep state
Database cluster state:               shut down
```

At the end of the WAL, we can see the CHECKPOINT_SHUTDOWN entry, which denotes the final checkpoint:

```
postgres$ /usr/local/pgsql/bin/pg_waldump \
-p /usr/local/pgsql/data/pg_wal -s 0/3EFC3280
rmgr: XLOG        len (rec/tot):    114/   114, tx:          0, lsn:
0/3EFC3280, prev 0/3EFC3208, desc: CHECKPOINT_SHUTDOWN redo
0/3EFC3280; tli 1; prev tli 1; fpw true; xid 0:889; oid 24754; multi
1; offset 0; oldest xid 726 in DB 1; oldest multi 1 in DB 1;
oldest/newest commit timestamp xid: 0/0; oldest running xid 0;
shutdown
pg_waldump: fatal: error in WAL record at 0/3EFC3280: invalid record
length at 0/3EFC32F8: wanted 24, got 0
```

The latest pg_waldump message shows that the utility has read the WAL to the end.

Let's start the instance again:

```
postgres$ pg_ctl start -l /home/postgres/logfile
```

10.5 Background Writing

If the backend needs to evict a dirty page from a buffer, it has to write this page to disk. Such a situation is undesired because it leads to waits—it is much better to perform writing asynchronously in the background.

This job is partially handled by checkpointer, but it is still not enough.

Therefore, PostgreSQL provides another process called bgwriter,[1] specifically for *background writing*. It relies on the same buffer search algorithm as eviction, except for the two main differences:

- The bgwriter process uses its own clock hand that never lags behind that of eviction and typically overtakes it.
- As the buffers are being traversed, the usage count is not reduced.

A dirty page is flushed to disk if the buffer is not pinned and has zero usage count. Thus, bgwriter runs before eviction and proactively writes to disk those pages that are highly likely to be evicted soon.

It raises the odds of the buffers selected for eviction being clean.

10.6 WAL Setup

Configuring Checkpoints

The checkpoint duration (to be more exact, the duration of writing dirty buffers to disk) is defined by the *checkpoint_completion_target* parameter. Its value specifies the fraction of time between the starts of two neighboring checkpoints that is allotted to writing. Avoid setting this parameter to one: as a result, the next checkpoint may be due before the previous one is complete. No disaster will happen, as it is impossible to execute more than one checkpoint at a time, but normal operation may still be disrupted. 0.9 v. 14

When configuring other parameters, we can use the following approach. First, we define an appropriate volume of WAL files to be stored between two neighboring checkpoints. The bigger the volume, the smaller the overhead, but this value will anyway be limited by the available free space and the acceptable recovery time.

To estimate the time required to generate this volume by *normal* load, you need to note the initial insert LSN and check the difference between this and the current insert positions from time to time.

[1] backend/postmaster/bgwriter.c

The received figure is assumed to be a typical interval between checkpoints, so we will use it as the *checkpoint_timeout* parameter value. The default setting is likely to be too small; it is usually increased, for example, to 30 minutes.

5min

p. 218

However, it is quite possible (and even probable) that the load will *sometimes* be higher, so the size of WAL files generated during this interval will be too big. In this case, the checkpoint must be executed more often. To set up such a trigger, we will limit the size of WAL files required for recovery by the *max_wal_size* parameter. When this threshold is exceeded, the server invokes an extra checkpoint.[1]

1GB

v. 11

WAL files required for recovery contain all the entries both for the latest completed checkpoint and for the current one, which is not completed yet. So to estimate their total volume you should multiply the calculated WAL size between checkpoints by 1 + *checkpoint_completion_target*.

> Prior to version 11, PostgreSQL kept WAL files for two completed checkpoints, so the multiplier was 2 + *checkpoint_completion_target*.

Following this approach, most checkpoints are executed on schedule, once per the *checkpoint_timeout* interval; but should the load increase, the checkpoint is triggered when WAL size exceeds the *max_wal_size* value.

The actual progress is periodically checked against the expected figures:[2]

The actual progress is defined by the fraction of cached pages that have already been processed.

The expected progress (by time) is defined by the fraction of time that has already elapsed, assuming that the checkpoint must be completed within the *checkpoint_timeout* × *checkpoint_completion_target* interval.

The expected progress (by size) is defined by the fraction of the already filled WAL files, their expected number being estimated based on the *max_wal_size* × *checkpoint_completion_target* value.

If dirty pages get written to disk ahead of schedule, checkpointer is paused for a while; if there is any delay by either of the parameters, it catches up as soon as

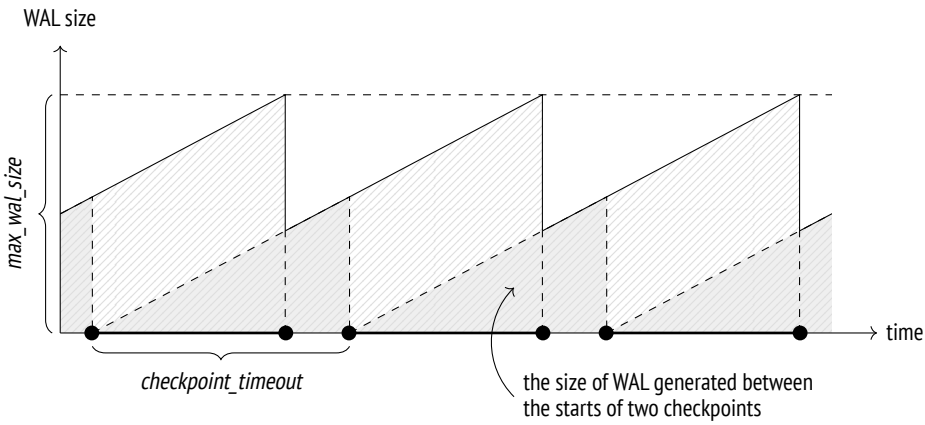
[1] backend/access/transam/xlog.c, LogCheckpointNeeded & CalculateCheckpointSegments functions
[2] backend/postmaster/checkpointer.c, IsCheckpointOnSchedule function

possible.[1] Since both time and data size are taken into account, PostgreSQL can manage scheduled and on-demand checkpoints using the same approach.

Once the checkpoint has been completed, WAL files that are not required for recovery anymore are deleted;[2] however, several files (up to *min_wal_size* in total) are kept for reuse and are simply renamed. 80MB

Such renaming reduces the overhead incurred by constant file creation and deletion, but you can turn off this feature using the *wal_recycle* parameter if you do not need it. v. 12 on

The following figure shows how the size of WAL files stored on disk changes under normal conditions.



It is important to keep in mind that the actual size of WAL files on disk may exceed the *max_wal_size* value:

- The *max_wal_size* parameter specifies the desired target value rather than a hard limit. If the load spikes, writing may lag behind the schedule.
- The server has no right to delete WAL files that are yet to be replicated or handled by continuous archiving. If enabled, this functionality must be constantly monitored, as it can easily cause a disk overflow.

[1] backend/postmaster/checkpointer.c, CheckpointWriteDelay function
[2] backend/access/transam/xlog.c, RemoveOldXlogFiles function

v. 12 0MB

- You can reserve a certain amount of space for WAL files by configuring the *wal_keep_size* parameter.

Configuring Background Writing

Once checkpointer is configured, you should also set up bgwriter. Together, these processes must be able to cope with writing dirty buffers to disk before backends need to reuse them.

200ms During its operation, bgwriter makes periodic pauses, sleeping for *bgwriter_delay* units of time.

The number of pages written between two pauses depends on the average number of buffers accessed by backends since the previous run (PostgreSQL uses a moving average to level out possible spikes and avoid depending on very old data at the same time). The calculated number is then multiplied by *bgwriter_lru_multiplier*. But in any case, the number of pages written in a single run cannot exceed the *bgwriter_lru_maxpages* value.

2

100

If no dirty buffers are detected (that is, nothing happens in the system), bgwriter sleeps until one of the backends accesses a buffer. Then it wakes up and continues its regular operation.

Monitoring

Checkpoint settings can and should be tuned based on monitoring data.

If size-triggered checkpoints have to be performed more often than defined by the *checkpoint_warning* parameter, PostgreSQL issues a warning. This setting should be brought in line with the expected peak load.

30s

off The *log_checkpoints* parameter enables printing checkpoint-related information into the server log. Let's turn it on:

```
=> ALTER SYSTEM SET log_checkpoints = on;
=> SELECT pg_reload_conf();
```

Now we will modify some data and execute a checkpoint:

```
=> UPDATE big SET s = 'BAR';
=> CHECKPOINT;
```

The server log shows the number of written buffers, some statistics on WAL file changes after the checkpoint, the duration of the checkpoint, and the distance (in bytes) between the starts of two neighboring checkpoints:

```
postgres$ tail -n 2 /home/postgres/logfile
LOG:  checkpoint starting: immediate force wait
LOG:  checkpoint complete: wrote 4100 buffers (25.0%); 0 WAL file(s)
added, 1 removed, 0 recycled; write=0.052 s, sync=0.004 s,
total=0.068 s; sync files=3, longest=0.002 s, average=0.002 s;
distance=9213 kB, estimate=9213 kB
```

The most useful data that can affect your configuration decisions is statistics on background writing and checkpoint execution provided in the pg_stat_bgwriter view.

> Prior to version 9.2, both tasks were performed by bgwriter; then a separate checkpointer process was introduced, but the common view remained unchanged.

```
=> SELECT * FROM pg_stat_bgwriter \gx
-[ RECORD 1 ]---------+------------------------------
checkpoints_timed     | 0
checkpoints_req       | 14
checkpoint_write_time | 34539
checkpoint_sync_time  | 184
buffers_checkpoint    | 14518
buffers_clean         | 14237
maxwritten_clean      | 133
buffers_backend       | 85267
buffers_backend_fsync | 0
buffers_alloc         | 85324
stats_reset           | 2022-11-25 22:57:04.320736+03
```

Among other things, this view displays the number of completed checkpoints:

- The checkpoints_timed field shows scheduled checkpoints (which are triggered when the *checkpoint_timeout* interval is reached).
- The checkpoints_req field shows on-demand checkpoints (including those triggered when the *max_wal_size* size is reached).

A large checkpoint_req value (as compared to checkpoints_timed) indicates that checkpoints are performed more often than expected.

The following statistics on the number of written pages are also very important:

- buffers_checkpoint pages written by checkpointer
- buffers_backend pages written by backends
- buffers_clean pages written by bgwriter

In a well-configured system, the buffers_backend value must be considerably lower than the sum of buffers_checkpoint and buffers_clean.

When setting up background writing, pay attention to the maxwritten_clean value: it shows how many times bgwriter had to stop because of exceeding the threshold defined by *bgwriter_lru_maxpages*.

The following call will drop the collected statistics:

```
=> SELECT pg_stat_reset_shared('bgwriter');
```

11

WAL Modes

11.1 Performance

While the server is running normally, WAL files are being constantly written to disk. However, these writes are sequential: there is almost no random access, so even HDDs can cope with this task. Since this type of load is very different from a typical data file access, it may be worth setting up a separate physical storage for WAL files and replacing the PGDATA/pg_wal catalog by a symbolic link to a directory in a mounted file system.

> There are a couple of situations when WAL files have to be both written and read. The first one is the obvious case of crash recovery; the second one is stream replication. The walsender[1] process reads WAL entries directly from files.[2] So if a replica does not receive WAL entries while the required pages are still in the OS buffers of the primary server, the data has to be read from disk. But the access will still be sequential rather than random.

WAL entries can be written in one of the following modes:

- The synchronous mode forbids any further operations until a transaction commit saves all the related WAL entries to disk.
- The asynchronous mode implies instant transaction commits, with WAL entries being written to disk later in the background.

The current mode is defined by the *synchronous_commit* parameter. on

[1] backend/replication/walsender.c
[2] backend/access/transam/xlogreader.c

Synchronous mode. To reliably register the fact of a commit, it is not enough to simply pass WAL entries to the operating system; you have to make sure that disk synchronization has completed successfully. Since synchronization implies actual I/O operations (which are quite slow), it is beneficial to perform it as seldom as possible.

For this purpose, the backend that completes the transaction and writes WAL entries to disk can make a small pause as defined by the *commit_delay* parameter. However, it will only happen if there are at least *commit_siblings* active transactions in the system:[1] during this pause, some of them may finish, and the server will manage to synchronize all the WAL entries in one go. It is a lot like holding doors of an elevator for someone to rush in.

0s

5

By default, there is no pause. It makes sense to modify the *commit_delay* parameter only for systems that perform a lot of short OLTP transactions.

After a potential pause, the process that completes the transaction flushes all the accumulated WAL entries to disk and performs synchronization (it is important to save the commit entry and all the previous entries related to this transaction; the rest is written just because it does not increase the cost).

From this time on, the ACID's durability requirement is guaranteed—the transaction is considered to be reliably committed.[2] That's why the synchronous mode is the default one.

The downside of the synchronous commit is longer latencies (the COMMIT command does not return control until the end of synchronization) and lower system throughput, especially for OLTP loads.

Asynchronous mode. To enable asynchronous commits,[3] you have to turn off the *synchronous_commit* parameter.

In the asynchronous mode, WAL entries are written to disk by the walwriter[4] process, which alternates between work and sleep. The duration of pauses is defined by the *wal_writer_delay* value.

200ms

[1] backend/access/transam/xlog.c, XLogFlush function
[2] backend/access/transam/xlog.c, RecordTransactionCommit function
[3] postgresql.org/docs/14/wal-async-commit.html
[4] backend/postmaster/walwriter.c

Waking up from a pause, the process checks the cache for new *completely filled* WAL pages. If any such pages have appeared, the process writes them to disk, skipping the current page. Otherwise, it writes the current half-empty page since it has woken up anyway.[1]

The purpose of this algorithm is to avoid flushing one and the same page several times, which brings noticeable performance gains for workloads with intensive data changes.

Although WAL cache is used as a ring buffer, walwriter stops when it reaches the last page of the cache; after a pause, the next writing cycle starts from the first page. So in the worst case walwriter needs three runs to get to a particular WAL entry: first, it will write all full pages located at the end of the cache, then it will get back to the beginning, and finally, it will handle the underfilled page containing the entry. But in most cases it takes one or two cycles.

Synchronization is performed each time the *wal_writer_flush_after* amount of data is written, and once again at the end of the writing cycle. 1MB

Asynchronous commits are faster than synchronous ones since they do not have to wait for physical writes to disk. But reliability suffers: you can lose the data committed within the $3 \times$ *wal_writer_delay* timeframe before a failure (which is 0.6 seconds by default).

In the real world, these two modes complement each other. In the synchronous mode, WAL entries related to a long transaction can still be written asynchronously to free WAL buffers. And vice versa, a WAL entry related to a page that is about to be evicted from the buffer cache will be immediately flushed to disk even in the asynchronous mode—otherwise, it is impossible to continue operation.

In most cases, a hard choice between performance and durability has to be made by the system designer.

The *synchronous_commit* parameter can also be set for particular transactions. If it is possible to classify all transactions at the application level as either absolutely critical (such as handling financial data) or less important, you can boost performance while risking to lose only non-critical transactions.

[1] backend/access/transam/xlog.c, XLogBackgroundFlush function

To get some idea of potential performance gains of the asynchronous commit, let's compare latency and throughput in the two modes using a pgbench test.[1]

First, initialize the required tables:

```
postgres$ /usr/local/pgsql/bin/pgbench -i internals
```

Start a 30-second test in the synchronous mode:

```
postgres$ /usr/local/pgsql/bin/pgbench -T 30 internals
pgbench (14.4)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
duration: 30 s
number of transactions actually processed: 23171
latency average = 1.295 ms
initial connection time = 1.956 ms
tps = 772.385024 (without initial connection time)
```

And now run the same test in the asynchronous mode:

```
=> ALTER SYSTEM SET synchronous_commit = off;
=> SELECT pg_reload_conf();
postgres$ /usr/local/pgsql/bin/pgbench -T 30 internals
pgbench (14.4)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
duration: 30 s
number of transactions actually processed: 75377
latency average = 0.398 ms
initial connection time = 2.219 ms
tps = 2512.600587 (without initial connection time)
```

[1] postgresql.org/docs/14/pgbench.html

In the asynchronous mode, this simple benchmark shows a significantly lower latency and higher throughput (TPS). Naturally, each particular system will have its own figures depending on the current load, but it is clear that the impact on short OLTP transactions can be quite tangible.

Let's restore the default settings:

```
=> ALTER SYSTEM RESET synchronous_commit;
=> SELECT pg_reload_conf();
```

11.2 Fault Tolerance

It is self-evident that write-ahead logging must guarantee crash recovery under any circumstances (unless the persistent storage itself is broken). There are many factors that can affect data consistency, but I will cover only the most important ones: caching, data corruption, and non-atomic writes.[1]

Caching

Before reaching a non-volatile storage (such as a hard disk), data can pass through various caches.

A disk write simply instructs the operating system to place the data into its cache (which is also prone to crashes, just like any other part of RAM). The actual writing is performed asynchronously, as defined by the settings of the I/O scheduler of the operating system.

Once the scheduler decides to flush the accumulated data, this data is moved to the cache of a storage device (like an HDD). Storage devices can also defer writing, for example, to group of adjacent pages together. A RAID controller adds one more caching level between the disk and the operating system.

Unless special measures are taken, the moment when the data is reliably stored on disk remains unknown. It is usually not so important because we have the WAL,

[1] postgresql.org/docs/14/wal-reliability.html

but WAL entries themselves must be reliably saved on disk right away.[1] It is equally true for the asynchronous mode—otherwise, it is impossible to guarantee that WAL entries get do disk ahead of the modified data.

The checkpointer process must also save the data in a reliable way, ensuring that dirty pages make it to disk from the OS cache. Besides, it has to synchronize all the file operations that have been performed by other processes (such as page writes or file deletions): when the checkpoint completes, the results of all these actions must be already saved on disk.[2]

There are also some other situations that demand fail-safe writing, such as executing unlogged operations at the minimal WAL level.

Operating systems provide various means to guarantee immediate writing of data into a non-volatile storage. All of them boil down to the following two main approaches: either a separate synchronization command is called after writing (such as fsync or fdatasync), or the requirement to perform synchronization (or even direct writing that bypasses OS cache) is specified when the file is being opened or written into.

The pg_test_fsync utility can help you determine the best way to synchronize the WAL depending on your OS and file system; the preferred method can be specified in the *wal_sync_method* parameter. For other operations, an appropriate synchronization method is selected automatically and cannot be configured.[3]

A subtle aspect here is that in each particular case the most suitable method depends on the hardware. For example, if you use a controller with a backup battery, you can take advantage of its cache, as the battery will protect the data in case of a power outage.

You should keep in mind that the asynchronous commit and lack of synchronization are two totally different stories. Turning off synchronization (by the *fsync* parameter) boosts system performance, yet any failure will lead to fatal data loss. The asynchronous mode guarantees crash recovery up to a consistent state, but some of the latest data updates may be missing.

on

[1] backend/access/transam/xlog.c, issue_xlog_fsync function
[2] backend/storage/sync/sync.c
[3] backend/storage/file/fd.c, pg_fsync function

Data Corruption

Technical equipment is imperfect, and data can get damaged both in memory and on disk, or while it is being transferred via interface cables. Such errors are usually handled at the hardware level, yet some can escape.

To catch issues in good time, PostgreSQL always protects WAL entries by checksums.

Checksums can be calculated for data pages as well.[1] It is done either during cluster initialization or by running the pg_checksums[2] utility when the server is stopped.[3] v. 12

In production systems, checksums must always be enabled, despite some (minor) calculation and verification overhead. It raises the chance of timely corruption discovery, even though some corner cases still remain:

- Checksum verification is performed only when the page is accessed, so data corruption can go unnoticed for a long time, up to the point when it gets into all backups and leaves no source of correct data.
- A zeroed page is considered correct, so if the file system zeroes out a page by mistake, this issue will not be discovered.
- Checksums are calculated only for the main fork of relations; other forks and files (such as transaction status in CLOG) remain unprotected.

Let's take a look at the read-only *data_checksums* parameter to make sure that checksums are enabled:

```
=> SHOW data_checksums;
 data_checksums
----------------
 on
(1 row)
```

Now stop the server and zero out several bytes in the zero page of the main fork of the table:

[1] backend/storage/page/README
[2] postgresql.org/docs/14/app-pgchecksums.html
[3] commitfest.postgresql.org/27/2260

```
=> SELECT pg_relation_filepath('wal');
 pg_relation_filepath
----------------------
 base/16391/16562
(1 row)
postgres$ pg_ctl stop
postgres$ dd if=/dev/zero of=/usr/local/pgsql/data/base/16391/16562 \
oflag=dsync conv=notrunc bs=1 count=8
8+0 records in
8+0 records out
8 bytes copied, 0,00620759 s, 1,3 kB/s
```

Start the server again:

```
postgres$ pg_ctl start -l /home/postgres/logfile
```

In fact, we could have left the server running—it is enough to write the page to disk and evict it from cache (otherwise, the server will continue using its cached version). But such a workflow is harder to reproduce.

Now let's attempt to read the table:

```
=> SELECT * FROM wal LIMIT 1;
WARNING:  page verification failed, calculated checksum 24386 but
expected 32432
ERROR:  invalid page in block 0 of relation base/16391/16562
```

If the data cannot be restored from a backup, it makes sense to at least try to read the damaged page (risking to get garbled output). For this purpose, you have to enable the *ignore_checksum_failure* parameter:

off

```
=> SET ignore_checksum_failure = on;
=> SELECT * FROM wal LIMIT 1;
WARNING:  page verification failed, calculated checksum 24386 but
expected 32432
 id
----
  2
(1 row)
```

Everything went fine in this case because we have damaged a non-critical part of the page header (the LSN of the latest WAL entry), not the data itself.

Non-Atomic Writes

A database page usually takes 8 kB, but at the low level writing is performed by blocks, which are often smaller (typically 512 bytes or 4 kB). Thus, if a failure occurs, a page may be written only partially. It makes no sense to apply regular WAL entries to such a page during recovery.

To avoid partial writes, PostgreSQL saves a full page image (FPI) in the WAL when this page is modified for the first time after the checkpoint start. This behavior is controlled by the *full_page_writes* parameter, but turning it off can lead to fatal data corruption. p. 200 on

If the recovery process comes across an FPI in the WAL, it will unconditionally write it to disk (without checking its LSN); just like any WAL entry, FPIs are protected by checksums, so their damage cannot go unnoticed. Regular WAL entries will then be applied to this state, which is guaranteed to be correct.

There is no separate WAL entry type for setting hint bits: this operation is considered non-critical because any query that accesses a page will set the required bits anew. However, any hint bit change will affect the page's checksum. So if checksums are enabled (or if the *wal_log_hints* parameter is on), hint bit modifications are logged as FPIs.[1] p. 80 off

Even though the logging mechanism excludes empty space from an FPI,[2] the size of the generated WAL files still significantly increases. The situation can be greatly improved if you enable FPI compression via the *wal_compression* parameter. off

Let's run a simple experiment using the pgbench utility. We will perform a checkpoint and immediately start a benchmark test with a hard-set number of transactions:

```
=> CHECKPOINT;
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 0/43B38F00
(1 row)
```

[1] backend/storage/buffer/bufmgr.c, MarkBufferDirtyHint function
[2] backend/access/transam/xloginsert.c, XLogRecordAssemble function

```
postgres$ /usr/local/pgsql/bin/pgbench -t 20000 internals
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 0/451B76A0
(1 row)
```

Here is the size of the generated WAL entries:

```
=> SELECT pg_size_pretty('0/451B76A0'::pg_lsn - '0/43B38F00'::pg_lsn);
 pg_size_pretty
----------------
 22 MB
(1 row)
```

In this example, FPIs take more than half of the total WAL size. You can see it for yourself in the collected statistics that show the number of WAL entries (N), the size of regular entries (Record size), and the FPI size for each resource type (Type):

```
postgres$ /usr/local/pgsql/bin/pg_waldump --stats \
-p /usr/local/pgsql/data/pg_wal -s 0/43B38F00 -e 0/451B76A0
Type            N      (%)   Record size      (%)   FPI size      (%)
----            -      ---   -----------      ---   --------      ---
XLOG         1843 (  1,51)         90307 (  1,13)   14825032 ( 97,12)
Transaction 20001 ( 16,38)        680114 (  8,53)          0 (  0,00)
Storage         1 (  0,00)            42 (  0,00)          0 (  0,00)
Standby         2 (  0,00)            96 (  0,00)          0 (  0,00)
Heap2       20217 ( 16,56)       1282808 ( 16,10)      16384 (  0,11)
Heap        80025 ( 65,53)       5914356 ( 74,21)     300944 (  1,97)
Btree          27 (  0,02)          1568 (  0,02)     122020 (  0,80)
            ------                --------            --------
Total      122116                 7969291 [34,30%]   15264380 [65,70%]
```

This ratio will be smaller if data pages get modified between checkpoints several times. It is yet another reason to perform checkpoints less often.

We will repeat the same experiment to see if compression can help.

```
=> ALTER SYSTEM SET wal_compression = on;
=> SELECT pg_reload_conf();
=> CHECKPOINT;
```

```
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 0/451B7750
(1 row)
postgres$ /usr/local/pgsql/bin/pgbench -t 20000 internals
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 0/45C74F48
(1 row)
```

Here is the WAL size with compression enabled:

```
=> SELECT pg_size_pretty('0/45C74F48'::pg_lsn - '0/451B7750'::pg_lsn);
 pg_size_pretty
----------------
 11 MB
(1 row)
postgres$ /usr/local/pgsql/bin/pg_waldump --stats \
-p /usr/local/pgsql/data/pg_wal -s 0/451B7750 -e 0/45C74F48
Type            N      (%)   Record size      (%)   FPI size      (%)
----            -      ---   -----------      ---   --------      ---
XLOG         1862 (  1,52)         94962 (  1,19)    2934302 ( 98,53)
Transaction 20001 ( 16,38)        680114 (  8,53)          0 (  0,00)
Storage         1 (  0,00)            42 (  0,00)          0 (  0,00)
CLOG            1 (  0,00)            30 (  0,00)          0 (  0,00)
Standby         3 (  0,00)           150 (  0,00)          0 (  0,00)
Heap2       20230 ( 16,56)       1285922 ( 16,13)        244 (  0,01)
Heap        80015 ( 65,52)       5912094 ( 74,14)      36650 (  1,23)
Btree          17 (  0,01)          1061 (  0,01)       6735 (  0,23)
            ------                --------             --------
Total      122130                 7974375 [72,81%]    2977931 [27,19%]
```

To sum it up, when there is a large number of FPIs caused by enabled checksums or *full_page_writes* (that is, almost always), it makes sense to use compression despite some additional CPU overhead.

11.3 WAL Levels

The main objective of write-ahead logging is to enable crash recovery. But if you extend the scope of logged information, a WAL can be used for other purposes too.

PostgreSQL provides minimal, replica, and logical logging levels. Each level includes everything that is logged on the previous one and adds some more information.

replica The level in use is defined by the *wal_level* parameter; its modification requires a server restart.

Minimal

The minimal level guarantees only crash recovery. To save space, the operations on relations that have been created or truncated within the current transaction are not logged if they incur insertion of large volumes of data (like in the case of CREATE TABLE AS SELECT and CREATE INDEX commands).[1] Instead of being logged, all the required data is immediately flushed to disk, and system catalog changes become visible right after the transaction commit.

If such an operation is interrupted by a failure, the data that has already made it to disk remains invisible and does not affect consistency. If a failure occurs when the operation is complete, all the data required for applying the subsequent WAL entries is already saved to disk.

v. 13 The volume of data that has to be written into a newly created relation for this
2MB optimization to take effect is defined by the *wal_skip_threshold* parameter.

Let's see what gets logged at the minimal level.

v. 10 By default, a higher replica level is used, which supports data replication. If you choose the minimal level, you also have to set the allowed number of walsender
10 processes to zero in the *max_wal_senders* parameter:

```
=> ALTER SYSTEM SET wal_level = minimal;
=> ALTER SYSTEM SET max_wal_senders = 0;
```

The server has to be restarted for these changes to take effect:

```
postgres$ pg_ctl restart -l /home/postgres/logfile
```

Note the current WAL position:

```
=> SELECT pg_current_wal_insert_lsn();
```

[1] include/utils/rel.h, RelationNeedsWAL macro

```
 pg_current_wal_insert_lsn
---------------------------
 0/45C77230
(1 row)
```

Truncate the table and keep inserting new rows within the same transaction until the *wal_skip_threshold* is exceeded:

```
=> BEGIN;
=> TRUNCATE TABLE wal;
=> INSERT INTO wal
     SELECT id FROM generate_series(1,100000) id;
=> COMMIT;
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 0/45C773D8
(1 row)
```

Instead of creating a new table, I run the TRUNCATE command as it generates fewer WAL entries.

Let's examine the generated WAL using the already familiar pg_waldump utility.

```
postgres$ /usr/local/pgsql/bin/pg_waldump \
-p /usr/local/pgsql/data/pg_wal -s 0/45C77230 -e 0/45C773D8#
rmgr: Storage     len (rec/tot):     42/    42, tx:          0, lsn:
0/45C77230, prev 0/45C771F8, desc: CREATE base/16391/24784
rmgr: Heap        len (rec/tot):    123/   123, tx:     139456, lsn:
0/45C77260, prev 0/45C77230, desc: UPDATE off 45 xmax 139456 flags
0x60 ; new off 48 xmax 0, blkref #0: rel 1663/16391/1259 blk 0
rmgr: Btree       len (rec/tot):     64/    64, tx:     139456, lsn:
0/45C772E0, prev 0/45C77260, desc: INSERT_LEAF off 176, blkref #0:
rel 1663/16391/2662 blk 2
rmgr: Btree       len (rec/tot):     64/    64, tx:     139456, lsn:
0/45C77320, prev 0/45C772E0, desc: INSERT_LEAF off 147, blkref #0:
rel 1663/16391/2663 blk 2
rmgr: Btree       len (rec/tot):     64/    64, tx:     139456, lsn:
0/45C77360, prev 0/45C77320, desc: INSERT_LEAF off 254, blkref #0:
rel 1663/16391/3455 blk 4
rmgr: Transaction len (rec/tot):     54/    54, tx:     139456, lsn:
0/45C773A0, prev 0/45C77360, desc: COMMIT 2022-11-25 23:00:47.409036
MSK; rels: base/16391/24783
```

p. 162 The first entry logs creation of a new file for the relation (since TRUNCATE virtually rewrites the table).

The next four entries are associated with system catalog operations. They reflect the changes in the pg_class table and its three indexes.

Finally, there is a commit-related entry. Data insertion is not logged.

Replica

During crash recovery, WAL entries are replayed to restore the data on disk up to a consistent state. Backup recovery works in a similar way, but it can also restore the database state up to the specified recovery target point using a WAL archive. The number of archived WAL entries can be quite high (for example, they can span several days), so the recovery period will include multiple checkpoints. Therefore, the minimal WAL level is not enough: it is impossible to repeat an operation if it is unlogged. For backup recovery, WAL files must include *all* the operations.

The same is true for replication: unlogged commands will not be sent to a replica and will not be replayed on it.

Things get even more complicated if a replica is used for executing queries. First p. 232 of all, it needs to have the information on exclusive locks acquired on the primary server since they may conflict with queries on the replica. Second, it must be able p. 92 to capture snapshots, which requires the information on active transactions. When we deal with a replica, both local transactions and those running on the primary server have to be taken into account.

The only way to send this data to a replica is to periodically write it into WAL files.[1] It is done by the bgwriter[2] process, once in 15 seconds (the interval is hard-coded).

The ability to perform data recovery from a backup and use physical replication is guaranteed at the replica level.

[1] backend/storage/ipc/standby, LogStandbySnapshot function
[2] backend/postmaster/bgwriter.c

The replica level is used by default, so we can simply reset the parameters configured above and restart the server: v. 10

```
=> ALTER SYSTEM RESET wal_level;
=> ALTER SYSTEM RESET max_wal_senders;

postgres$ pg_ctl restart -l /home/postgres/logfile
```

Let's repeat the same workflow as before (but now we will insert only one row to get a neater output):

```
=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 0/462989F0
(1 row)

=> BEGIN;
=> TRUNCATE TABLE wal;
=> INSERT INTO wal VALUES (42);
=> COMMIT;

=> SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 0/46298CB0
(1 row)
```

Check out the generated WAL entries.

Apart from what we have seen at the minimal level, we have also got the following entries:

- replication-related entries of the Standby resource manager: RUNNING_XACTS (active transactions) and LOCK
- the entry that logs the INSERT+INIT operation, which initializes a new page and inserts a new row into this page

```
postgres$ /usr/local/pgsql/bin/pg_waldump \
-p /usr/local/pgsql/data/pg_wal -s 0/462989F0 -e 0/46298CB0
rmgr: Standby     len (rec/tot):     42/    42, tx:     139458, lsn:
0/462989F0, prev 0/46298978, desc: LOCK xid 139458 db 16391 rel 16562
rmgr: Storage     len (rec/tot):     42/    42, tx:     139458, lsn:
0/46298A20, prev 0/462989F0, desc: CREATE base/16391/24786
rmgr: Heap        len (rec/tot):    123/   123, tx:     139458, lsn:
0/46298A50, prev 0/46298A20, desc: UPDATE off 49 xmax 139458 flags
0x60 ; new off 50 xmax 0, blkref #0: rel 1663/16391/1259 blk 0
rmgr: Btree       len (rec/tot):     64/    64, tx:     139458, lsn:
0/46298AD0, prev 0/46298A50, desc: INSERT_LEAF off 178, blkref #0:
rel 1663/16391/2662 blk 2
rmgr: Btree       len (rec/tot):     64/    64, tx:     139458, lsn:
0/46298B10, prev 0/46298AD0, desc: INSERT_LEAF off 149, blkref #0:
rel 1663/16391/2663 blk 2
rmgr: Btree       len (rec/tot):     64/    64, tx:     139458, lsn:
0/46298B50, prev 0/46298B10, desc: INSERT_LEAF off 256, blkref #0:
rel 1663/16391/3455 blk 4
rmgr: Heap        len (rec/tot):     59/    59, tx:     139458, lsn:
0/46298B90, prev 0/46298B50, desc: INSERT+INIT off 1 flags 0x00,
blkref #0: rel 1663/16391/24786 blk 0
rmgr: Standby     len (rec/tot):     42/    42, tx:          0, lsn:
0/46298BD0, prev 0/46298B90, desc: LOCK xid 139458 db 16391 rel 16562
rmgr: Standby     len (rec/tot):     54/    54, tx:          0, lsn:
0/46298C00, prev 0/46298BD0, desc: RUNNING_XACTS nextXid 139459
latestCompletedXid 139457 oldestRunningXid 139458; 1 xacts: 139458
rmgr: Transaction len (rec/tot):    114/   114, tx:     139458, lsn:
0/46298C38, prev 0/46298C00, desc: COMMIT 2022-11-25 23:01:03.404607
MSK; rels: base/16391/24785; inval msgs: catcache 51 catcache 50
relcache 16562
```

Logical

Last but not least, the logical level enables logical decoding and logical replication. It has to be activated on the publishing server.

If we take a look at WAL entries, we will see that this level is almost the same as replica: it adds the entries related to replication sources and some arbitrary logical entries that may be generated by applications. For the most part, logical decoding depends on the information about active transactions (RUNNING_XACTS) because it requires capturing a snapshot to track system catalog changes.

Part III

Locks

12

Relation-Level Locks

12.1 About Locks

Locks control concurrent access to shared resources.

Concurrent access implies that several processes try to get one and the same resource at the same time. It makes no difference whether these processes are executed in parallel (if the hardware permits) or sequentially in the time-sharing mode. If there is no concurrent access, there is no need to acquire locks (for example, shared buffer cache requires locking, while local cache can do without it).

Before accessing a resource, the process must *acquire* a lock on it; when the operation is complete, this lock must be *released* for the resource to become available to other processes. If locks are managed by the database system, the established order of operations is maintained automatically; if locks are controlled by the application, the protocol must be enforced by the application itself.

At a low level, a lock is simply a chunk of shared memory that defines the lock status (whether it is acquired or not); it can also provide some additional information, such as the process number or acquisition time.

> As you can guess, a shared memory segment is a resource in its own right. Concurrent access to such resources is regulated by synchronization primitives (such as semaphores or mutexes) provided by the operating system. They guarantee strictly consecutive execution of the code that accesses a shared resource. At the lowest level, these primitives are based on atomic CPU instructions (such as test-and-set or compare-and-swap).

In general, we can use locks to protect any resource as long as it can be unambiguously identified and assigned a particular lock address.

For example, we can lock a database object, such as a table (identified by oid in the system catalog), a data page (identified by a filename and a position within this file), a row version (identified by a page and an offset within this page). We can also lock a memory structure, such as a hash table or a buffer (identified by an assigned ID). We can even lock an abstract resource that has no physical representation.

But it is not always possible to acquire a lock at once: a resource can be already locked by someone else. Then the process either joins the queue (if it is allowed for this particular lock type) or tries again some time later. Either way, it has to wait for the lock to be released.

I would like to single out two factors that can greatly affect locking efficiency.

Granularity, or the "grain size" of a lock. Granularity is important if resources form a hierarchy.

For example, a table consists of pages, which, in their turn, consist of tuples. All these objects can be protected by locks. Table-level locks are coarse-grained; they forbid concurrent access even if the processes need to get to different pages or rows.

Row-level locks are fine-grained, so they do not have this drawback; however, the number of locks grows. To avoid using too much memory for lock-related metadata, PostgreSQL can apply various methods, one of them being *lock escalation*: if the number of fine-grained locks exceeds a certain threshold, they are replaced by a single lock of coarser granularity.

A set of modes in which a lock can be acquired.

As a rule, only two modes are applied. The *exclusive* mode is incompatible with all the other modes, including itself. The *shared* mode allows a resource to be locked by several processes at a time. The shared mode can be used for reading, while the exclusive mode is applied for writing.

In general, there may be other modes too. Names of modes are unimportant, it is their compatibility matrix that matters.

Finer granularity and support for multiple compatible modes give more opportunities for concurrent execution.

All locks can be classified by their duration.

Long-term locks are acquired for a potentially long time (in most cases, till the end of the transaction); they typically protect such resources as relations and rows. These locks are usually managed by PostgreSQL automatically, but a user still has some control over this process.

Long-term locks offer multiple modes that enable various concurrent operations on data. They usually have extensive infrastructure (including such features as wait queues, deadlock detection, and instrumentation) since its maintenance is anyway much cheaper than operations on protected data.

Short-term locks are acquired for fractions of a second and rarely last longer than several CPU instructions; they usually protect data structures in the shared memory. PostgreSQL manages such locks in a fully automated way.

Short-term locks typically offer very few modes and only basic infrastructure, which may have no instrumentation at all.

PostgreSQL supports various types of locks.[1] *Heavyweight locks* (which are acquired on relations and other objects) and *row-level locks* are considered long-term. Short-term locks comprise various *locks on memory structures*. Besides, there is also a distinct group of *predicate locks*, which, despite their name, are not locks at all.

p. 239
p. 274
p. 268

12.2 Heavyweight Locks

Heavyweight locks are long-term ones. Acquired at the *object* level, they are mainly used for relations, but can also be applied to some other types of objects. Heavyweight locks typically protect objects from concurrent updates or forbid their usage during restructuring, but they can address other needs too. Such a vague definition is deliberate: locks of this type are used for all kinds of purposes. The only thing they have in common is their internal structure.

Unless explicitly specified otherwise, the term *lock* usually implies a heavyweight lock.

[1] backend/storage/lmgr/README

Heavyweight locks are located in the server's shared memory[1] and can be displayed
64 in the pg_locks view. Their total number is limited by the *max_locks_per_transaction*
100 value multiplied by *max_connections*.

All transactions use a common pool of locks, so one transaction can acquire more than *max_locks_per_transaction* locks. What really matters is that the total number of locks in the system does not exceed the defined limit. Since the pool is initialized when the server is launched, changing any of these two parameters requires a server restart.

If a resource is already locked in an incompatible mode, the process trying to acquire another lock joins the queue. Waiting processes do not waste CPU time: they fall asleep until the lock is released and the operating system wakes them up.

p. 256 Two transactions can find themselves in a *deadlock* if the first transaction is unable to continue its operation until it gets a resource locked by the other transaction, which, in its turn, needs a resource locked by the first transaction. This case is rather simple; a deadlock can also involve more than two transactions. Since deadlocks cause infinite waits, PostgreSQL detects them automatically and aborts one of the affected transactions to ensure that normal operation can continue.

Different types of heavyweight locks serve different purposes, protect different resources, and support different modes, so we will consider them separately.

The following list provides the names of lock types as they appear in the locktype column of the pg_locks view:

p. 231 **transactionid** and **virtualxid** — a lock on a transaction ID

p. 232 **relation** — a relation-level lock

p. 245 **tuple** — a lock acquired on a tuple

p. 263 **object** — a lock on an object that is not a relation

p. 265 **extend** — a relation extension lock

p. 265 **page** — a page-level lock used by some index types

p. 266 **advisory** — an advisory lock

[1] backend/storage/lmgr/lock.c

Almost all heavyweight locks are acquired automatically as needed and are released automatically when the corresponding transaction completes. There are some exceptions though: for example, a relation-level lock can be set explicitly, while advisory locks are always managed by users.

12.3 Locks on Transaction IDs

Each transaction always holds an exclusive lock on its own ID (both virtual and real, if available). p. 85

PostgreSQL offers two locking modes for this purpose, exclusive and shared. Their compatibility matrix is very simple: the shared mode is compatible with itself, while the exclusive mode cannot be combined with any mode.

	Shared	Exclusive
Shared		×
Exclusive	×	×

To track completion of a particular transaction, a process can request a lock on this transaction's ID, in any mode. Since the transaction itself is already holding an exclusive lock on its own ID, another lock is impossible to acquire. The process requesting this lock joins the queue and falls asleep. Once the transaction completes, the lock is released, and the queued process wakes up. Clearly, it will not manage to acquire the lock because the corresponding resource has already disappeared, but this lock is not what is actually needed anyway.

Let's start a transaction in a separate session and get the process ID (PID) of the backend:

```
=> BEGIN;
=> SELECT pg_backend_pid();
 pg_backend_pid
----------------
          29131
(1 row)
```

The started transaction holds an exclusive lock on its own virtual ID:

```
=> SELECT locktype, virtualxid, mode, granted
FROM pg_locks WHERE pid = 29131;
  locktype  | virtualxid |     mode      | granted
------------+------------+---------------+---------
 virtualxid | 5/2        | ExclusiveLock | t
(1 row)
```

Here locktype is the type of the lock, virtualxid is the virtual transaction ID (which identifies the locked resource), and mode is the locking mode (exclusive in this case). The granted flag shows whether the requested lock has been acquired.

Once the transaction gets a real ID, the corresponding lock is added to this list:

```
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
             139461
(1 row)
```

```
=> SELECT locktype, virtualxid, transactionid AS xid, mode, granted
FROM pg_locks WHERE pid = 29131;
   locktype    | virtualxid |  xid   |     mode      | granted
---------------+------------+--------+---------------+---------
 virtualxid    | 5/2        |        | ExclusiveLock | t
 transactionid |            | 139461 | ExclusiveLock | t
(2 rows)
```

Now this transaction holds exclusive locks on both its IDs.

12.4 Relation-Level Locks

PostgreSQL provides as many as eight modes in which a relation (a table, an index, or any other object) can be locked.[1] Such a variety allows you to maximize the number of concurrent commands that can be run on a relation.

The next page shows the compatibility matrix extended with examples of commands that require the corresponding locking modes. There is no point in memorizing all these modes or trying to find the logic behind their naming, but it is

[1] postgresql.org/docs/14/explicit-locking#LOCKING-TABLES.html

definitely useful to look through this data, draw some general conclusions, and refer to this table as required.

	AS	RS	RE	SUE	S	SRE	E	AE	
Access Share								×	SELECT
Row Share							×	×	SELECT FOR UPDATE/SHARE
Row Exclusive					×	×	×	×	INSERT, UPDATE, DELETE
Share Update Exclusive				×	×	×	×	×	VACUUM, CREATE INDEX CONCURRENTLY
Share			×	×		×	×	×	CREATE INDEX
Share Row Exclusive			×	×	×	×	×	×	CREATE TRIGGER
Exclusive		×	×	×	×	×	×	×	REFRESH MAT. VIEW CONCURRENTLY
Access Exclusive	×	×	×	×	×	×	×	×	DROP, TRUNCATE, VACUUM FULL, LOCK TABLE, REFRESH MAT. VIEW

The Access Share mode is the weakest one; it can be used with any other mode except Access Exclusive, which is incompatible with all the modes. Thus, a SELECT command can be run in parallel with almost any operation, but it does not let you drop a table that is being queried.

The first four modes allow concurrent heap modifications, while the other four do not. For example, the CREATE INDEX command uses the Share mode, which is compatible with itself (so you can create several indexes on a table concurrently) and with the modes used by read-only operations. As a result, SELECT commands can run in parallel with index creation, while INSERT, UPDATE, and DELETE commands will be blocked.

Conversely, unfinished transactions that modify heap data block the CREATE INDEX command. Instead, you can call CREATE INDEX CONCURRENTLY, which uses a weaker Share Update Exclusive mode: it takes longer to create an index (and this operation can even fail), but in return, concurrent data updates are allowed.

The ALTER TABLE command has multiple flavors that use different locking modes (Share Update Exclusive, Share Row Exclusive, Access Exclusive). All of them are described in the documentation.[1]

[1] postgresql.org/docs/14/sql-altertable.html

Examples in this part of the book rely on the accounts table again:

```
=> TRUNCATE accounts;
=> INSERT INTO accounts(id, client, amount)
VALUES
  (1, 'alice',   100.00),
  (2, 'bob',     200.00),
  (3, 'charlie', 300.00);
```

We will have to access the pg_locks table more than once, so let's create a view that shows all IDs in a single column, thus making the output more concise:

```
=> CREATE VIEW locks AS
SELECT pid,
  locktype,
  CASE locktype
    WHEN 'relation' THEN relation::regclass::text
    WHEN 'transactionid' THEN transactionid::text
    WHEN 'virtualxid' THEN virtualxid
  END AS lockid,
  mode,
  granted
FROM pg_locks
ORDER BY 1, 2, 3;
```

The transaction that is still running in the first session updates a row. This operation locks the accounts table and all its indexes, which results in two new locks of the relation type acquired in the Row Exclusive mode:

```
|    => UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```

```
=> SELECT locktype, lockid, mode, granted
FROM locks WHERE pid = 29131;
   locktype    |    lockid     |       mode       | granted
---------------+---------------+------------------+---------
 relation      | accounts      | RowExclusiveLock | t
 relation      | accounts_pkey | RowExclusiveLock | t
 transactionid | 139461        | ExclusiveLock    | t
 virtualxid    | 5/2           | ExclusiveLock    | t
(4 rows)
```

12.5 Wait Queue

Heavyweight locks form a fair wait queue.[1] A process joins the queue if it attempts to acquire a lock that is incompatible either with the current lock or with the locks requested by other processes already in the queue.

While the first session is working on an update, let's try to create an index on this table in another session:

```
=> SELECT pg_backend_pid();
 pg_backend_pid
----------------
          29610
(1 row)
=> CREATE INDEX ON accounts(client);
```

The command hangs, waiting for the resource to be released. The transaction tries to lock the table in the Share mode but cannot do it:

```
=> SELECT locktype, lockid, mode, granted
FROM locks WHERE pid = 29610;
  locktype  |  lockid  |     mode      | granted
------------+----------+---------------+---------
 relation   | accounts | ShareLock     | f
 virtualxid | 6/3      | ExclusiveLock | t
(2 rows)
```

Now let the third session start the VACUUM FULL command. It will also join the queue because it requires the Access Exclusive mode, which conflicts with all the other modes:

```
=> SELECT pg_backend_pid();
 pg_backend_pid
----------------
          29813
(1 row)
=> VACUUM FULL accounts;
```

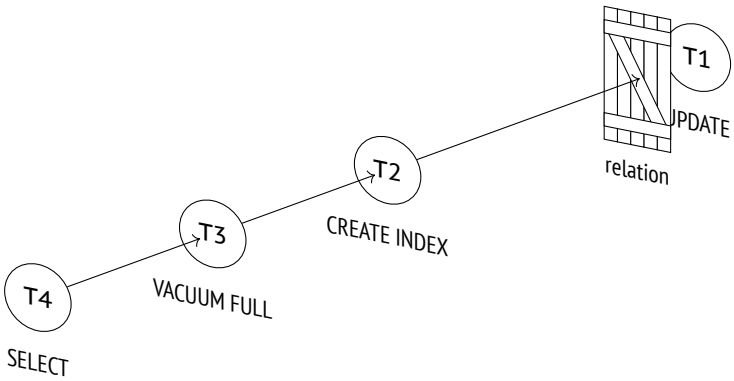
[1] backend/storage/lmgr/lock.c, LockAcquire function

```
=> SELECT locktype, lockid, mode, granted
FROM locks WHERE pid = 29813;
   locktype    |  lockid  |        mode         | granted
---------------+----------+---------------------+---------
 relation      | accounts | AccessExclusiveLock | f
 transactionid | 139465   | ExclusiveLock       | t
 virtualxid    | 7/4      | ExclusiveLock       | t
(3 rows)
```

All the subsequent contenders will now have to join the queue, regardless of their locking mode. Even simple SELECT queries will honestly follow VACUUM FULL, although they are compatible with the Row Exclusive lock held by the first session performing the update.

```
=> SELECT pg_backend_pid();
 pg_backend_pid
----------------
          30023
(1 row)
=> SELECT * FROM accounts;
```

```
=> SELECT locktype, lockid, mode, granted
FROM locks WHERE pid = 30023;
  locktype  |  lockid  |      mode       | granted
------------+----------+-----------------+---------
 relation   | accounts | AccessShareLock | f
 virtualxid | 8/3      | ExclusiveLock   | t
(2 rows)
```



The pg_blocking_pids function gives a high-level overview of all waits. It shows the IDs of all processes queued before the specified one that are already holding or would like to acquire an incompatible lock: v. 9.6

```
=> SELECT pid,
  pg_blocking_pids(pid),
  wait_event_type,
  state,
  left(query,50) AS query
FROM pg_stat_activity
WHERE pid IN (29131,29610,29813,30023) \gx
-[ RECORD 1 ]----+--------------------------------------------------
pid              | 29131
pg_blocking_pids | {}
wait_event_type  | Client
state            | idle in transaction
query            | UPDATE accounts SET amount = amount + 100.00 WHERE
-[ RECORD 2 ]----+--------------------------------------------------
pid              | 29610
pg_blocking_pids | {29131}
wait_event_type  | Lock
state            | active
query            | CREATE INDEX ON accounts(client);
-[ RECORD 3 ]----+--------------------------------------------------
pid              | 29813
pg_blocking_pids | {29131,29610}
wait_event_type  | Lock
state            | active
query            | VACUUM FULL accounts;
-[ RECORD 4 ]----+--------------------------------------------------
pid              | 30023
pg_blocking_pids | {29813}
wait_event_type  | Lock
state            | active
query            | SELECT * FROM accounts;
```

To get more details, you can review the information provided in the pg_locks table.[1]

Once the transaction is completed (either committed or aborted), all its locks are released.[2] The first process in the queue gets the requested lock and wakes up.

[1] wiki.postgresql.org/wiki/Lock_dependency_information
[2] backend/storage/lmgr/lock.c, LockReleaseAll & LockRelease functions

Here the transaction commit in the first session leads to sequential execution of all the queued processes:

```
=> ROLLBACK;
ROLLBACK

CREATE INDEX

VACUUM

 id | client  | amount
----+---------+--------
  1 | alice   | 100.00
  2 | bob     | 200.00
  3 | charlie | 300.00
(3 rows)
```

13

Row-Level Locks

13.1 Lock Design

Thanks to snapshot isolation, heap tuples do not have to be locked for reading. However, two write transactions must not be allowed to modify one and the same row at the same time. Rows must be locked in this case, but heavyweight locks are not a very good choice for this purpose: each of them takes space in the server's shared memory (hundreds of bytes, not to mention all the supporting infrastructure), and PostgreSQL internal mechanisms are not designed to handle a huge number of concurrent heavyweight locks.

Some database systems solve this problem by lock escalation: if row-level locks are too many, they are replaced by a single lock of finer granularity (for example, by a page-level or table-level lock). It simplifies the implementation, but can greatly limit system throughput.

In PostgreSQL, the information on whether a particular row is locked is kept only in the header of its current heap tuple. Row-level locks are virtually attributes in heap pages rather than actual locks, and they are not reflected in RAM in any way.

A row is typically locked when it is being updated or deleted. In both cases, the current version of the row is marked as deleted. The attribute used for this purpose is the current transaction's ID specified in the xmax field, and it is the same ID (combined with additional hint bits) that indicates that the row is locked. If a transaction wants to modify a row but sees an active transaction ID in the xmax field of its current version, it has to wait for this transaction to complete. Once it is over, all the locks are released, and the waiting transaction can proceed. *p. 81*

This mechanism allows locking as many rows as required at no extra cost.

The downside of this solution is that other processes cannot form a queue, as RAM contains no information about such locks. Therefore, heavyweight locks are still required: a process waiting for a row to be released requests a lock on the ID of the transaction currently busy with this row. Once the transaction completes, the row becomes available again. Thus, the number of heavyweight locks is proportional to the number of concurrent processes rather than rows being modified.

13.2 Row-Level Locking Modes

Row-level locks support four modes.[1] Two of them implement exclusive locks that can be acquired by only one transaction at a time, while the other two provide shared locks that can be held by several transactions simultaneously.

Here is the compatibility matrix of these modes:

	Key Share	Share	No Key Update	Update
Key Share				×
Share			×	×
No Key Update		×	×	×
Update	×	×	×	×

Exclusive Modes

The Update mode allows modifying any tuple fields and even deleting the whole tuple, while the No Key Update mode permits only those changes that do not involve any fields related to unique indexes (in other words, foreign keys must not be affected).

The UPDATE command automatically chooses the weakest locking mode possible; keys usually remain unchanged, so rows are typically locked in the No Key Update mode.

[1] postgresql.org/docs/14/explicit-locking#LOCKING-ROWS.html

Let's create a function that uses pageinspect to display some tuple metadata that we are interested in, namely the xmax field and several hint bits:

```
=> CREATE FUNCTION row_locks(relname text, pageno integer)
RETURNS TABLE(
  ctid tid, xmax text,
  lock_only text, is_multi text,
  keys_upd text, keyshr text,
  shr text
)
AS $$
SELECT (pageno,lp)::text::tid,
  t_xmax,
  CASE WHEN t_infomask  & 128   = 128   THEN 't' END,
  CASE WHEN t_infomask  & 4096  = 4096  THEN 't' END,
  CASE WHEN t_infomask2 & 8192  = 8192  THEN 't' END,
  CASE WHEN t_infomask  & 16    = 16    THEN 't' END,
  CASE WHEN t_infomask  & 16+64 = 16+64 THEN 't' END
FROM heap_page_items(get_raw_page(relname,pageno))
ORDER BY lp;
$$ LANGUAGE sql;
```

Now start a transaction on the accounts table to update the balance of the first account (the key remains the same) and the ID of the second account (the key gets updated):

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
=> UPDATE accounts SET id = 20 WHERE id = 2;
```

The page now contains the following metadata:

```
=> SELECT * FROM row_locks('accounts',0) LIMIT 2;
 ctid  |  xmax  | lock_only | is_multi | keys_upd | keyshr | shr
-------+--------+-----------+----------+----------+--------+-----
 (0,1) | 139470 |           |          |          |        |
 (0,2) | 139470 |           |          | t        |        |
(2 rows)
```

The locking mode is defined by the keys_updated hint bit.

```
=> ROLLBACK;
```

The SELECT FOR command uses the same xmax field as a locking attribute, but in this case the xmax_lock_only hint bit must also be set. This bit indicates that the tuple is locked but not deleted, which means that it is still current:

```
=> BEGIN;
=> SELECT * FROM accounts WHERE id = 1 FOR NO KEY UPDATE;
=> SELECT * FROM accounts WHERE id = 2 FOR UPDATE;
=> SELECT * FROM row_locks('accounts',0) LIMIT 2;
 ctid  |  xmax  | lock_only | is_multi | keys_upd | keyshr | shr
-------+--------+-----------+----------+----------+--------+-----
 (0,1) | 139471 | t         |          |          |        |
 (0,2) | 139471 | t         |          | t        |        |
(2 rows)
=> ROLLBACK;
```

Shared Modes

The Share mode can be applied when a row needs to be read, but its modification by another transaction must be forbidden. The Key Share mode allows updating any tuple fields except key attributes.

Of all the shared modes, the PostgreSQL core uses only Key Share, which is applied when foreign keys are being checked. Since it is compatible with the No Key Update exclusive mode, foreign key checks do not interfere with concurrent updates of non-key attributes. As for applications, they can use any shared modes they like.

Let me stress once again that simple SELECT commands never use row-level locks.

```
=> BEGIN;
=> SELECT * FROM accounts WHERE id = 1 FOR KEY SHARE;
=> SELECT * FROM accounts WHERE id = 2 FOR SHARE;
```

Here is what we see in the heap tuples:

```
=> SELECT * FROM row_locks('accounts',0) LIMIT 2;
 ctid  |  xmax  | lock_only | is_multi | keys_upd | keyshr | shr
-------+--------+-----------+----------+----------+--------+-----
 (0,1) | 139472 | t         |          |          | t      |
 (0,2) | 139472 | t         |          |          | t      | t
(2 rows)
```

The xmax_keyshr_lock bit is set for both operations, but you can recognize the Share mode by other hint bits.[1]

13.3 Multitransactions

As we have seen, the locking attribute is represented by the xmax field, which is set to the ID of the transaction that has acquired the lock. So how is this attribute set for a shared lock held by several transactions at a time?

When dealing with shared locks, PostgreSQL applies so-called *multitransactions* (multixacts).[2] A multitransaction is a group of transactions that is assigned a separate ID. Detailed information on group members and their locking modes is stored in files under the PGDATA/pg_multixact directory. For faster access, locked pages are cached in the shared memory of the server;[3] all changes are logged to ensure fault tolerance.

Multixact IDs have the same 32-bit length as regular transaction IDs, but they are issued independently. It means that transactions and multitransactions can potentially have the same IDs. To differentiate between the two, PostgreSQL uses an additional hint bit: xmax_is_multi.

Let's add one more exclusive lock acquired by another transaction (Key Share and No Key Update modes are compatible):

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```

```
=> SELECT * FROM row_locks('accounts',0) LIMIT 2;
 ctid  |  xmax  | lock_only | is_multi | keys_upd | keyshr | shr
-------+--------+-----------+----------+----------+--------+-----
 (0,1) | 1      |           | t        |          |        |
 (0,2) | 139472 | t         |          |          | t      | t
(2 rows)
```

[1] include/access/htup_details.h
[2] backend/access/transam/multixact.c
[3] backend/access/transam/slru.c

The xmax_is_multi bit shows that the first row uses a multitransaction ID instead of a regular one.

Without going into further implementation details, let's display the information on all the possible row-level locks using the pgrowlocks extension:

```
=> CREATE EXTENSION pgrowlocks;
=> SELECT * FROM pgrowlocks('accounts') \gx
-[ RECORD 1 ]----------------------------
locked_row | (0,1)
locker     | 1
multi      | t
xids       | {139472,139473}
modes      | {"Key Share","No Key Update"}
pids       | {30574,30874}
-[ RECORD 2 ]----------------------------
locked_row | (0,2)
locker     | 139472
multi      | f
xids       | {139472}
modes      | {"For Share"}
pids       | {30574}
```

It looks a lot like querying the pg_locks view, but the pgrowlocks function has to access heap pages, as RAM contains no information on row-level locks.

```
=> COMMIT;
```

```
=> ROLLBACK;
```

p. 143 Since multixact IDs are 32-bit, they are subject to wraparound because of counter limits, just like regular transaction IDs. Therefore, PostgreSQL has to process multixact IDs in a way similar to freezing: old multixact IDs are replaced with new ones (or with a regular transaction ID if only one transaction is holding the lock by that time).[1]

But while regular transaction IDs are frozen only in the xmin field (as a non-empty xmax indicates that the tuple is outdated and will soon be removed), it is the xmax field that has to be frozen for multitransactions: the current row version may be repeatedly locked by new transactions in a shared mode.

[1] backend/access/heap/heapam.c, FreezeMultiXactId function

Freezing of multitransactions can be managed by server parameters, which are similar to those provided for regular freezing: *vacuum_multixact_freeze_min_age*, *vacuum_multixact_freeze_table_age*, *autovacuum_multixact_freeze_max_age*, as well as *vacuum_multixact_failsafe_age*. v. 14

13.4 Wait Queue

Exclusive Modes

Since a row-level lock is just an attribute, the queue is arranged in a not-so-trivial way. When a transaction is about to modify a row, it has to follow these steps:[1]

1 If the xmax field and the hint bits indicate that the row is locked in an incompatible mode, acquire an exclusive heavyweight lock on the tuple that is being modified.

2 If necessary, wait until all the incompatible locks are released by requesting a lock on the ID of the xmax transaction (or several transactions if xmax contains a mutixact ID).

3 Write its own ID into xmax in the tuple header and set the required hint bits.

4 Release the tuple lock if it was acquired in the first step.

A *tuple* lock is yet another kind of heavyweight locks, which has the tuple type (not to be confused with a regular row-level lock).

It may seem that steps 1 and 4 are redundant and it is enough to simply wait until all the locking transactions are over. However, if several transactions are trying to update one and the same row, all of them will be waiting on the transaction currently processing this row. Once it completes, they will find themselves in a race condition for the right to lock the row, and some "unlucky" transactions may have to wait for an indefinitely long time. Such a situation is called *resource starvation*.

A tuple lock identifies the first transaction in the queue and guarantees that it will be the next one to get the lock.

[1] backend/access/heap/README.tuplock

But you can see it for yourself. Since PostgreSQL acquires many different locks during its operation, and each of them is reflected in a separate row in the pg_locks table, I am going to create yet another view on top of pg_locks. It will show this information in a more concise form, keeping only those locks that we are currently interested in (the ones related to the accounts table and to the transaction itself, except for any locks on virtual IDs):

```
=> CREATE VIEW locks_accounts AS
SELECT pid,
  locktype,
  CASE locktype
    WHEN 'relation' THEN relation::regclass::text
    WHEN 'transactionid' THEN transactionid::text
    WHEN 'tuple' THEN relation::regclass||'('||page||','||tuple||')'
  END AS lockid,
  mode,
  granted
FROM pg_locks
WHERE locktype in ('relation','transactionid','tuple')
  AND (locktype != 'relation' OR relation = 'accounts'::regclass)
ORDER BY 1, 2, 3;
```

Let's start the first transaction and update a row:

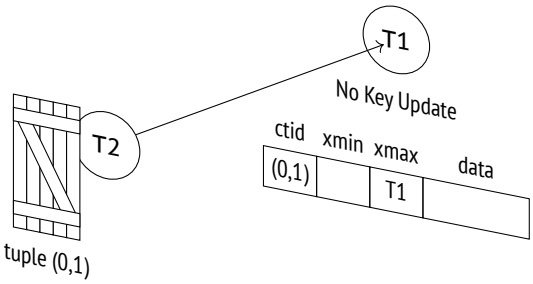
```
=> BEGIN;
=> SELECT txid_current(), pg_backend_pid();
 txid_current | pg_backend_pid
--------------+----------------
       139475 |          30874
(1 row)
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```

The transaction has completed all the four steps of the workflow and is now holding a lock on the table:

```
=> SELECT * FROM locks_accounts WHERE pid = 30874;
  pid  |   locktype    |  lockid  |       mode       | granted
-------+---------------+----------+------------------+---------
 30874 | relation      | accounts | RowExclusiveLock | t
 30874 | transactionid | 139475   | ExclusiveLock    | t
(2 rows)
```

Start the second transaction and try to update the same row. The transaction will hang, waiting on a lock:

```
=> BEGIN;
=> SELECT txid_current(), pg_backend_pid();
 txid_current | pg_backend_pid
--------------+----------------
       139476 |          30945
(1 row)
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```



The second transaction only gets as far as the second step. For this reason, apart from locking the table and its own ID, it adds two more locks, which are also reflected in the pg_locks view: the tuple lock acquired at the first step and the lock of the ID of the second transaction requested at the second step:

```
=> SELECT * FROM locks_accounts WHERE pid = 30945;
  pid  |   locktype    |    lockid     |       mode       | granted
-------+---------------+---------------+------------------+---------
 30945 | relation      | accounts      | RowExclusiveLock | t
 30945 | transactionid | 139475        | ShareLock        | f
 30945 | transactionid | 139476        | ExclusiveLock    | t
 30945 | tuple         | accounts(0,1) | ExclusiveLock    | t
(4 rows)
```

The third transaction will get stuck on the first step. It will try to acquire a lock on the tuple and will stop at this point:

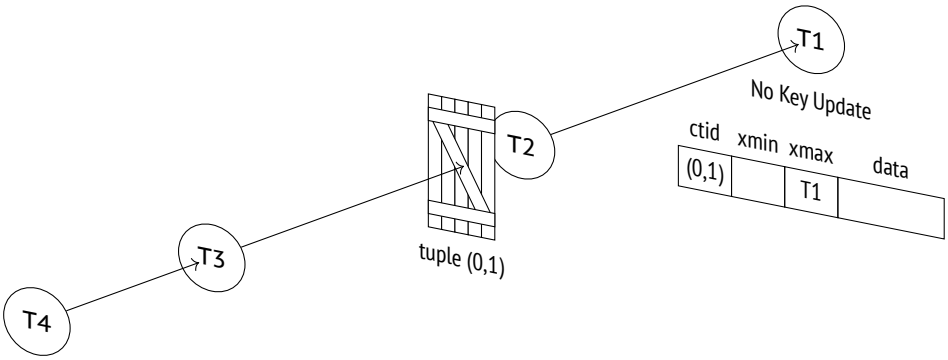
```
=> BEGIN;
=> SELECT txid_current(), pg_backend_pid();
 txid_current | pg_backend_pid
--------------+----------------
       139477 |          31016
(1 row)
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```

```
=> SELECT * FROM locks_accounts WHERE pid = 31016;
  pid  |   locktype    |    lockid     |       mode       | granted
-------+---------------+---------------+------------------+---------
 31016 | relation      | accounts      | RowExclusiveLock | t
 31016 | transactionid | 139477        | ExclusiveLock    | t
 31016 | tuple         | accounts(0,1) | ExclusiveLock    | f
(3 rows)
```

The fourth and all the subsequent transactions trying to update this row will not differ from the third transaction in this respect: all of them will be waiting on the same tuple lock.

```
=> BEGIN;
=> SELECT txid_current(), pg_backend_pid();
 txid_current | pg_backend_pid
--------------+----------------
       139478 |          31087
(1 row)
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```

```
=> SELECT * FROM locks_accounts WHERE pid = 31016;
  pid  |   locktype    |    lockid     |       mode       | granted
-------+---------------+---------------+------------------+---------
 31016 | relation      | accounts      | RowExclusiveLock | t
 31016 | transactionid | 139477        | ExclusiveLock    | t
 31016 | tuple         | accounts(0,1) | ExclusiveLock    | f
(3 rows)
```



To get the full picture of the current waits, you can extend the pg_stat_activity view with the information on locking processes:

```
=> SELECT pid,
  wait_event_type,
  wait_event,
  pg_blocking_pids(pid)
FROM pg_stat_activity
WHERE pid IN (30874,30945,31016,31087);
  pid  | wait_event_type |  wait_event   | pg_blocking_pids
-------+-----------------+---------------+------------------
 30874 | Client          | ClientRead    | {}
 30945 | Lock            | transactionid | {30874}
 31016 | Lock            | tuple         | {30945}
 31087 | Lock            | tuple         | {30945,31016}
(4 rows)
```

If the first transaction is aborted, everything will work as expected: all the subsequent transactions will move one step further without jumping the queue.

And yet it is more likely that the first transaction will be committed. At the Repeatable Read or Serializable isolation levels, it would result in a serialization failure, so the second transaction would have to be aborted[1] (and all the subsequent transactions in the queue would get aborted too). But at the Read Committed isolation level the modified row will be re-read, and its update will be retried.

So, the first transaction is committed:

```
|    => COMMIT;
```

The second transaction wakes up and successfully completes the third and the fourth steps of the workflow:

```
||    UPDATE 1

=> SELECT * FROM locks_accounts WHERE pid = 30945;
  pid  |   locktype    |  lockid  |       mode       | granted
-------+---------------+----------+------------------+---------
 30945 | relation      | accounts | RowExclusiveLock | t
 30945 | transactionid | 139476   | ExclusiveLock    | t
(2 rows)
```

As soon as the second transaction releases the tuple lock, the third one also wakes up, but it sees that the xmax field of the new tuple contains a different ID already.

[1] backend/executor/nodeModifyTable.c, ExecUpdate function

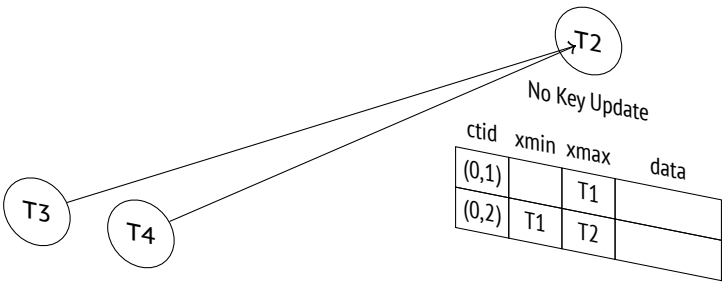
At this point, the above workflow is over. At the Read Committed isolation level, one more attempt to lock the row is performed,[1] but it does not follow the outlined steps. The third transaction is now waiting for the second one to complete without trying to acquire a tuple lock:

```
=> SELECT * FROM locks_accounts WHERE pid = 31016;
  pid  |   locktype    |  lockid  |       mode       | granted
-------+---------------+----------+------------------+---------
 31016 | relation      | accounts | RowExclusiveLock | t
 31016 | transactionid | 139476   | ShareLock        | f
 31016 | transactionid | 139477   | ExclusiveLock    | t
(3 rows)
```

The fourth transaction does the same:

```
=> SELECT * FROM locks_accounts WHERE pid = 31087;
  pid  |   locktype    |  lockid  |       mode       | granted
-------+---------------+----------+------------------+---------
 31087 | relation      | accounts | RowExclusiveLock | t
 31087 | transactionid | 139476   | ShareLock        | f
 31087 | transactionid | 139478   | ExclusiveLock    | t
(3 rows)
```

Now both the third and the fourth transactions are waiting for the second one to complete, risking to get into a race condition. The queue has virtually fallen apart.



If other transactions had joined the queue while it still existed, all of them would have been dragged into this race.

[1] backend/access/heap/heapam_handler.c, heapam_tuple_lock function

Conclusion: it is not a good idea to update one and the same table row in multiple concurrent processes. Under high load, this hotspot can quickly turn into a bottleneck that causes performance issues.

Let's commit all the started transactions.

```
||      => COMMIT;

|||      UPDATE 1
|||      => COMMIT;

||||      UPDATE 1
||||      => COMMIT;
```

Shared Modes

PostgreSQL acquires shared locks only for referential integrity checks. Using them in a high-load application can lead to resource starvation, and a two-level locking model cannot prevent such an outcome.

Let's recall the steps a transaction should take to lock a row:

1. If the xmax field and hint bits indicate that the row is locked in the *exclusive* mode, acquire an exclusive heavyweight tuple lock.
2. If required, wait for all the *incompatible* locks to be released by requesting a lock on the ID of the xmax transaction (or several transactions if xmax contains a multixact ID).
3. Write its own ID into xmax in the tuple header and set the required hint bits.
4. Release the tuple lock if it was acquired in the first step.

The first two steps imply that if the locking modes are *compatible*, the transaction will *jump the queue*.

Let's repeat our experiment from the very beginning.

```
=> TRUNCATE accounts;
```

```
=> INSERT INTO accounts(id, client, amount)
VALUES
  (1,'alice',100.00),
  (2,'bob',200.00),
  (3,'charlie',300.00);
```

Start the first transaction:

```
=> BEGIN;
=> SELECT txid_current(), pg_backend_pid();
 txid_current | pg_backend_pid
--------------+----------------
       139481 |          30874
(1 row)
```

The row is now locked in a shared mode:

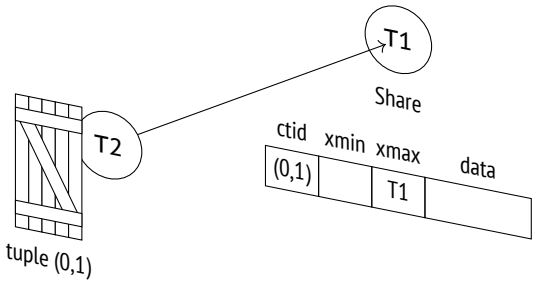
```
=> SELECT * FROM accounts WHERE id = 1 FOR SHARE;
```

The second transaction tries to update the same row, but it is not allowed: Share and No Key Update modes are incompatible:

```
=> BEGIN;
=> SELECT txid_current(), pg_backend_pid();
 txid_current | pg_backend_pid
--------------+----------------
       139482 |          30945
(1 row)
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```

Waiting for the first transaction to complete, the second transaction is holding the tuple lock, just like in the previous example:

```
=> SELECT * FROM locks_accounts WHERE pid = 30945;
  pid  |   locktype    |    lockid     |       mode       | granted
-------+---------------+---------------+------------------+---------
 30945 | relation      | accounts      | RowExclusiveLock | t
 30945 | transactionid | 139481        | ShareLock        | f
 30945 | transactionid | 139482        | ExclusiveLock    | t
 30945 | tuple         | accounts(0,1) | ExclusiveLock    | t
(4 rows)
```

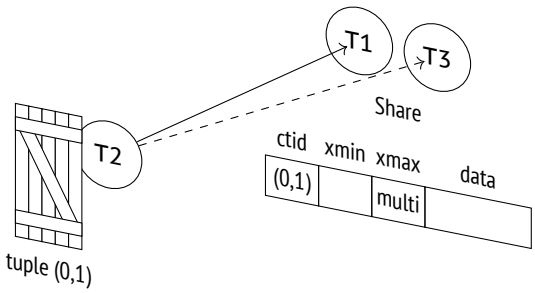


Now let the third transaction lock the row in a shared mode. Such a lock is compatible with the already acquired lock, so this transaction jumps the queue:

```
=> BEGIN;
=> SELECT txid_current(), pg_backend_pid();
 txid_current | pg_backend_pid
--------------+----------------
       139483 |          31016
(1 row)
=> SELECT * FROM accounts WHERE id = 1 FOR SHARE;
```

We have got two transactions locking the same row:

```
=> SELECT * FROM pgrowlocks('accounts') \gx
-[ RECORD 1 ]---------------
locked_row | (0,1)
locker     | 2
multi      | t
xids       | {139481,139483}
modes      | {Share,Share}
pids       | {30874,31016}
```



If the first transaction completes at this point, the second one will wake up to see that the row is still locked and will get back to the queue—but this time it will find itself behind the third transaction:

```
|    => COMMIT;
```

```
=> SELECT * FROM locks_accounts WHERE pid = 30945;
  pid  |   locktype    |    lockid     |       mode       | granted
-------+---------------+---------------+------------------+---------
 30945 | relation      | accounts      | RowExclusiveLock | t
 30945 | transactionid | 139482        | ExclusiveLock    | t
 30945 | transactionid | 139483        | ShareLock        | f
 30945 | tuple         | accounts(0,1) | ExclusiveLock    | t
(4 rows)
```

And only when the third transaction completes will the second one be able to perform an update (unless other shared locks appear within this time interval).

```
|||     => COMMIT;
```

```
||     UPDATE 1
||     => COMMIT;
```

Foreign key checks are unlikely to cause any issues, as key attributes usually remain unchanged and Key Share can be used together with No Key Update. But in most cases, you should avoid shared row-level locks in applications.

13.5 No-Wait Locks

SQL commands usually wait for the requested resources to be freed. But sometimes it makes sense to cancel the operation if the lock cannot be acquired immediately. For this purpose, commands like SELECT, LOCK, and ALTER offer the NOWAIT clause.

Let's lock a row:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```

The command with the NOWAIT clause immediately completes with an error if the requested resource is locked:

```
=> SELECT * FROM accounts
FOR UPDATE NOWAIT;
ERROR:  could not obtain lock on row in relation "accounts"
```

Such an error can be captured and handled by the application code.

The UPDATE and DELETE commands do not have the NOWAIT clause. Instead, you can try to lock the row using the SELECT FOR UPDATE NOWAIT command and then update or delete it if the attempt is successful.

In some rare cases, it may be convenient to skip the already locked rows and start processing the available ones right away. This is exactly what SELECT FOR does when run with the SKIP LOCKED clause:

```
=> SELECT * FROM accounts
ORDER BY id
FOR UPDATE SKIP LOCKED
LIMIT 1;
 id | client | amount
----+--------+--------
  2 | bob    | 200.00
(1 row)
```

In this example, the first (already locked) row was skipped, and the query locked and returned the second row.

This approach enables us to process rows in batches or set up parallel processing of event queues. However, avoid inventing other use cases for this command—most tasks can be addressed using much simpler methods. p. 164

Last but not least, you can avoid long waits by setting a timeout:

```
=> SET lock_timeout = '1s';
=> ALTER TABLE accounts DROP COLUMN amount;
ERROR:  canceling statement due to lock timeout
```

The command completes with an error because it has failed to acquire a lock within one second. A timeout can be set not only at the session level, but also at lower levels, for example, for a particular transaction.

This method prevents long waits during table processing when the command requiring an exclusive lock is executed under load. If an error occurs, this command can be retried after a while.

> While *statement_timeout* limits the total time of operator execution, the *lock_timeout* parameter defines the maximum time that can be spent waiting on a lock.

```
=> ROLLBACK;
```

13.6 Deadlocks

A transaction may sometimes require a resource that is currently being used by another transaction, which, in its turn, may be waiting on a resource locked by the third transaction, and so on. Such transactions get queued using heavyweight locks.

But occasionally a transaction already in the queue may need yet another resource, so it has to join the same queue again and wait for this resource to be released. A *deadlock*[1] occurs: the queue now has a circular dependency that cannot resolve on its own.

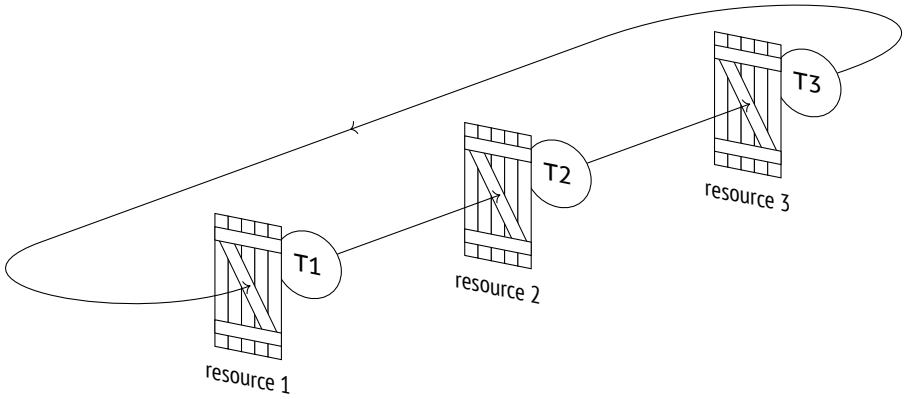
For better visualization, let's draw a wait-for graph. Its nodes represent active processes, while the edges shown as arrows point from the processes waiting on locks to the processes holding these locks. If the graph has a *cycle*, that is, a node can reach itself following the arrows, it means that a deadlock has occurred.

> The illustrations here show transactions rather than processes. This substitution is usually acceptable because one transaction is executed by one process, and locks can only be acquired within a transaction. But in general, it is more correct to talk about processes, as some locks may not be released right away when the transaction is complete.

If a deadlock has occurred, and none of its participants has set a timeout, transactions will be waiting on each other forever. That's why the lock manager[2] performs automatic deadlock detection.

However, this check requires some effort, which should not be wasted each time a lock is requested (after all, deadlocks do not happen too often). So if the process

[1] postgresql.org/docs/14/explicit-locking#LOCKING-DEADLOCKS.html
[2] backend/storage/lmgr/README



makes an unsuccessful attempt to acquire a lock and falls asleep after joining the queue, PostgreSQL automatically sets a timeout as defined by the *deadlock_timeout* parameter.[1] If the resource becomes available earlier—great, then the extra cost of the check will be avoided. But if the wait continues after the *deadlock_timeout* units of time, the waiting process wakes up and initiates the check.[2]

1s

This check effectively consists in building a wait-for graph and searching it for cycles.[3] To "freeze" the current state of the graph, PostgreSQL stops any processing of heavyweight locks for the whole duration of the check.

If no deadlocks are detected, the process falls asleep again; sooner or later its turn will come.

If a deadlock is detected, one of the transactions will be forced to terminate, thus releasing its locks and enabling other transactions to continue their execution. In most cases, it is the transaction initiating the check that gets interrupted, but if the cycle includes an autovacuum process that is not currently freezing tuples to prevent wraparound, the server terminates autovacuum as having lower priority.

Deadlocks usually indicate bad application design. To discover such situations, you have two things to watch out for: the corresponding messages in the server log and an increasing deadlocks value in the pg_stat_database table.

[1] backend/storage/lmgr/proc.c, ProcSleep function
[2] backend/storage/lmgr/proc.c, CheckDeadLock function
[3] backend/storage/lmgr/deadlock.c

Deadlocks by Row Updates

Although deadlocks are ultimately caused by heavyweight locks, it is mostly row-level locks acquired in different order that lead to them.

Suppose a transaction is going to transfer $100 between two accounts. It starts by drawing this sum from the first account:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount - 100.00 WHERE id = 1;
UPDATE 1
```

At the same time, another transaction is going to transfer $10 from the second account to the first one. It begins by drawing this sum from the second account:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount - 10.00 WHERE id = 2;
UPDATE 1
```

Now the first transaction attempts to increase the amount in the second account but sees that the corresponding row is locked:

```
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 2;
```

Then the second transaction tries to update the first account but also gets locked:

```
=> UPDATE accounts SET amount = amount + 10.00 WHERE id = 1;
```

This circular wait will never resolve on its own. Unable to obtain the resource within one second, the first transaction initiates a deadlock check and gets aborted by the server:

```
ERROR:  deadlock detected
DETAIL:  Process 30574 waits for ShareLock on transaction 139489;
blocked by process 30874.
Process 30874 waits for ShareLock on transaction 139488; blocked by
process 30574.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,2) in relation "accounts"
```

Now the second transaction can continue. It wakes up and performs an update:

```
|    UPDATE 1
```

Let's complete the transactions.

```
|    => ROLLBACK;
```

```
=> ROLLBACK;
```

The right way to perform such operations is to lock resources in the same order. For example, in this particular case the accounts could have been locked in ascending order based on their numbers.

Deadlocks Between Two UPDATE Statements

In some cases deadlocks seem impossible, and yet they do occur.

We usually assume that SQL commands are atomic, but are they really? Let's take a closer look at UPDATE: this command locks rows as they are being updated rather than all at once, and it does not happen simultaneously. So if one UPDATE command modifies several rows in one order while the other is doing the same in a different order, a deadlock can occur.

Let's reproduce this scenario. First, we are going to build an index on the amount column, in descending order:

```
=> CREATE INDEX ON accounts(amount DESC);
```

To be able to observe the process, we can write a function that slows things down:

```
=> CREATE FUNCTION inc_slow(n numeric)
RETURNS numeric
AS $$
  SELECT pg_sleep(1);
  SELECT n + 100.00;
$$ LANGUAGE sql;
```

The first UPDATE command is going to update all the tuples. The execution plan relies on a sequential scan of the whole table.

```
=> EXPLAIN (costs off)
UPDATE accounts SET amount = inc_slow(amount);
        QUERY PLAN
--------------------------
 Update on accounts
   -> Seq Scan on accounts
(2 rows)
```

To make sure that the heap page stores the rows in ascending order based on the amount column, we have to truncate the table and insert the rows anew:

```
=> TRUNCATE accounts;
=> INSERT INTO accounts(id, client, amount)
VALUES
  (1,'alice',100.00),
  (2,'bob',200.00),
  (3,'charlie',300.00);
=> ANALYZE accounts;
=> SELECT ctid, * FROM accounts;
 ctid  | id | client  | amount
-------+----+---------+--------
 (0,1) |  1 | alice   | 100.00
 (0,2) |  2 | bob     | 200.00
 (0,3) |  3 | charlie | 300.00
(3 rows)
```

The sequential scan will update the rows in the same order (it is not always true
p. 179 for large tables though).

Let's start the update:

```
| => UPDATE accounts SET amount = inc_slow(amount);
```

Meanwhile, we are going to forbid sequential scans in another session:

```
|| => SET enable_seqscan = off;
```

As a result, the planner chooses an index scan for the next UPDATE command.

```
|| => EXPLAIN (costs off)
|| UPDATE accounts SET amount = inc_slow(amount)
|| WHERE amount > 100.00;
```

```
                          QUERY PLAN
-------------------------------------------------------
 Update on accounts
   -> Index Scan using accounts_amount_idx on accounts
       Index Cond: (amount > 100.00)
(3 rows)
```

The second and third rows satisfy the condition; since the index is descending, the rows will get updated in the reverse order.

Let's start the next update:

```
=> UPDATE accounts SET amount = inc_slow(amount)
WHERE amount > 100.00;
```

The pgrowlocks extension shows that the first operator has already updated the first row (0,1), while the second one has managed to update the last row (0,3):

```
=> SELECT locked_row, locker, modes FROM pgrowlocks('accounts');
 locked_row | locker |       modes
------------+--------+-------------------
 (0,1)      | 139495 | {"No Key Update"}     ⟵ first
 (0,3)      | 139496 | {"No Key Update"}     ⟵ second
(2 rows)
```

Another second passes. The first operator has updated the second row, and the other one would like to do it too, but it is not allowed.

```
=> SELECT locked_row, locker, modes FROM pgrowlocks('accounts');
 locked_row | locker |       modes
------------+--------+-------------------
 (0,1)      | 139495 | {"No Key Update"}
 (0,2)      | 139495 | {"No Key Update"}     ⟵ the first one wins
 (0,3)      | 139496 | {"No Key Update"}
(3 rows)
```

Now the first operator would like to update the last table row, but it is already locked by the second operator. A deadlock has occurred.

One of the transactions is aborted:

```
ERROR:  deadlock detected
DETAIL:  Process 30945 waits for ShareLock on transaction 139495;
blocked by process 30874.
Process 30874 waits for ShareLock on transaction 139496; blocked by
process 30945.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,2) in relation "accounts"
```

And the other completes its execution:

```
UPDATE 3
```

Although such situations seem impossible, they do occur in high-load systems when batch row updates are performed.

14

Miscellaneous Locks

14.1 Non-Object Locks

To lock a resource that is not considered a *relation*, PostgreSQL uses heavyweight locks of the object type.[1] You can lock almost anything that is stored in the system catalog: tablespaces, subscriptions, schemas, roles, policies, enumerated data types, and so on.

Let's start a transaction that creates a table:

```
=> BEGIN;
=> CREATE TABLE example(n integer);
```

Now take a look at non-relation locks in the pg_locks table:

```
=> SELECT database,
  (
    SELECT datname FROM pg_database WHERE oid = database
  ) AS dbname,
  classid,
  (
    SELECT relname FROM pg_class WHERE oid = classid
  ) AS classname,
  objid,
  mode,
  granted
FROM pg_locks
WHERE locktype = 'object'
  AND pid = pg_backend_pid() \gx
```

[1] backend/storage/lmgr/lmgr.c, LockDatabaseObject & LockSharedObject functions

```
-[ RECORD 1 ]--------------
database  | 16391
dbname    | internals
classid   | 2615
classname | pg_namespace
objid     | 2200
mode      | AccessShareLock
granted   | t
```

The locked resource is defined here by three values:

database — the oid of the database that contains the object being locked (or zero if this object is common to the whole cluster)

classid — the oid listed in pg_class that corresponds to the name of the system catalog table defining the type of the resource

objid — the oid listed in the system catalog table referenced by classid

The database value points to the internals database; it is the database to which the current session is connected. The classid column points to the pg_namespace table, which lists schemas.

Now we can decipher the objid:

```
=> SELECT nspname FROM pg_namespace WHERE oid = 2200;
 nspname
---------
 public
(1 row)
```

Thus, PostgreSQL has locked the public schema to make sure that no one can delete it while the transaction is still running.

Similarly, object deletion requires exclusive locks on both the object itself and all the resources it depends on.[1]

```
=> ROLLBACK;
```

[1] backend/catalog/dependency.c, performDeletion function

14.2 Relation Extension Locks

As the number of tuples in a relation grows, PostgreSQL inserts new tuples into free space in the already available pages whenever possible. But it is clear that at some point it will have to add new pages, that is, to *extend the relation*. In terms of the physical layout, new pages get added to the end of the corresponding file (which, in turn, can lead to creation of a new file).

For new pages to be added by only one process at a time, this operation is protected by a special heavyweight lock of the extend type.[1] Such a lock is also used by index vacuuming to forbid adding new pages during an index scan.

Relation extension locks behave a bit differently from what we have seen so far:

- They are released as soon as the extension is created, without waiting for the transaction to complete.
- They cannot cause a deadlock, so they are not included into the wait-for graph.

  However, a deadlock check will still be performed if the procedure of extending a relation is taking longer than *deadlock_timeout*. It is not a typical situation, but it can happen if a large number of processes perform multiple insertions concurrently. In this case, the check can be called multiple times, virtually paralyzing normal system operation.

  To minimize this risk, heap files are extended by several pages at once (in proportion to the number of processes awaiting the lock, but by not more than 512 pages per operation).[2] An exception to this rule is B-tree index files, which are extended by one page at a time.[3] v. 9.6

14.3 Page Locks

A page-level heavyweight lock of the page type[4] is applied only by GIN indexes, and only in the following case.

[1] backend/storage/lmgr/lmgr.c, LockRelationForExtension function
[2] backend/access/heap/hio.c, RelationAddExtraBlocks function
[3] backend/access/nbtree/nbtpage.c, _bt_getbuf function
[4] backend/storage/lmgr/lmgr.c, LockPage function

GIN indexes can speed up search of elements in compound values, such as words in text documents. They can be roughly described as B-trees that store separate words rather than the whole documents. When a new document is added, the index has to be thoroughly updated to include each word that appears in this document.

To improve performance, GIN indexes allow deferred insertion, which is controlled
on by the *fastupdate* storage parameter. New words are first quickly added into an unordered *pending list*, and after a while all the accumulated entries are moved into the main index structure. Since different documents are likely to contain duplicate words, this approach proves to be quite cost-effective.

To avoid concurrent transfer of words by several processes, the index metapage is locked in the exclusive mode until all the words are moved from the pending list to the main index. This lock does not interfere with regular index usage.

Just like relation extension locks, page locks are released immediately when the task is complete, without waiting for the end of the transaction, so they never cause deadlocks.

14.4 Advisory Locks

Unlike other heavyweight locks (such as relation locks), *advisory locks*[1] are never acquired automatically: they are controlled by the application developer. These locks are convenient to use if the application requires dedicated locking logic for some particular purpose.

Suppose we need to lock a resource that does not correspond to any database object (which we could lock using SELECT FOR or LOCK TABLE commands). In this case, the resource needs to be assigned a numeric ID. If the resource has a unique name, the easiest way to do it is to generate a hash code for this name:

```
=> SELECT hashtext('resource1');
 hashtext
-----------
 991601810
(1 row)
```

[1] postgresql.org/docs/14/explicit-locking#ADVISORY-LOCKS.html

PostgreSQL provides a whole class of functions for managing advisory locks.[1] Their names begin with the pg_advisory prefix and can contain the following words that hint at the function purpose:

lock — acquire a lock

try — acquire a lock if it can be done without waits

unlock — release the lock

share — use a shared locking mode (by default, the exclusive mode is used)

xact — acquire and hold a lock till the end of the transaction (by default, the lock is held till the end of the session)

Let's acquire an exclusive lock until the end of the session:

```
=> BEGIN;
=> SELECT pg_advisory_lock(hashtext('resource1'));
=> SELECT locktype, objid, mode, granted
FROM pg_locks WHERE locktype = 'advisory' AND pid = pg_backend_pid();
 locktype |   objid   |     mode      | granted
----------+-----------+---------------+---------
 advisory | 991601810 | ExclusiveLock | t
(1 row)
```

For advisory locks to actually work, other processes must also observe the established order when accessing the resource; it must be guaranteed by the application.

The acquired lock will be held even after the transaction is complete:

```
=> COMMIT;
=> SELECT locktype, objid, mode, granted
FROM pg_locks WHERE locktype = 'advisory' AND pid = pg_backend_pid();
 locktype |   objid   |     mode      | granted
----------+-----------+---------------+---------
 advisory | 991601810 | ExclusiveLock | t
(1 row)
```

Once the operation on the resource is over, the lock has to be explicitly released:

```
=> SELECT pg_advisory_unlock(hashtext('resource1'));
```

[1] postgresql.org/docs/14/functions-admin#FUNCTIONS-ADVISORY-LOCKS.html

14.5 Predicate Locks

The term *predicate lock* appeared as early as the first attempts to implement full isolation based on locks.[1] The problem confronted at that time was that locking all the rows to be read and updated still could not guarantee full isolation. Indeed, if *new* rows that satisfy the filter condition get inserted into the table, they will
p. 47 become *phantoms*.

For this reason, it was suggested to lock conditions (predicates) rather than rows. If you run a query with the $a > 10$ predicate, locking this predicate will not allow adding new rows into the table if they satisfy this condition, so phantoms will be avoided. The trouble is that if a query with a different predicate appears, such as $a < 20$, you have to find out whether these predicates overlap. In theory, this problem is algorithmically unsolvable; in practice, it can be solved only for a very simple class of predicates (like in this example).

In PostgreSQL, the Serializable isolation level is implemented in a different way: it uses the Serializable Snapshot Isolation (SSI) protocol.[2] The term *predicate lock* still remains, but its sense has radically changed. In fact, such "locks" do not lock anything: they are used to track data dependencies between different transactions.

p. 62 It is proved that snapshot isolation at the Repeatable Read level allows no anomalies except for the *write skew* and the *read-only transaction anomaly*. These two anomalies result in certain patterns in the data dependence graph that can be discovered at a relatively low cost.

The problem is that we must differentiate between two types of dependencies:

- The first transaction reads a row that is later updated by the second transaction (RW dependency).
- The first transaction modifies a row that is later read by the second transaction (WR dependency).

[1] *K. P. Eswaran, J. N. Gray, R. A. Lorie, I. L. Traiger*. The notions of consistency and predicate locks in a database system

[2] backend/storage/lmgr/README-SSI
backend/storage/lmgr/predicate.c

WR dependencies can be detected using regular locks, but RW dependencies have to be tracked via predicate locks. Such tracking is turned on automatically at the Serializable isolation level, and that's exactly why it is important to use this level for *all* transactions (or at least all the interconnected ones). If any transaction is running at a different level, it will not set (or check) predicate locks, so the Serializable level will be downgraded to Repeatable Read.

I would like to stress once again that despite their name, predicate locks do not lock anything. Instead, a transaction is checked for "dangerous" dependencies when it is about to be committed, and if PostgreSQL suspects an anomaly, this transaction will be aborted.

Let's create a table with an index that will span several pages (it can be achieved by using a low *fillfactor* value):

```
=> CREATE TABLE pred(n numeric, s text);

=> INSERT INTO pred(n) SELECT n FROM generate_series(1,10000) n;

=> CREATE INDEX ON pred(n) WITH (fillfactor = 10);

=> ANALYZE pred;
```

If the query performs a sequential scan, a predicate lock is acquired on the whole table (even if some of the rows do not satisfy the provided filter conditions).

```
=> SELECT pg_backend_pid();
 pg_backend_pid
----------------
          34903
(1 row)
=> BEGIN ISOLATION LEVEL SERIALIZABLE;
=> EXPLAIN (analyze, costs off, timing off, summary off)
  SELECT * FROM pred WHERE n > 100;
                 QUERY PLAN
---------------------------------------------
 Seq Scan on pred (actual rows=9900 loops=1)
   Filter: (n > '100'::numeric)
   Rows Removed by Filter: 100
(3 rows)
```

Although predicate locks have their own infrastructure, the pg_locks view displays them together with heavyweight locks. All predicate locks are always acquired in the SIRead mode, which stands for Serializable Isolation Read:

```
=> SELECT relation::regclass, locktype, page, tuple
FROM pg_locks WHERE mode = 'SIReadLock' AND pid = 34903
ORDER BY 1, 2, 3, 4;
 relation | locktype | page | tuple
----------+----------+------+-------
 pred     | relation |      |
(1 row)
```

```
=> ROLLBACK;
```

Note that predicate locks may be held longer than the transaction duration, as they are used to track dependencies *between* transactions. But anyway, they are managed automatically.

If the query performs an index scan, the situation improves. For a B-tree index, it is enough to set a predicate lock on the read heap tuples and on the scanned leaf pages of the index. It will "lock" the whole range that has been read, not only the exact values.

```
=> BEGIN ISOLATION LEVEL SERIALIZABLE;
=> EXPLAIN (analyze, costs off, timing off, summary off)
  SELECT * FROM pred WHERE n BETWEEN 1000 AND 1001;
                            QUERY PLAN
-----------------------------------------------------------------
 Index Scan using pred_n_idx on pred (actual rows=2 loops=1)
   Index Cond: ((n >= '1000'::numeric) AND (n <= '1001'::numeric))
(2 rows)
```

```
=> SELECT relation::regclass, locktype, page, tuple
FROM pg_locks WHERE mode = 'SIReadLock' AND pid = 34903
ORDER BY 1, 2, 3, 4;
  relation  | locktype | page | tuple
------------+----------+------+-------
 pred       | tuple    |    4 |    96
 pred       | tuple    |    4 |    97
 pred_n_idx | page     |   28 |
(3 rows)
```

The number of leaf pages corresponding to the already scanned tuples can change: for example, an index page can be split when new rows get inserted into the table. However, PostgreSQL takes it into account and locks newly appeared pages too:

```
=> INSERT INTO pred
  SELECT 1000+(n/1000.0) FROM generate_series(1,999) n;
=> SELECT relation::regclass, locktype, page, tuple
FROM pg_locks WHERE mode = 'SIReadLock' AND pid = 34903
ORDER BY 1, 2, 3, 4;
  relation  | locktype | page | tuple
------------+----------+------+-------
 pred       | tuple    |    4 |    96
 pred       | tuple    |    4 |    97
 pred_n_idx | page     |   28 |
 pred_n_idx | page     |  266 |
 pred_n_idx | page     |  267 |
 pred_n_idx | page     |  268 |
 pred_n_idx | page     |  269 |
(7 rows)
```

Each read tuple is locked separately, and there may be quite a few of such tuples. Predicate locks use their own pool allocated at the server start. The total number of predicate locks is limited by the *max_pred_locks_per_transaction* value multiplied 64
by *max_connections* (despite the parameter names, predicate locks are not being 100
counted per separate transactions).

Here we get the same problem as with row-level locks, but it is solved in a different way: *lock escalation* is applied.[1]

As soon as the number of tuple locks related to one page exceeds the value of the v. 10
max_pred_locks_per_page parameter, they are replaced by a single page-level lock. 2

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
  SELECT * FROM pred WHERE n BETWEEN 1000 AND 1002;
                           QUERY PLAN
-----------------------------------------------------------------
 Index Scan using pred_n_idx on pred (actual rows=3 loops=1)
   Index Cond: ((n >= '1000'::numeric) AND (n <= '1002'::numeric))
(2 rows)
```

[1] backend/storage/lmgr/predicate.c, PredicateLockAcquire function

Instead of three locks of the tuple type we now have one lock of the page type:

```
=> SELECT relation::regclass, locktype, page, tuple
FROM pg_locks WHERE mode = 'SIReadLock' AND pid = 34903
ORDER BY 1, 2, 3, 4;
  relation  | locktype | page | tuple
------------+----------+------+-------
 pred       | page     |    4 |
 pred_n_idx | page     |   28 |
 pred_n_idx | page     |  266 |
 pred_n_idx | page     |  267 |
 pred_n_idx | page     |  268 |
 pred_n_idx | page     |  269 |
(6 rows)

=> ROLLBACK;
```

v. 10 -2 64 Escalation of page-level locks follows the same principle. If the number of such locks for a particular relation exceeds the *max_pred_locks_per_relation* value, they get replaced by a single relation-level lock. (If this parameter is set to a negative value, the threshold is calculated as *max_pred_locks_per_transaction* divided by the absolute value of *max_pred_locks_per_relation*; thus, the default threshold is 32).

Lock escalation is sure to lead to multiple false-positive serialization errors, which negatively affects system throughput. So you have to find an appropriate balance between performance and spending the available RAM on locks.

Predicate locks support the following index types:

- B-trees
- v. 11 hash indexes, GiST, and GIN

If an index scan is performed, but the index does not support predicate locks, the whole index will be locked. It is only to be expected that the number of transactions aborted for no good reason will also increase in this case.

For more efficient operation at the Serializable level, it makes sense to explicitly declare read-only transactions as such using the READ ONLY clause. If the lock manager sees that a read-only transaction will not conflict with other transactions,[1] it

[1] backend/storage/lmgr/predicate.c, SxactIsROSafe macro

can release the already set predicate locks and refrain from acquiring new ones. And if such a transaction is also declared DEFERRABLE, the read-only transaction anomaly will be avoided too. *p. 66*

15

Locks on Memory Structures

15.1 Spinlocks

To protect data structures in shared memory, PostgreSQL uses several types of lighter and less expensive locks rather than regular heavyweight ones.

The simplest locks are *spinlocks*. They are usually acquired for a very short time interval (no longer than several CPU cycles) to protect particular memory cells from concurrent updates.

Spinlocks are based on atomic CPU instructions, such as compare-and-swap.[1] They only support the exclusive locking mode. If the required resource is already locked, the process busy-waits, repeating the command (it "spins" in the loop, hence the name). If the lock cannot be acquired within the specified time interval, the process pauses for a while and then starts another loop.

This strategy makes sense if the probability of a conflict is estimated as very low, so after an unsuccessful attempt the lock is likely to be acquired within several instructions.

Spinlocks have neither deadlock detection nor instrumentation. From the practical standpoint, we should simply know about their existence; the whole responsibility for their correct implementation lies with PostgreSQL developers.

[1] backend/storage/lmgr/s_lock.c

15.2 Lightweight Locks

Next, there are so-called 🔒 *lightweight locks*, or lwlocks.[1] Acquired for the time needed to process a data structure (for example, a hash table or a list of pointers), lightweight locks are typically short; however, they can take longer when used to protect I/O operations.

Lightweight locks support two modes: exclusive (for data modification) and shared (for read-only operations). There is no queue as such: if several processes are waiting on a lock, one of them will get access to the resource in a more or less random fashion. In high-load systems with multiple concurrent processes, it can lead to some unpleasant effects.

Deadlock checks are not provided; we have to trust PostgreSQL developers that lightweight locks are implemented correctly. However, these locks do have instrumentation, so, unlike spinlocks, they can be observed.

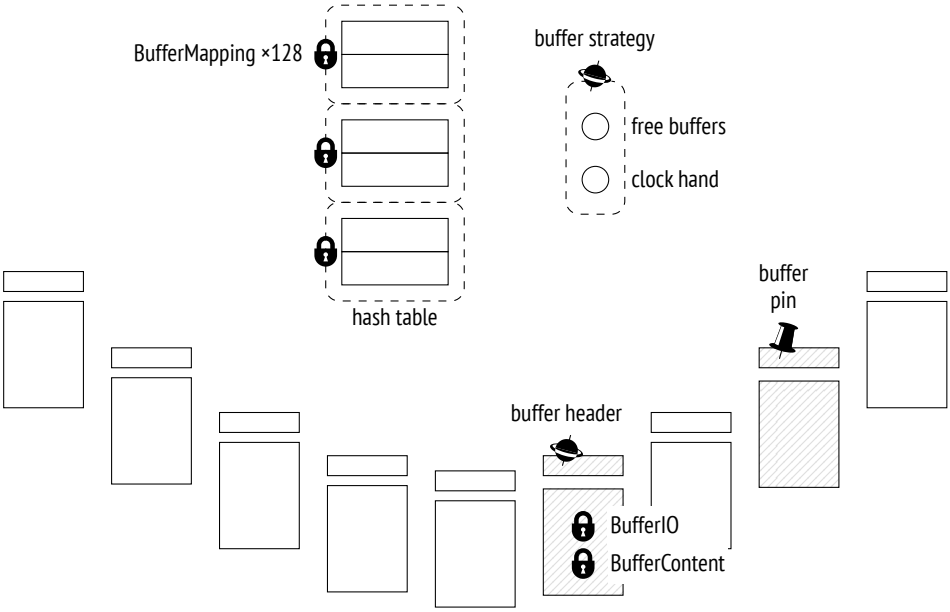
15.3 Examples

To get some idea of how and where spinlocks and lightweight locks can be used, let's take a look at two shared memory structures: buffer cache and WAL buffers. I will name only some of the locks; the full picture is too complex and is likely to interest only PostgreSQL core developers.

Buffer Cache

To access a hash table used to locate a particular buffer in the cache, the process must acquire a 🔒 BufferMapping lightweight lock either in the shared mode for reading or in the exclusive mode if any modifications are expected. *p. 169*

[1] backend/storage/lmgr/lwlock.c



The hash table is accessed very frequently, so this lock often becomes a bottleneck. To maximize granularity, it is structured as a *tranche* of 128 individual lightweight locks, each protecting a separate part of the hash table.[1]

> A hash table lock was converted into a tranche of 16 locks as early as 2006, in PostgreSQL 8.2; ten years later, when version 9.5 was released, the size of the tranche was increased to 128, but it may still be not enough for modern multi-core systems.

To get access to the buffer header, the process acquires a buffer header spinlock[2] (the name is arbitrary, as spinlocks have no user-visible names). Some operations, such as incrementing the usage counter, do not require explicit locks and can be performed using atomic CPU instructions.

To read a page in a buffer, the process acquires a BufferContent lock in the header of this buffer.[3] It is usually held only while tuple pointers are being read; later on, the protection provided by buffer pinning will be enough. If the buffer content has to be modified, the BufferContent lock must be acquired in the exclusive mode.

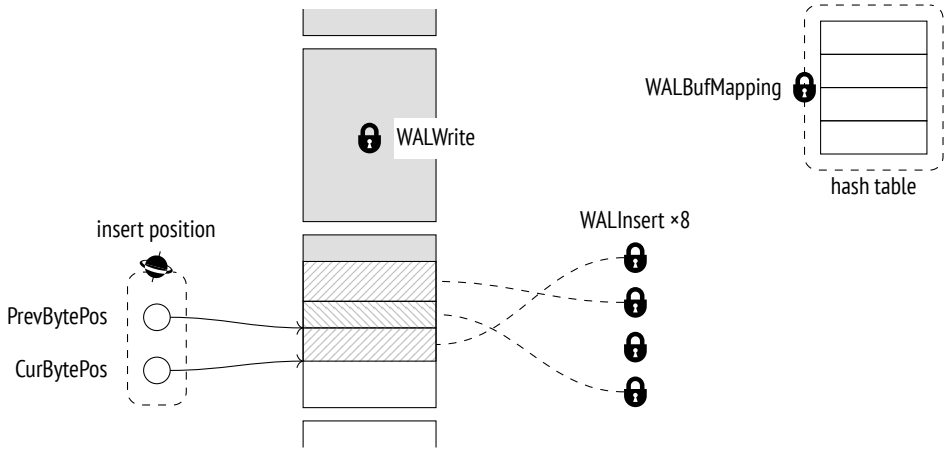
p. 171

[1] backend/storage/buffer/bufmgr.c
include/storage/buf_internals.h, BufMappingPartitionLock function
[2] backend/storage/buffer/bufmgr.c, LockBufHdr function
[3] include/storage/buf_internals.h

When a buffer is read from disk (or written to disk), PostgreSQL also acquires a BufferIO lock in the buffer header; it is virtually an attribute used as a lock rather than an actual lock.[1] It signals other processes requesting access to this page that they have to wait until the I/O operation is complete.

The pointer to free buffers and the clock hand of the eviction mechanism are protected by a single common buffer strategy spinlock.[2]

WAL Buffers



WAL cache also uses a hash table to map pages to buffers. Unlike the buffer cache hash table, it is protected by a single WALBufMapping lightweight lock because WAL cache is smaller (it usually takes $\frac{1}{32}$ of the buffer cache size) and buffer access is more ordered.[3]

Writing of WAL pages to disk is protected by a WALWrite lightweight lock, which ensures that this operation is performed by one process at a time.

To create a WAL entry, the process first reserves some space within the WAL page and then fills it with data. Space reservation is strictly ordered; the process must acquire an insert position spinlock that protects the insertion pointer.[4] But

[1] backend/storage/buffer/bufmgr.c, StartBufferIO function
[2] backend/storage/buffer/freelist.c
[3] backend/access/transam/xlog.c, AdvanceXLInsertBuffer function
[4] backend/access/transam/xlog.c, ReserveXLogInsertLocation function

once the space is reserved, it can be filled by several concurrent processes. For this purpose, each process must acquire *any* of the eight lightweight locks constituting the WALInsert tranche.[1]

15.4 Monitoring Waits

Without doubt, locks are indispensable for correct PostgreSQL operation, but they can lead to undesirable waits. It is useful to track such waits to understand their origin.

off The easiest way to get an overview of long-term locks is to turn the *log_lock_waits* parameter on; it enables extensive logging of all the locks that cause a transaction
1s to wait for more than *deadlock_timeout*. This data is displayed when a deadlock
p. 256 check completes, hence the parameter name.

v. 9.6 However, the pg_stat_activity view provides much more useful and complete information. Whenever a process—either a system process or a backend—cannot proceed with its task because it is waiting for something, this wait is reflected in the wait_event_type and wait_event fields, which show the type and name of the wait, respectively.

All waits can be classified as follows.[2]

Waits on various locks constitute quite a large group:

Lock — heavyweight locks

LWLock — lightweight locks

BufferPin — pinned buffers

But processes can be waiting for other events too:

IO — input/output, when it is required to read or write some data

[1] backend/access/transam/xlog.c, WALInsertLockAcquire function
[2] postgresql.org/docs/14/monitoring-stats#WAIT-EVENT-TABLE.html

Client — data sent by the client (psql spends in this state most of the time)

IPC — data sent by another process

Extension — a specific event registered by an extension

Sometimes a process simply does not perform any useful work. Such waits are usually "normal," meaning that they do not indicate any issues. This group comprises the following waits:

Activity — background processes in their main cycle

Timeout — timer

Locks of each wait type are further classified by wait names. For example, waits on lightweight locks get the name of the lock or the corresponding tranche.[1]

You should bear in mind that the pg_stat_activity view displays only those waits that are handled in the source code in an appropriate way.[2] Unless the name of the wait appears in this view, the process is not in the state of wait of any known type. Such time should be considered *unaccounted for*; it does not necessarily mean that the process is not waiting on anything—we simply do not know what is happening at the moment.

```
=> SELECT backend_type, wait_event_type AS event_type, wait_event
FROM pg_stat_activity;
         backend_type          | event_type |     wait_event
-------------------------------+------------+---------------------
 logical replication launcher | Activity   | LogicalLauncherMain
 autovacuum launcher          | Activity   | AutoVacuumMain
 client backend               |            |
 background writer            | Activity   | BgWriterMain
 checkpointer                 | Activity   | CheckpointerMain
 walwriter                    | Activity   | WalWriterMain
(6 rows)
```

Here all the background processes were idle when the view was sampled, while the client backend was busy executing the query and was not waiting on anything.

[1] postgresql.org/docs/14/monitoring-stats#WAIT-EVENT-LWLOCK-TABLE.html
[2] include/utils/wait_event.h

15.5 Sampling

Unfortunately, the pg_stat_activity view shows only the *current* information on waits; statistics are not accumulated. The only way to collect wait data over time is to *sample* the view at regular intervals.

We have to take into account the stochastic nature of sampling. The shorter the wait as compared to the sampling interval, the lower the chance to detect this wait. Thus, longer sampling intervals require more samples to reflect the actual state of things (but as you increase the sampling rate, the overhead also rises). For the same reason, sampling is virtually useless for analyzing short-lived sessions.

PostgreSQL provides no built-in tools for sampling; however, we can still try it out using the pg_wait_sampling[1] extension. To do so, we have to specify its library in the *shared_preload_libraries* parameter and restart the server:

```
=> ALTER SYSTEM SET shared_preload_libraries = 'pg_wait_sampling';
postgres$ pg_ctl restart -l /home/postgres/logfile
```

Now let's install the extension into the database:

```
=> CREATE EXTENSION pg_wait_sampling;
```

This extension can display the history of waits, which is saved in its ring buffer. However, it is much more interesting to get the waiting profile—the accumulated statistics for the whole duration of the session.

For example, let's take a look at the waits during benchmarking. We have to start the pgbench utility and determine its process ID while it is running:

```
postgres$ /usr/local/pgsql/bin/pgbench -T 60 internals
=> SELECT pid FROM pg_stat_activity
WHERE application_name = 'pgbench';
  pid
-------
 36520
(1 row)
```

Once the test is complete, the waits profile will look as follows:

[1] github.com/postgrespro/pg_wait_sampling

```
=> SELECT pid, event_type, event, count
FROM pg_wait_sampling_profile WHERE pid = 36520
ORDER BY count DESC LIMIT 4;
  pid  | event_type |     event      | count
-------+------------+----------------+-------
 36520 | IO         | WALSync        |  4380
 36520 | IO         | WALWrite       |   306
 36520 | Client     | ClientRead     |    38
 36520 | IO         | DataFileExtend |     7
(4 rows)
```

By default (set by the *pg_wait_sampling.profile_period* parameter) samples are taken 100 times per second. So to estimate the duration of waits in seconds, you have to divide the count value by 100. 10ms

In this particular case, most of the waits are related to flushing WAL entries to disk. It is a good illustration of the unaccounted-for wait time: the WALSync event was not instrumented until PostgreSQL 12; for lower versions, a waits profile would not contain the first row, although the wait itself would still be there. v. 12

And here is how the profile will look like if we artificially slow down the file system for each I/O operation to take 0.1 seconds (I use slowfs[1] for this purpose) :

```
postgres$ /usr/local/pgsql/bin/pgbench -T 60 internals
=> SELECT pid FROM pg_stat_activity
WHERE application_name = 'pgbench';
  pid
-------
 36953
(1 row)

=> SELECT pid, event_type, event, count
FROM pg_wait_sampling_profile WHERE pid = 36953
ORDER BY count DESC LIMIT 4;
  pid  | event_type |     event      | count
-------+------------+----------------+-------
 36953 | IO         | WALWrite       |  4379
 36953 | LWLock     | WALWrite       |  1527
 36953 | IO         | WALSync        |    22
 36953 | IO         | DataFileExtend |    20
(4 rows)
```

[1] github.com/nirs/slowfs

Now I/O operations are the slowest ones—mainly those that are related to writing WAL files to disk in the synchronous mode. Since WAL writing is protected by a WALWrite lightweight lock, the corresponding row also appears in the profile.

Clearly, the same lock is acquired in the previous example too, but since the wait is shorter than the sampling interval, it either is sampled very few times or does not make it into the profile at all. It illustrates once again that to analyze short waits you have to sample them for quite a long time.

Part IV

Query Execution

16

Query Execution Stages

16.1 Demo Database

The examples in the previous parts of the book were based on simple tables with only a handful of rows. This and subsequent parts deal with query execution, which is more demanding in this respect: we need related tables that have a much larger number of rows. Instead of inventing a new data set for each example, I took an existing demo database that illustrates passenger air traffic in Russia.[1] It has several versions; we will use the bigger one created on August 15, 2017. To install this version, you have to extract the file containing the database copy from the archive[2] and run this file in psql.

When developing this demo database, we tried to make its schema simple enough to be understood without extra explanations; at the same time, we wanted it to be complex enough to allow writing meaningful queries. The database is filled with true-to-life data, which makes the examples more comprehensive and should be interesting to work with.

Here I will cover the main database objects only briefly; if you would like to review the whole schema, you can take a look at its full description referenced in the footnote.

The main entity is a **booking** (mapped to the bookings table). One booking can include several passengers, each with a separate electronic **ticket** (tickets). A passenger does not constitute a separate entity; for the purpose of our experiments, we will assume that all passengers are unique.

[1] postgrespro.com/community/demodb
[2] edu.postgrespro.com/demo-big-en-20170815.zip

Each ticket includes one or more **flight segments** (mapped to the ticket_flights table). A single ticket can have several flight segments in two cases: either it is a round-trip ticket, or it is issued for connecting flights. Although there is no corresponding constraint in the schema, all tickets in a booking are assumed to have the same flight segments.

Each **flight** (flights) goes from one **airport** (airports) to another. Flights with the same flight number have the same points of departure and destination but different departure dates.

The routes view is based on the flights table; it displays the information on **routes** that does not depend on particular flight dates.

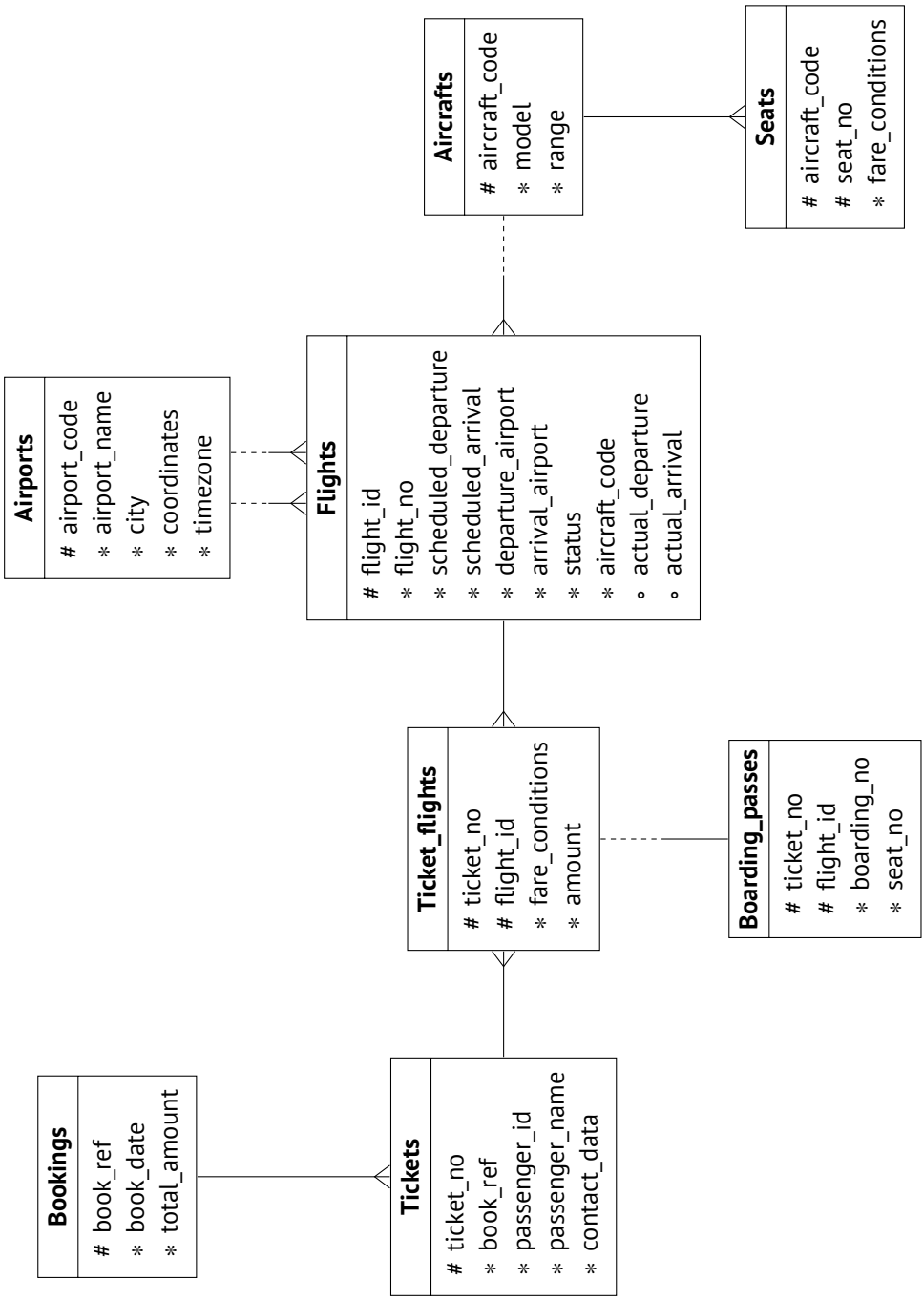
At check-in, each passenger is issued a **boarding pass** (boarding_passes) with a seat number. A passenger can check in for a flight only if this flight is included into the ticket. Flight-seat combinations must be unique, so it is impossible to issue two boarding passes for the same seat.

The number of **seats** (seats) in an aircraft and their distribution between different travel classes depend on the particular model of the **aircraft** (aircrafts) that performs the flight. It is assumed that each aircraft model can have only one cabin configuration.

Some tables have surrogate primary keys, while others use natural ones (some of them being composite). It is done solely for demonstration purposes and is by no means an example to follow.

The demo database can be thought of as a dump of a real system: it contains a snapshot of data taken at a particular time in the past. To display this time, you can call the bookings.now() function. Use this function in demo queries that would demand the now() function in real life.

The names of airports, cities, and aircraft models are stored in the airports_data and aircrafts_data tables; they are provided in two languages, English and Russian. To construct examples for this chapter, I will typically query the airports and aircrafts views shown in the entity-relationship diagram; these views choose the output
en language based on the *bookings.lang* parameter value. The names of some base tables can still appear in query plans though.


Airports
airport_code
* airport_name
* city
* coordinates
* timezone
Aircrafts
aircraft_code
* model
* range
Seats
aircraft_code
seat_no
* fare_conditions
Flights
flight_id
* flight_no
* scheduled_departure
* scheduled_arrival
* departure_airport
* arrival_airport
* status
* aircraft_code
∘ actual_departure
∘ actual_arrival
Ticket_flights
ticket_no
flight_id
* fare_conditions
* amount
Boarding_passes
ticket_no
flight_id
* boarding_no
* seat_no
Bookings
book_ref
* book_date
* total_amount
Tickets
ticket_no
* book_ref
* passenger_id
* passenger_name
* contact_data

16.2 Simple Query Protocol

A simple version of the client-server protocol[1] enables SQL query execution: it sends the text of a query to the server and gets the full execution result in response, no matter how many rows it contains.[2] A query sent to the server passes several stages: it is parsed, transformed, planned, and then executed.

Parsing

First of all, PostgreSQL has to *parse*[3] the query text to understand what needs to be executed.

Lexical and syntactic analyisis. The *lexer* splits the query text into a set of *lexemes*[4] (such as keywords, string literals, and numeric literals), while the *parser* validates this set against the SQL language grammar.[5] PostgreSQL relies on standard parsing tools, namely Flex and Bison utilities.

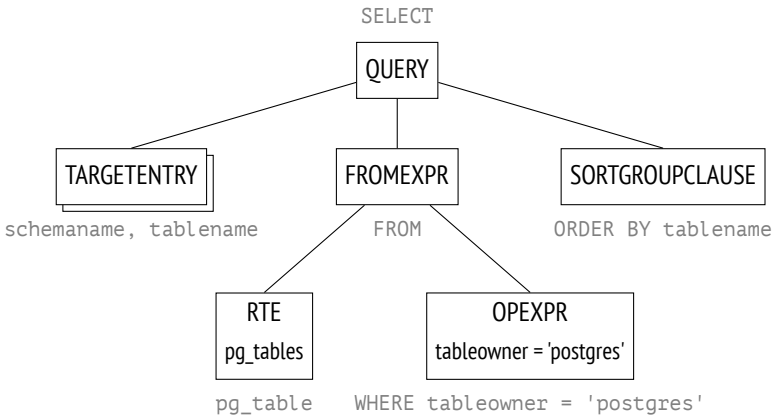
The parsed query is reflected in the backend's memory as an abstract syntax tree.

For example, let's take a look at the following query:

```
SELECT schemaname, tablename
FROM pg_tables
WHERE tableowner = 'postgres'
ORDER BY tablename;
```

The lexer singles out five keywords, five identifiers, a string literal, and three single-letter lexemes (a comma, an equals sign, and a semicolon). The parser uses these lexemes to build the parse tree, which is shown in the illustration below in a very simplified form. The captions next to the tree nodes specify the corresponding parts of the query:

[1] postgresql.org/docs/14/protocol.html
[2] backend/tcop/postgres.c, exec_simple_query function
[3] postgresql.org/docs/14/parser-stage.html
backend/parser/README
[4] backend/parser/scan.l
[5] backend/parser/gram.y



A rather obscure RTE abbreviation stands for *Range Table Entry*. PostgreSQL source code uses the term *range table* to refer to tables, subqueries, join results—in other words, to any *sets of rows* that can be processed by SQL operators.[1]

Semantic analysis. The purpose of *semantic analysis*[2] is to determine whether the database contains any tables or other objects that this query refers to by name, and whether the user has permission to access these objects. All the information required for semantic analysis is stored in the system catalog. *p. 22*

Having received the parse tree, the semantic analyzer performs its further restructuring, which includes adding references to specific database objects, data types, and other information.

If you enable the *debug_print_parse* parameter, you can view the full parse tree in the server log, but it has little practical sense.

Transformation

At the next stage, the query can be *transformed* (*rewritten*).[3]

[1] include/nodes/parsenodes.h
[2] backend/parser/analyze.c
[3] postgresql.org/docs/14/rule-system.html

PostgreSQL core uses transformations for several purposes. One of them is to replace the name of the view in the parse tree with the subtree corresponding to the base query of this view.

Another case of using transformations is row-level security implementation.[1]

v. 14 The SEARCH and CYCLE clauses of recursive queries also get transformed during this stage.[2]

In the example above, pg_tables is a view; if we placed its definition into the query text, it would look as follows:

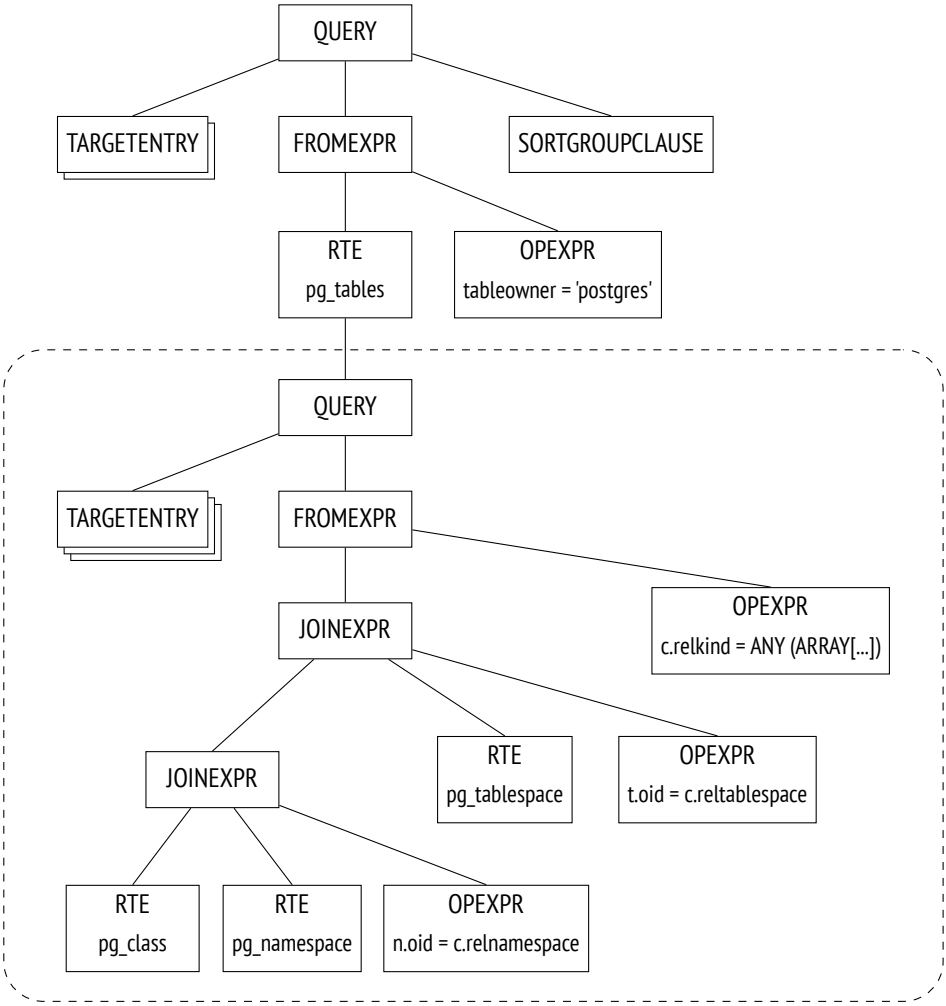
```
SELECT schemaname, tablename
FROM (
    -- pg_tables
    SELECT n.nspname AS schemaname,
      c.relname AS tablename,
      pg_get_userbyid(c.relowner) AS tableowner,
      ...
    FROM pg_class c
      LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
      LEFT JOIN pg_tablespace t ON t.oid = c.reltablespace
    WHERE c.relkind = ANY (ARRAY['r'::char, 'p'::char])
)
WHERE tableowner = 'postgres'
ORDER BY tablename;
```

However, the server does not process the text representation of the query; all manipulations are performed on the parse tree. The illustration shows a reduced version of the transformed tree (you can view its full version in the server log if you enable the *debug_print_rewritten* parameter).

The parse tree reflects the syntactic structure of the query, but it says nothing about the order in which the operations should be performed.

PostgreSQL also supports custom transformations, which the user can implement via the *rewrite rule system*.[3]

[1] backend/rewrite/rowsecurity.c
[2] backend/rewrite/rewriteSearchCycle.c
[3] postgresql.org/docs/14/rules.html



The rule system support was proclaimed as one of the main objectives of Postgres development;[1] it was still an academic project when the rules were first implemented, but since then they have been redesigned multiple times. The rule system is a very powerful mechanism, but it is rather hard to comprehend and debug. It was even proposed to remove the rules from PostgreSQL altogether, but the idea did not find unanimous support. In most cases, it is safer and easier to use triggers instead of rules.

[1] *M. Stonebraker, L. A. Rowe*. The Design of Postgres

Planning

SQL is a declarative language: queries specify *what* data to fetch, but not *how* to fetch it.

Any query has several execution paths. Each operation shown in the parse tree can be completed in a number of ways: for example, the result can be retrieved by reading the whole table (and filtering out redundancies), or by finding the required rows via an index scan. Data sets are always joined in pairs, so there is a huge number of options that differ in the order of joins. Besides, there are various join algorithms: for example, the executor can scan the rows of the first data set and search for the matching rows in the other set, or both data sets can be first sorted and then merged together. For each algorithm, we can find a use case where it performs better than others.

The execution times of optimal and non-optimal plans can differ by orders of magnitude, so the *planner*[1] that *optimizes* the parsed query is one of the most complex components of the system.

Plan tree. The execution plan is also represented as a tree, but its nodes deal with physical operations on data rather than logical ones.

If you would like to explore full plan trees, you can dump them into the server log by enabling the *debug_print_plan* parameter. But in practice it is usually enough to view the text representation of the plan displayed by the EXPLAIN command.[2]

The following illustration highlights the main nodes of the tree. It is exactly these nodes that are shown in the output of the EXPLAIN command provided below.

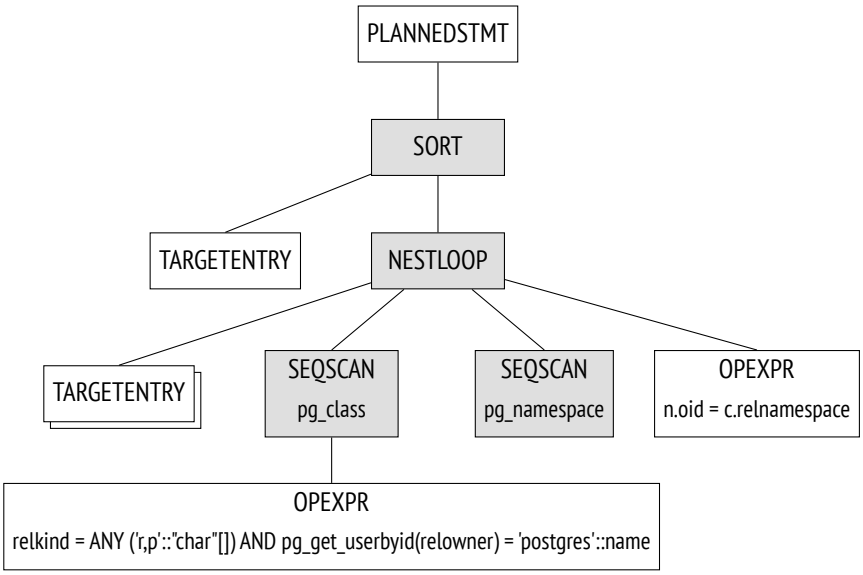
For now, let's pay attention to the following two points:

- The tree contains only two queried tables out of three: the planner saw that one of the tables is not required for retrieving the result and removed it from the plan tree.
- For each node of the tree, the planner provides the estimated cost and the number of rows expected to be processed.

[1] postgresql.org/docs/14/planner-optimizer.html
[2] postgresql.org/docs/14/using-explain.html

```
=> EXPLAIN SELECT schemaname, tablename
FROM pg_tables
WHERE tableowner = 'postgres'
ORDER BY tablename;
                              QUERY PLAN
----------------------------------------------------------------------
 Sort  (cost=21.03..21.04 rows=1 width=128)
   Sort Key: c.relname
   -> Nested Loop Left Join  (cost=0.00..21.02 rows=1 width=128)
       Join Filter: (n.oid = c.relnamespace)
       -> Seq Scan on pg_class c  (cost=0.00..19.93 rows=1 width=72)
           Filter: ((relkind = ANY ('{r,p}'::"char"[])) AND (pg_g...
       -> Seq Scan on pg_namespace n  (cost=0.00..1.04 rows=4 wid...
(7 rows)
```

Seq Scan nodes shown in the query plan correspond to reading the table, while the Nested Loop node represents the join operation. *p. 335* *p. 398*



Plan search. PostgreSQL uses a *cost-based optimizer*;[1] it goes over potential plans and estimates the resources required for their execution (such as I/O operations or CPU cycles). Normalized to a numeric value, this estimation is called the *cost* of the plan. Of all the considered plans, the one with the lowest cost is selected.

[1] backend/optimizer/README

The problem is that the number of potentially available plans grows exponentially with the number of joined tables, so it is impossible to consider them all—even for relatively simple queries. The search is typically narrowed down using the dynamic programming algorithm combined with some heuristics. It allows the planner to find a mathematically accurate solution for queries with a larger number of tables within acceptable time.

An accurate solution does not guarantee that the selected plan is *really* the optimal one, as the planner uses simplified mathematical models and may lack reliable input data.

Managing the order of joins. A query can be structured in a way that limits the search scope to some extent (at a risk of missing the optimal plan).

v. 12

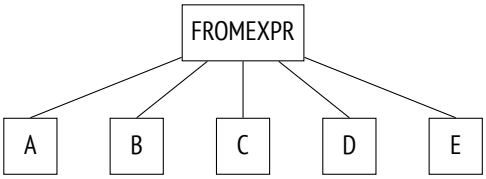
- Common table expressions and the main query can be optimized separately; to guarantee such behavior, you can specify the MATERIALIZED clause.[1]
- Subqueries run within non-SQL functions are always optimized separately. (SQL functions can sometimes be inlined into the main query.[2])
- If you set the *join_collapse_limit* parameter and use explicit JOIN clauses in the query, the order of some joins will be defined by the query syntax structure; the *from_collapse_limit* parameter has the same effect on subqueries.[3]

The latter point may have to be explained. Let's take a look at the query that does not specify any explicit joins for tables listed in the FROM clause:

```
SELECT ...
FROM a, b, c, d, e
WHERE ...
```

Here the planner will have to consider all the possible pairs of joins. The query is represented by the following part of the parse tree (shown schematically):

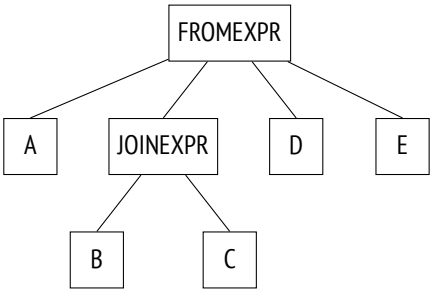
[1] postgresql.org/docs/14/queries-with.html
[2] wiki.postgresql.org/wiki/Inlining_of_SQL_functions
[3] postgresql.org/docs/14/explicit-joins.html



In the next example, joins have a certain structure defined by the JOIN clause:

```
SELECT ...
FROM a, b JOIN c ON ..., d, e
WHERE ...
```

The parse tree reflects this structure:



The planner typically flattens the join tree, so that it looks like the one in the first example. The algorithm recursively traverses the tree and replaces each JOINEXPR node with a flat list of its elements.[1]

However, such collapsing is performed only if the resulting flat list has no more than *join_collapse_limit* elements. In this particular case, the JOINEXPR node would 8
not be collapsed if the *join_collapse_limit* value were less than five.

For the planner, it means the following:

- Table B must be joined with table C (or vice versa, C must be joined with B; the order of joins within a pair is not restricted).
- Tables A, D, E and the result of joining B and C can be joined in any order.

[1] backend/optimizer/plan/initsplan.c, deconstruct_jointree function

If the *join_collapse_limit* parameter is set to one, the order defined by explicit JOIN clauses is preserved.

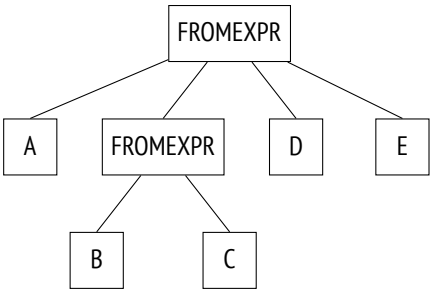
As for FULL OUTER JOIN operands, they are *never* collapsed, regardless of the value of the *join_collapse_limit* parameter.

8 The *from_collapse_limit* parameter controls subquery flattening in a similar way. Although subqueries do not look like JOIN clauses, the similarity becomes apparent at the parse tree level.

Here is a sample query:

```
SELECT ...
FROM a,
  (
    SELECT ... FROM b, c WHERE ...
  ) bc,
  d, e
WHERE ...
```

The corresponding join tree is shown below. The only difference here is that this tree contains the FROMEXPR node instead of JOINEXPR (hence the parameter name).



Genetic query optimization. A flattened tree may contain too many elements at one level—either tables or join results, which have to be optimized separately. Planning time depends exponentially on the number of data sets that have to be joined, so it can grow beyond all reasonable limits.

on 12 If the *geqo* parameter is enabled and the number of elements at one level exceeds the *geqo_threshold* value, the planner will use the *genetic algorithm* to optimize the

query.[1] This algorithm is much faster than its dynamic programming counterpart, but it cannot guarantee that the found plan will be optimal. So the rule of thumb is to avoid using the genetic algorithm by reducing the number of elements that have to be optimized.

The genetic algorithm has several configurable parameters,[2] but I am not going to cover them here.

Choosing the best plan. Whether the plan can be considered optimal or not depends on how a particular client is going to use the query result. If the client needs the full result at once (for example, to create a report), the plan should optimize retrieval of all the rows. But if the priority is to return the first rows as soon as possible (for example, to display them on screen), the optimal plan might be completely different.

To make this choice, PostgreSQL calculates two components of the cost:

```
=> EXPLAIN
SELECT schemaname, tablename
FROM pg_tables
WHERE tableowner = 'postgres'
ORDER BY tablename;
                              QUERY PLAN
----------------------------------------------------------------------
 Sort  (cost=21.03..21.04 rows=1 width=128)
   Sort Key: c.relname
   -> Nested Loop Left Join  (cost=0.00..21.02 rows=1 width=128)
       Join Filter: (n.oid = c.relnamespace)
       -> Seq Scan on pg_class c  (cost=0.00..19.93 rows=1 width=72)
           Filter: ((relkind = ANY ('{r,p}'::"char"[])) AND (pg_g...
       -> Seq Scan on pg_namespace n  (cost=0.00..1.04 rows=4 wid...
(7 rows)
```

The first component (the startup cost) represents the price you pay to prepare for node execution, while the second component (the total cost) comprises all the expenses incurred by fetching the result.

[1] postgresql.org/docs/14/geqo.html
backend/optimizer/geqo/geqo_main.c

[2] postgresql.org/docs/14/runtime-config-query#RUNTIME-CONFIG-QUERY-GEQO.html

It is sometimes stated that the startup cost is the cost of retrieving the first row of the result set, but it is not quite accurate.

To single out the preferred plans, the optimizer checks whether the query uses a cursor (either via the DECLARE command provided in SQL or declared *explicitly* in PL/pgSQL).[1] If not, the client is assumed to need the whole result at once, and the optimizer chooses the plan with the least total cost.

If the query is executed with a cursor, the selected plan must optimize retrieval
0.1 of only *cursor_tuple_fraction* of all rows. To be more exact, PostgreSQL chooses the plan with the smallest value of the following expression:[2]

$$\text{startup cost} + \textit{cursor_tuple_fraction}\,(\text{total cost} - \text{startup cost})$$

An outline of cost estimation. To estimate the total cost of a plan, we have to get cost estimations for all its nodes. The cost of a node depends on its type (it is obvious that the cost of reading heap data is not the same as the sorting cost) and on the amount of data processed by this node (larger data volumes typically incur higher costs). While node types are known, the amount of data can only be projected based on the estimated *cardinality* of input sets (the number of rows the node takes as input) and the *selectivity* of the node (the fraction of rows remaining
p. 308 at the output). These calculations rely on the collected *statistics*, such as table sizes and data distribution in table columns.

Thus, the performed optimization depends on correctness of statistical data that is gathered and updated by autovacuum.

If cardinality estimation is accurate for each node, the calculated cost is likely to adequately reflect the actual cost. The main planning flaws usually result from incorrect estimation of cardinality and selectivity, which can be caused by inaccurate or outdated statistics, inability to use it, or—to a lesser extent—by imperfect planning models.

Cardinality estimation. To calculate the cardinality of a node, the planner has to recursively complete the following steps:

[1] backend/optimizer/plan/planner.c, standard_planner function
[2] backend/optimizer/util/pathnode.c, compare_fractional_path_costs function

1 Estimate the cardinality of each child node and assess the number of input rows that the node will receive from them.

2 Estimate the selectivity of the node, that is, the fraction of input rows that will remain at the output.

The cardinality of the node is the product of these two values.

Selectivity is represented by a number from 0 to 1. The smaller the number, the higher the selectivity, and vice versa, a number that is close to one denotes low selectivity. It may seem illogical, but the idea is that a *highly selective* condition rejects almost all the rows, while the one that dismisses only a few has *low selectivity*.

First, the planner estimates cardinalities of leaf nodes that define data access methods. These calculations rely on the collected statistics, such as the total size of the table.

Selectivity of filter conditions depends on their types. In the most trivial case, it can be assumed to be a constant value, although the planner tries to use all the available information to refine the estimation. In general, it is enough to know how to estimate simple filter conditions; if a condition includes logical operations, its selectivity is calculated by the following formulas:[1]

$$sel_{x \text{ and } y} = sel_x \, sel_y$$
$$sel_{x \text{ or } y} = 1 - (1 - sel_x)(1 - sel_y) = sel_x + sel_y - sel_x \, sel_y$$

Unfortunately, these formulas assume that predicates x and y do not depend on each other. For correlated predicates, such estimations will be inaccurate. p. 327

To estimate the cardinality of joins, the planner has to get the cardinality of the Cartesian product (that is, the product of cardinalities of two data sets) and estimate the selectivity of join conditions, which is again dependent on condition types.

Cardinality of other nodes (such as sorting or aggregation) is estimated in a similar manner.

It is important to note that incorrect cardinality estimation for lower plan nodes affects all the subsequent calculations, leading to inaccurate total cost estimation

[1] backend/optimizer/path/clausesel.c, clauselist_selectivity_ext & clauselist_selectivity_or functions

and a poor plan choice. To make things worse, the planner has no statistics on join results, only on tables.

Cost estimation. The process of estimating the cost is also recursive. To calculate the cost of a subtree, it is required to calculate and sum up the costs of all its child nodes and then add the cost of the parent node itself.

To estimate the cost of a node, PostgreSQL applies the mathematical model of the operation performed by this node, using the already estimated node cardinality as input. For each node, both startup and total costs are calculated.

Some operations have no prerequisites, so their execution starts immediately; such nodes have zero startup cost.

Other operations, on the contrary, need to wait for some preliminary actions to complete. For example, a sort node usually has to wait for *all* the data from its child nodes before it can proceed to its own tasks. The startup cost of such nodes is usually higher than zero: this price has to be paid even if the above node (or the client) needs only one row of the whole output.

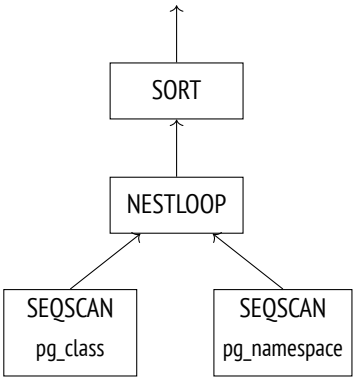
All calculations performed by the planner are simply estimations, which may have nothing to do with the actual execution time. Their only purpose is to enable comparison of different plans for *the same* query in *the same* conditions. In other cases, it makes no sense to compare queries (especially different ones) in terms of their cost. For example, the cost could have been underestimated because of outdated statistics; once the statistics are refreshed, the calculated figure may rise, but since the estimation becomes more accurate, the server will choose a better plan.

Execution

The plan built during query optimization now has to be *executed*.[1]

The executor opens a *portal* in the backend's memory;[2] it is an object that keeps the state of the query currently being executed. This state is represented as a tree

[1] postgresql.org/docs/14/executor.html
backend/executor/README
[2] backend/utils/mmgr/portalmem.c



that repeats the structure of the plan tree. The nodes of this tree operate like an assembly line, requesting and sending rows from one another.

Query execution starts at the root. The root node (which represents the SORT operation in this example) pulls the data from its child node. Having received all the rows, it sorts them and passes them on to the client.

Some nodes (like the NESTLOOP node shown in this illustration) join data sets received from different sources. Such a node pulls the data from two child nodes, and, having received a pair of rows that satisfy the join condition, passes the resulting row upwards right away (unlike sorting, which has to get *all* the rows first). At this point, the execution of the node is interrupted until its parent requests the next row. If only a partial result is required (for example, there is a LIMIT clause in the query), the operation will not be performed in full.

The two SEQSCAN leaf nodes of the tree are responsible for table scans. When the parent node requests the data from these nodes, they fetch the subsequent row from the corresponding table.

Thus, some nodes do not store any rows, passing them upwards immediately, but others (such as SORT) have to keep potentially large volumes of data. For this purpose, a *work_mem* chunk is allocated in the backend's memory; if it is not enough, the remaining data is spilled into temporary files on disk.[1] 4MB

A plan can have several nodes that need a data storage, so PostgreSQL may allocate several memory chunks, each of the *work_mem* size. The total size of RAM that a query can use is not limited in any way.

[1] backend/utils/sort/tuplestore.c

16.3 Extended Query Protocol

When using the simple query protocol, each command (even if it is being repeated multiple times) has to go through all the aforementioned stages:

1 parsing

2 transformation

3 planning

4 execution

However, there is no point in parsing one and the same query time and again. Repeated parsing of queries that differ only in constants does not make much sense either—the parse tree structure still remains the same.

Another downside of the simple query protocol is that the client receives the whole result at once, regardless of the number of rows it may contain.

In general, it is possible to get over these limitations using SQL commands. To deal with the first one, you can PREPARE the query before running the EXECUTE command; the second concern can be addressed by creating a cursor with DECLARE and returning rows via FETCH. But in this case, naming of these newly created objects must be handled by the client, while the server gets additional overhead of parsing extra commands.

The extended client-server protocol provides an alternative solution, enabling precise control over separate operator execution stages at the command level of the protocol itself.

Preparation

During the *preparation* stage, the query is parsed and transformed as usual, but the resulting parse tree is kept in the backend's memory.

PostgreSQL has no global cache for queries. The disadvantage of this architecture is obvious: each backend has to parse all the incoming queries, even if the same query has been already parsed by another backend. But there are some benefits

too. Global cache can easily become a bottleneck because of locks. A client running multiple small but different queries (like the ones varying only in constants) generates much traffic and can negatively affect performance of the whole instance. In PostgreSQL, queries are parsed locally, so there is no impact on other processes. p. 274

A prepared query can be parameterized. Here is a simple example using SQL commands (although it is not the same as preparation at the protocol level, the ultimate effect is the same):

```
=> PREPARE plane(text) AS
SELECT * FROM aircrafts WHERE aircraft_code = $1;
```

All the named prepared statements are shown in the pg_prepared_statements view:

```
=> SELECT name, statement, parameter_types
FROM pg_prepared_statements \gx
-[ RECORD 1 ]---+-------------------------------------------------
name            | plane
statement       | PREPARE plane(text) AS                          +
                | SELECT * FROM aircrafts WHERE aircraft_code = $1;
parameter_types | {text}
```

You will not find any unnamed statements here (the ones that use the extended query protocol or PL/pgSQL). The statements prepared by other backends are not displayed either: it is impossible to access the other session's memory.

Parameter Binding

Before a prepared statement gets executed, the actual parameter values have to be bound.

```
=> EXECUTE plane('733');
 aircraft_code |     model      | range
---------------+----------------+-------
 733           | Boeing 737-300 |  4200
(1 row)
```

The advantage of binding parameters in prepared statements over concatenating literals with query strings is that it makes SQL injections absolutely impossible: a bound parameter value cannot modify the already built parse tree in any way.

To reach the same security level without prepared statements, you would have to carefully escape each value received from an untrusted source.

Planning and Execution

When it comes to prepared statement execution, query planning is performed based on the actual parameter values; then the plan is passed on to the executor.

Different parameter values may imply different optimal plans, so it is important to take the exact values into account. For example, when looking for expensive bookings, the planner assumes that there are not so many matching rows and uses an index scan:

```
=> CREATE INDEX ON bookings(total_amount);
=> EXPLAIN SELECT * FROM bookings
WHERE total_amount > 1000000;
                            QUERY PLAN
-------------------------------------------------------------------
 Bitmap Heap Scan on bookings  (cost=82.13..9184.16 rows=4348 wid...
   Recheck Cond: (total_amount > '1000000'::numeric)
   -> Bitmap Index Scan on bookings_total_amount_idx  (cost=0.00....
       Index Cond: (total_amount > '1000000'::numeric)
(4 rows)
```

But if the provided condition is satisfied by all the bookings, there is no point in using an index, as the whole table has to be scanned:

```
=> EXPLAIN SELECT * FROM bookings WHERE total_amount > 100;
                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on bookings  (cost=0.00..39835.88 rows=2111110 width=21)
   Filter: (total_amount > '100'::numeric)
(2 rows)
```

In some cases, the planner may keep both the parse tree and the query plan to avoid repeated planning. Such a plan does not take parameter values into account, so it is called a *generic plan* (as compared to *custom plans* based on the actual values).[1]

[1] backend/utils/cache/plancache.c, choose_custom_plan function

An obvious case when the server can use a generic plan without compromising performance is a query with no parameters.

The first five optimizations of parameterized prepared statements always rely on the actual parameter values; the planner calculates the average cost of custom plans based on these values. Starting from the sixth execution, if the generic plan turns out to be more efficient than custom plans on average (taking into account that custom plans have to be built anew every time),[1] the planner keeps the generic plan and continues using it, skipping the optimization stage.

The plane prepared statement has already been executed once. After the next three executions, the server still uses custom plans—you can tell by the parameter value in the query plan:

```
=> EXECUTE plane('763');
=> EXECUTE plane('773');
=> EXPLAIN EXECUTE plane('319');
                            QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on aircrafts_data ml  (cost=0.00..1.39 rows=1 width=52)
   Filter: ((aircraft_code)::text = '319'::text)
(2 rows)
```

After the fifth execution, the planner switches to the generic plan: it does not differ from the custom ones and has the same cost, but the backend can build it once and skip the optimization stage, thus reducing planning overhead. The EXPLAIN command now shows that the parameter is referred to by position rather than by its value:

```
=> EXECUTE plane('320');
=> EXPLAIN EXECUTE plane('321');
                            QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on aircrafts_data ml  (cost=0.00..1.39 rows=1 width=52)
   Filter: ((aircraft_code)::text = $1)
(2 rows)
```

[1] backend/utils/cache/plancache.c, cached_plan_cost function

We can easily imagine an unhappy turn of events when the first several custom plans are more expensive than the generic plan; subsequent plans could have been more efficient, but the planner will not consider them at all. Besides, it compares *estimations* rather than actual costs, which can also lead to miscalculations.

v. 12 auto

However, if the planner makes a mistake, you can override the automatic decision and select either the generic or a custom plan by setting the *plan_cache_mode* parameter accordingly:

```
=> SET plan_cache_mode = 'force_custom_plan';
=> EXPLAIN EXECUTE plane('CN1');
                              QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on aircrafts_data ml  (cost=0.00..1.39 rows=1 width=52)
   Filter: ((aircraft_code)::text = 'CN1'::text)
(2 rows)
```

v. 14

Among other things, the pg_prepared_statements view shows statistics on chosen plans:

```
=> SELECT name, generic_plans, custom_plans
FROM pg_prepared_statements;
 name  | generic_plans | custom_plans
-------+---------------+--------------
 plane |             1 |            6
(1 row)
```

Getting the Results

The extended query protocol allows retrieving data in batches rather than all at once. SQL cursors have almost the same effect (except that there is some extra work for the server, and the planner optimizes fetching of the first *cursor_tuple_fraction* rows, not the whole result set):

```
=> BEGIN;
=> DECLARE cur CURSOR FOR
  SELECT *
  FROM aircrafts
  ORDER BY aircraft_code;
```

```
=> FETCH 3 FROM cur;
 aircraft_code |      model      | range
---------------+-----------------+-------
 319           | Airbus A319-100 |  6700
 320           | Airbus A320-200 |  5700
 321           | Airbus A321-200 |  5600
(3 rows)
=> FETCH 2 FROM cur;
 aircraft_code |     model      | range
---------------+----------------+-------
 733           | Boeing 737-300 |  4200
 763           | Boeing 767-300 |  7900
(2 rows)
=> COMMIT;
```

If the query returns many rows and the client needs them all, the system throughput highly depends on the batch size. The more rows in a batch, the less communication overhead is incurred by accessing the server and getting the response. But as the batch size grows, these benefits become less tangible: while the difference between fetching rows one by one and in batches of ten rows can be enormous, it is much less noticeable if you compare batches of 100 and 1000 rows.

17

Statistics

17.1 Basic Statistics

Basic relation-level statistics[1] are stored in the pg_class table of the system catalog and include the following data:

- number of tuples in a relation (reltuples)
- relation size, in pages (relpages)
- number of pages tagged in the visibility map (relallvisible)

p. 29

Here are these values for the flights table:

```
=> SELECT reltuples, relpages, relallvisible
FROM pg_class WHERE relname = 'flights';
 reltuples | relpages | relallvisible
-----------+----------+---------------
    214867 |     2624 |          2624
(1 row)
```

If the query does not impose any filter conditions, the reltuples value serves as the cardinality estimation:

```
=> EXPLAIN SELECT * FROM flights;
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=63)
(1 row)
```

[1] postgresql.org/docs/14/planner-stats.html

Statistics are collected during table analysis, both manual and automatic.[1] Furthermore, since basic statistics are of paramount importance, this data is calculated during some other operations as well (VACUUM FULL and CLUSTER,[2] CREATE INDEX and REINDEX[3]) and is refined during vacuuming.[4] p. 126

For analysis purposes, 300×*default_statistics_target* random rows are sampled. The sample size required to build statistics of a particular accuracy has low dependency on the volume of analyzed data, so the size of the table is not taken into account.[5] 100

Sampled rows are picked from the same number (300 × *default_statistics_target*) of random pages.[6] Obviously, if the table itself is smaller, fewer pages may be read, and fewer rows will be selected for analysis.

In large tables, statistics collection does not include all the rows, so estimations can diverge from actual values. It is perfectly normal: if the data is changing, statistics cannot be accurate all the time anyway. Accuracy of up to an order of magnitude is usually enough to choose an adequate plan.

Let's create a copy of the flights table with autovacuum disabled, so that we can control the autoanalysis start time:

```
=> CREATE TABLE flights_copy(LIKE flights)
WITH (autovacuum_enabled = false);
```

There is no statistics for the new table yet:

```
=> SELECT reltuples, relpages, relallvisible
FROM pg_class WHERE relname = 'flights_copy';
 reltuples | relpages | relallvisible
-----------+----------+---------------
        -1 |        0 |             0
(1 row)
```

[1] backend/commands/analyze.c, do_analyze_rel function
[2] backend/commands/cluster.c, copy_table_data function
[3] backend/catalog/heap.c, index_update_stats function
[4] backend/access/heap/vacuumlazy.c, heap_vacuum_rel function
[5] backend/commands/analyze.c, std_typanalyze function
[6] backend/commands/analyze.c, acquire_sample_rows function
backend/utils/misc/sampling.c

v. 14 The value reltuples = −1 is used to differentiate between a table that has not been analyzed yet and a really empty table without any rows.

It is highly likely that some rows will get inserted into the table right after its creation. So being unaware of the current state of things, the planner assumes that the table contains 10 pages:

```
=> EXPLAIN SELECT * FROM flights_copy;
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on flights_copy  (cost=0.00..14.10 rows=410 width=170)
(1 row)
```

The number of rows is estimated based on the size of a single row, which is shown in the plan as width. Row width is typically an average value calculated during analysis, but since no statistics have been collected yet, here it is just an approximation based on the column data types.[1]

Now let's copy the data from the flights table and perform the analysis:

```
=> INSERT INTO flights_copy SELECT * FROM flights;
INSERT 0 214867
=> ANALYZE flights_copy;
```

The collected statistics reflects the actual number of rows (the table size is small enough for the analyzer to gather statistics on all the data):

```
=> SELECT reltuples, relpages, relallvisible
FROM pg_class WHERE relname = 'flights_copy';
 reltuples | relpages | relallvisible
-----------+----------+---------------
    214867 |     2624 |             0
(1 row)
```

p. 381 The relallvisible value is used to estimate the cost of an index-only scan. This value is updated by VACUUM:

```
=> VACUUM flights_copy;
```

[1] backend/access/table/tableam.c, table_block_relation_estimate_size function

```
=> SELECT relallvisible FROM pg_class WHERE relname = 'flights_copy';
 relallvisible
---------------
          2624
(1 row)
```

Now let's double the number of rows without updating statistics and check the cardinality estimation in the query plan:

```
=> INSERT INTO flights_copy SELECT * FROM flights;
=> SELECT count(*) FROM flights_copy;
 count
--------
 429734
(1 row)
=> EXPLAIN SELECT * FROM flights_copy;
                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on flights_copy  (cost=0.00..9545.34 rows=429734 width=63)
(1 row)
```

Despite the outdated pg_class data, the estimation turns out to be accurate:

```
=> SELECT reltuples, relpages
FROM pg_class WHERE relname = 'flights_copy';
 reltuples | relpages
-----------+----------
    214867 |     2624
(1 row)
```

The thing is that if the planner sees a gap between relpages and the actual file size, it can scale the reltuples value to improve estimation accuracy.[1] Since the file size has doubled as compared to relpages, the planner adjusts the estimated number of rows, assuming that data density remains the same:

```
=> SELECT reltuples *
  (pg_relation_size('flights_copy') / 8192) / relpages AS tuples
FROM pg_class WHERE relname = 'flights_copy';
```

[1] backend/access/table/tableam.c, table_block_relation_estimate_size function

```
 tuples
--------
 429734
(1 row)
```

Naturally, such an adjustment may not always work (for example, if we delete some rows, the estimation will remain the same), but in some cases it allows the planner to hold on until significant changes trigger the next analysis run.

17.2 NULL Values

Frowned upon by theoreticians,[1] NULL values still play an important role in relational databases: they provide a convenient way to reflect the fact that a value is either unknown or does not exist.

However, a special value demands special treatment. Apart from theoretical inconsistencies, there are also multiple practical challenges that have to be taken into account. Regular Boolean logic is replaced by the three-valued one, so NOT IN behaves *unexpectedly*. It is unclear whether NULL values should be treated as greater than or less than regular values (hence the NULLS FIRST and NULLS LAST clauses for sorting). It is not quite obvious whether NULL values must be taken into account by aggregate functions. Strictly speaking, NULL values are not values at all, so the planner requires additional information to process them.

Apart from the simplest basic statistics collected at the relation level, the analyzer also gathers statistics for each column of the relation. This data is stored in the pg_statistic table of the system catalog,[2] but you can also access it via the pg_stats view, which provides this information in a more convenient format.

The *fraction of NULL values* belongs to column-level statistics; calculated during the analysis, it is shown as the null_frac attribute.

For example, when searching for the flights that have not departed yet, we can rely on their departure times being undefined:

```
=> EXPLAIN SELECT * FROM flights WHERE actual_departure IS NULL;
```

[1] sigmodrecord.org/publications/sigmodRecord/0809/p20.date.pdf
[2] include/catalog/pg_statistic.h

```
                          QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..4772.67 rows=15356 width=63)
   Filter: (actual_departure IS NULL)
(2 rows)
```

To estimate the result, the planner multiplies the total number of rows by the fraction of NULL values:

```
=> SELECT round(reltuples * s.null_frac) AS rows
FROM pg_class
  JOIN pg_stats s ON s.tablename = relname
WHERE s.tablename = 'flights'
  AND s.attname = 'actual_departure';
 rows
-------
 15356
(1 row)
```

And here is the actual row count:

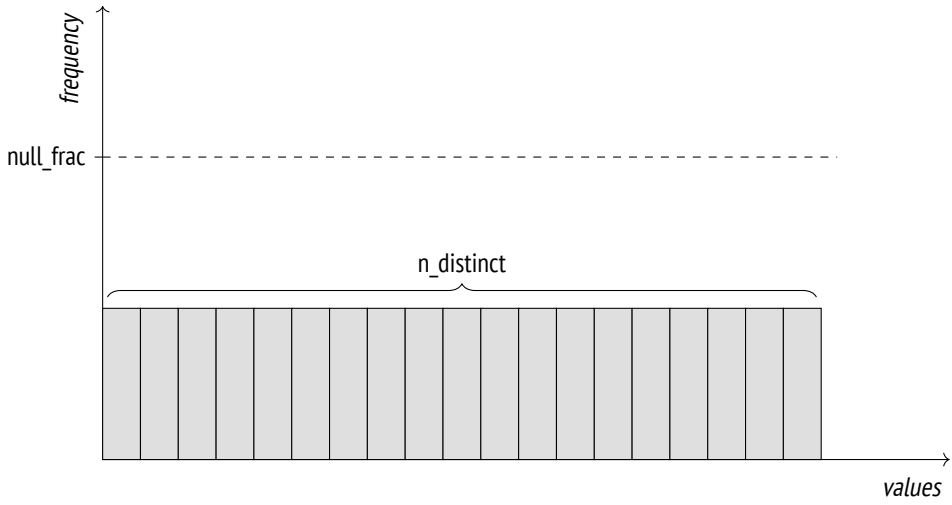
```
=> SELECT count(*) FROM flights WHERE actual_departure IS NULL;
 count
-------
 16348
(1 row)
```

17.3 Distinct Values

The n_distinct field of the pg_stats view shows the number of distinct values in a column.

If n_distinct is negative, its absolute value denotes the fraction of distinct values in a column rather than their actual count. For example, –1 indicates that all column values are unique, while –3 means that each value appears in three rows on average. The analyzer uses fractions if the estimated number of distinct values exceeds 10 % of the total row count; in this case, further data updates are unlikely to change this ratio.[1]

[1] backend/commands/analyze.c, compute_distinct_stats function



If uniform data distribution is expected, the number of distinct values is used instead. For example, when estimating the cardinality of the "*column = expression*" condition, the planner assumes that the *expression* can take any column value with equal probability if its exact value is unknown at the planning stage:[1]

```
=> EXPLAIN SELECT *
FROM flights
WHERE departure_airport = (
  SELECT airport_code FROM airports WHERE city = 'Saint Petersburg'
);
                              QUERY PLAN
----------------------------------------------------------------------
 Seq Scan on flights  (cost=30.56..5340.40 rows=2066 width=63)
   Filter: (departure_airport = $0)
   InitPlan 1 (returns $0)
     -> Seq Scan on airports_data ml  (cost=0.00..30.56 rows=1 wi...
          Filter: ((city ->> lang()) = 'Saint Petersburg'::text)
(5 rows)
```

Here the InitPlan node is executed only once, and the calculated value is used in the main plan.

[1] backend/utils/adt/selfuncs.c, var_eq_non_const function

```
=> SELECT round(reltuples / s.n_distinct) AS rows
FROM pg_class
  JOIN pg_stats s ON s.tablename = relname
WHERE s.tablename = 'flights'
  AND s.attname = 'departure_airport';
 rows
------
 2066
(1 row)
```

If the estimated number of distinct values is incorrect (because a limited number of rows have been analyzed), it can be overridden at the column level:

```
ALTER TABLE ...
  ALTER COLUMN ...
  SET (n_distinct = ...);
```

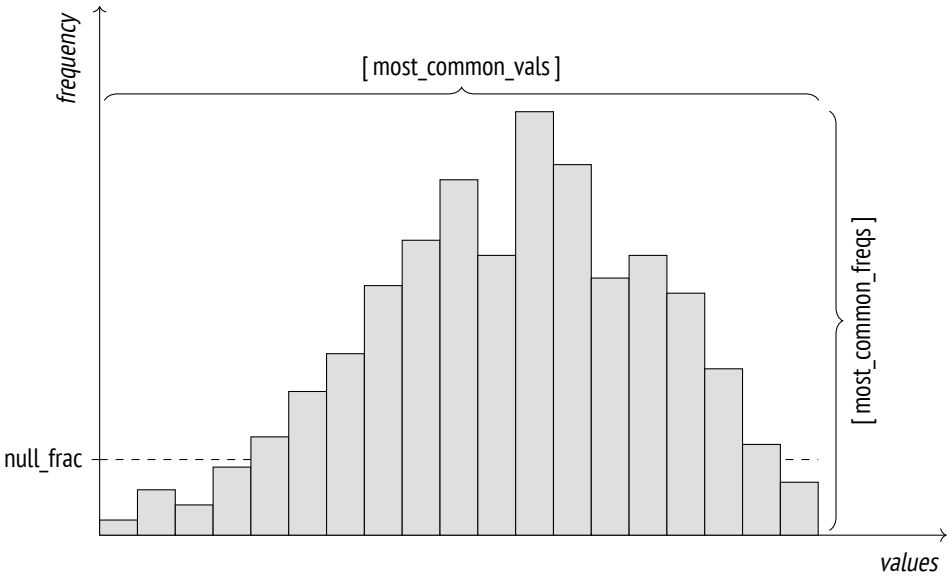
If all data always had uniform distribution, this information (coupled with minimal and maximal values) would be sufficient. However, for non-uniform distribution (which is much more common in practice), such estimation is inaccurate:

```
=> SELECT min(cnt), round(avg(cnt)) avg, max(cnt)
FROM (
  SELECT departure_airport, count(*) cnt
  FROM flights
  GROUP BY departure_airport
) t;
 min | avg  |  max
-----+------+-------
 113 | 2066 | 20875
(1 row)
```

17.4 Most Common Values

If data distribution is non-uniform, the estimation is fine-tuned based on statistics on most common values (MCV) and their frequencies. The pg_stats view displays these arrays in the most_common_vals and most_common_freqs fields, respectively.

Here is an example of such statistics on various types of aircraft:




```
=> SELECT most_common_vals AS mcv,
  left(most_common_freqs::text,60) || '...' AS mcf
FROM pg_stats
WHERE tablename = 'flights' AND attname = 'aircraft_code' \gx
-[ RECORD 1 ]-------------------------------------------------------
mcv | {CN1,CR2,SU9,321,763,733,319,773}
mcf | {0.27736667,0.27023333,0.26093334,0.0597,0.038266666,0.03796...
```

To estimate the selectivity of the "*column* = *value*" condition, it is enough to find this value in the most_common_vals array and take its frequency from the most_common_freqs array element with the same index:[1]

```
=> EXPLAIN SELECT * FROM flights WHERE aircraft_code = '733';
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..5309.84 rows=8158 width=63)
   Filter: (aircraft_code = '733'::bpchar)
(2 rows)
```

[1] backend/utils/adt/selfuncs.c, var_eq_const function

```
=> SELECT round(reltuples * s.most_common_freqs[
  array_position((s.most_common_vals::text::text[]),'733')
])
FROM pg_class
  JOIN pg_stats s ON s.tablename = relname
WHERE s.tablename = 'flights'
  AND s.attname = 'aircraft_code';
 round
-------
  8158
(1 row)
```

It is obvious that such estimation will be close to the actual value:

```
=> SELECT count(*) FROM flights WHERE aircraft_code = '733';
 count
-------
  8263
(1 row)
```

The MCV list is also used to estimate selectivity of inequality conditions. For example, a condition like "*column* < *value*" requires the analyzer to search through most_common_vals for all the values that are smaller than the target one and sum up the corresponding frequencies listed in most_common_freqs.[1]

MCV statistics work best when distinct values are not too many. The maximum size of arrays is defined by the *default_statistics_target* parameter, which also limits the number of rows to be randomly sampled for the purpose of analysis. 100

In some cases, it makes sense to increase the default parameter value, thus expanding the MCV list and improving the accuracy of estimations. You can do it at the column level:

```
ALTER TABLE ...
  ALTER COLUMN ...
  SET STATISTICS ...;
```

The sample size will also grow, but only for the specified table.

[1] backend/utils/adt/selfuncs.c, scalarineqsel function

Since the MCV array stores actual values, it may take quite a lot of space. To keep the pg_statistic size under control and avoid loading the planner with useless work, values that are larger than 1 kB are excluded from analysis and statistics. But since such large values are likely to be unique, they would probably not make it into most_common_vals anyway.

17.5 Histogram

If distinct values are too many to be stored in an array, PostgreSQL employs a histogram. In this case, values are distributed between several *buckets* of the histogram. The number of buckets is also limited by the *default_statistics_target* parameter.

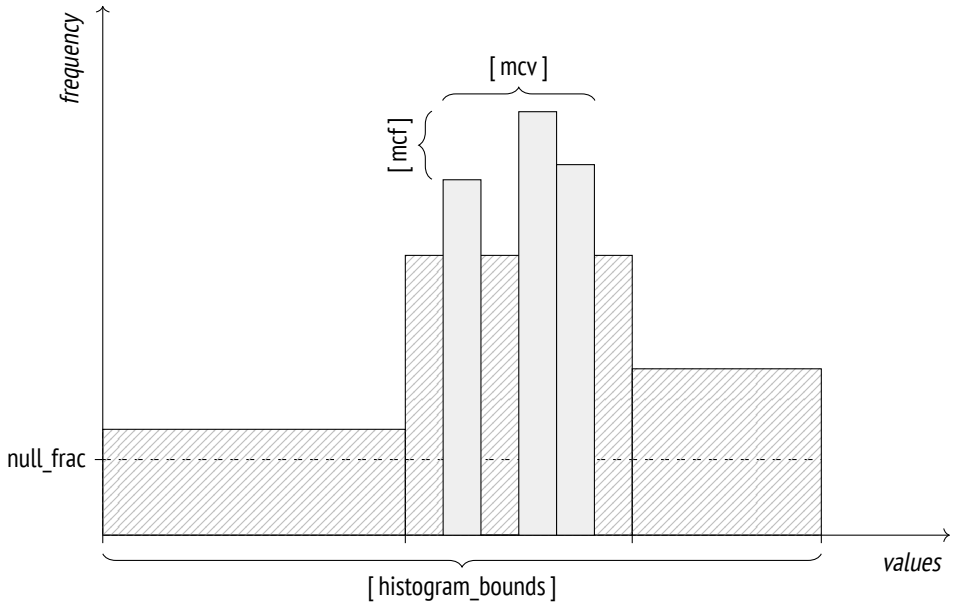
The bucket width is selected in such a way that each bucket gets approximately the same number of values (this property is reflected in the diagram by the equality of areas of big hatched rectangles). The values included into MCV lists are not taken into account. As a result, the cumulative frequency of values in each bucket equals $\frac{1}{\textit{number of buckets}}$.

The histogram is stored in the histogram_bounds field of the pg_stats view as an array of buckets' boundary values:

```
=> SELECT left(histogram_bounds::text,60) || '...' AS hist_bounds
FROM pg_stats s
WHERE s.tablename = 'boarding_passes' AND s.attname = 'seat_no';
                           hist_bounds
-------------------------------------------------------------
 {10A,10A,11B,11H,12G,13B,14B,14H,15G,16B,17B,17H,19B,19B,19J...
(1 row)
```

Combined with the MCV list, the histogram is used for operations like estimating the selectivity of *greater than* and *less than* conditions.[1] For example, let's take a look at the number of boarding passes issued for back rows:

[1] backend/utils/adt/selfuncs.c, ineq_histogram_selectivity function




```
=> EXPLAIN SELECT * FROM boarding_passes WHERE seat_no > '30C';
                              QUERY PLAN
----------------------------------------------------------------------
 Seq Scan on boarding_passes  (cost=0.00..157350.10 rows=3014932 ...
   Filter: ((seat_no)::text > '30C'::text)
(2 rows)
```

I have intentionally selected the seat number that lies right on the boundary between two histogram buckets.

The selectivity of this condition will be estimated at $\frac{N}{number\ of\ buckets}$, where N is the number of buckets holding the values that satisfy the condition (that is, the ones located to the right of the specified value). It must also be taken into account that MCVs are not included into the histogram.

Incidentally, NULL values do not appear in the histogram either, but the seat_no column contains no such values anyway:

```
=> SELECT s.null_frac FROM pg_stats s
WHERE s.tablename = 'boarding_passes' AND s.attname = 'seat_no';
```

```
 null_frac
-----------
         0
(1 row)
```

First, let's find the fraction of MCVs that satisfy the condition:

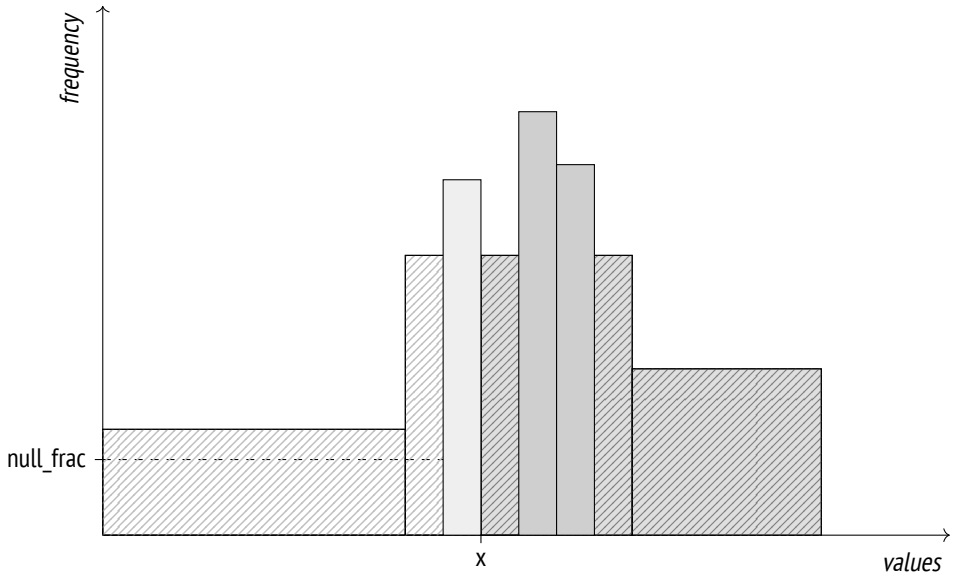
```
=> SELECT sum(s.most_common_freqs[
  array_position((s.most_common_vals::text::text[]),v)
])
FROM pg_stats s, unnest(s.most_common_vals::text::text[]) v
WHERE s.tablename = 'boarding_passes' AND s.attname = 'seat_no'
  AND v > '30C';
  sum
--------
 0.2172
(1 row)
```

The overall MCV share (ignored by the histogram) is:

```
=> SELECT sum(s.most_common_freqs[
  array_position((s.most_common_vals::text::text[]),v)
])
FROM pg_stats s, unnest(s.most_common_vals::text::text[]) v
WHERE s.tablename = 'boarding_passes' AND s.attname = 'seat_no';
    sum
-----------
 0.6735997
(1 row)
```

Since the values that conform to the specified condition take exactly N buckets (out of 100 buckets possible), we get the following estimation:

```
=> SELECT round( reltuples * (
    0.2172 -- MCV share
  + (1 - 0.6735997 - 0) * (50 / 100.0) -- histogram share
))
FROM pg_class
WHERE relname = 'boarding_passes';
  round
---------
 3014933
(1 row)
```



In the generic case of non-boundary values, the planner applies linear interpolation to take into account the fraction of the bucket that contains the target value.

Here is the actual number of back seats:

```
=> SELECT count(*) FROM boarding_passes WHERE seat_no > '30C';
  count
---------
 2986429
(1 row)
```

As you increase the *default_statistics_target* value, estimation accuracy may improve, but as our example shows, the histogram combined with the MCV list usually gives good results even if the column contains many unique values:

```
=> SELECT n_distinct FROM pg_stats
WHERE tablename = 'boarding_passes' AND attname = 'seat_no';
 n_distinct
------------
        461
(1 row)
```

It makes sense to improve estimation accuracy only if it leads to better planning. Increasing the *default_statistics_target* value without giving it much thought may

slow down planning and analysis without bringing any benefits in return. That said, reducing this parameter value (down to zero) can lead to a bad plan choice, even though it does speed up planning and analysis. Such savings are usually unjustified.

17.6 Statistics for Non-Scalar Data Types

For non-scalar data types, PostgreSQL can gather statistics not only on the distribution of values, but also on the distribution of elements used to construct these values. It improves planning accuracy when you query columns that do not conform to the first normal form.

- The most_common_elems and most_common_elem_freqs arrays show the list of the *most common elements* and the frequency of their usage.

  These statistics are collected and used to estimate selectivity of operations on arrays[1] and tsvector[2] data types.

- The elem_count_histogram array shows the histogram of the *number of distinct elements* in a value.

  This data is collected and used for estimating selectivity of operations on arrays only.

- For range types, PostgreSQL builds distribution histograms for range length and lower and upper boundaries of the range. These histograms are used for estimating selectivity of various operations on these types,[3] but the pg_stats view does not display them.

v. 14 Similar statistics are also collected for multirange data types.[4]

[1] postgresql.org/docs/14/arrays.html
backend/utils/adt/array_typanalyze.c
backend/utils/adt/array_selfuncs.c

[2] postgresql.org/docs/14/datatype-textsearch.html
backend/tsearch/ts_typanalyze.c
backend/tsearch/ts_selfuncs.c

[3] postgresql.org/docs/14/rangetypes.html
backend/utils/adt/rangetypes_typanalyze.c
backend/utils/adt/rangetypes_selfuncs.c

[4] backend/utils/adt/multirangetypes_selfuncs.c

17.7 Average Field Width

The avg_width field of the pg_stats view shows the average size of values stored in a column. Naturally, for types like integer or char(3) this size is always the same, but for data types of variable length, such as text, it can vary a lot from column to column:

```
=> SELECT attname, avg_width FROM pg_stats
WHERE (tablename, attname) IN ( VALUES
  ('tickets', 'passenger_name'), ('ticket_flights','fare_conditions')
);
     attname     | avg_width
-----------------+-----------
 fare_conditions |         8
 passenger_name  |        16
(2 rows)
```

This statistic is used to estimate the amount of memory required for operations like sorting or hashing.

17.8 Correlation

The correlation field of the pg_stats view shows the correlation between the physical order of data and the logical order defined by comparison operations. If values are stored strictly in ascending order, their correlation will be close to 1; if they are arranged in descending order, their correlation will be close to −1. The more chaotic is data distribution on disk, the closer is the correlation to zero.

```
=> SELECT attname, correlation
FROM pg_stats WHERE tablename = 'airports_data'
ORDER BY abs(correlation) DESC;
   attname    | correlation
--------------+-------------
 coordinates  |
 airport_code | -0.21120238
 city         |  -0.1970127
 airport_name | -0.18223621
 timezone     |  0.17961165
(5 rows)
```

Note that this statistic is not gathered for the coordinates column: *less than* and *greater than* operators are not defined for the point type.

p. 374 Correlation is used for cost estimation of index scans.

17.9 Expression Statistics

Column-level statistics can be used only if either the left or the right part of the comparison operation refers to the column itself and does not contain any expressions. For example, the planner cannot predict how computing a function of a column will affect statistics, so for conditions like "*function-call = constant*" the selectivity is always estimated at 0.5 %:[1]

```
=> EXPLAIN SELECT * FROM flights
WHERE extract(
  month FROM scheduled_departure AT TIME ZONE 'Europe/Moscow'
) = 1;
                              QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..6384.17 rows=1074 width=63)
   Filter: (EXTRACT(month FROM (scheduled_departure AT TIME ZONE ...
(2 rows)
=> SELECT round(reltuples * 0.005)
FROM pg_class WHERE relname = 'flights';
 round
-------
  1074
(1 row)
```

The planner knows nothing about semantics of functions, even of standard ones. Our general knowledge suggests that the flights performed in January will make roughly $\frac{1}{12}$ of the total number of flights, which exceeds the projected value by one order of magnitude.

To improve the estimation, we have to collect expression statistics rather than rely on the column-level one. There are two ways to do it.

[1] backend/utils/adt/selfuncs.c, eqsel function

Extended Expression Statistics

v. 14

The first option is to use *extended expression statistics*.[1] Such statistics are not collected by default; you have to manually create the corresponding database object by running the CREATE STATISTICS command:

```
=> CREATE STATISTICS flights_expr ON (extract(
    month FROM scheduled_departure AT TIME ZONE 'Europe/Moscow'
))
FROM flights;
```

Once the data is gathered, the estimation accuracy improves:

```
=> ANALYZE flights;
=> EXPLAIN SELECT * FROM flights
WHERE extract(
  month FROM scheduled_departure AT TIME ZONE 'Europe/Moscow'
) = 1;
                              QUERY PLAN
----------------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..6384.17 rows=16581 width=63)
   Filter: (EXTRACT(month FROM (scheduled_departure AT TIME ZONE ...
(2 rows)
```

For the collected statistics to be applied, the query must specify the expression in exactly the same form that was used by the CREATE STATISTICS command.

The size limit for extended statistics can be adjusted separately, by running the ALTER STATISTICS command. For example: v. 13

```
=> ALTER STATISTICS flights_expr SET STATISTICS 42;
```

All the metadata related to extended statistics is stored in the pg_statistic_ext table of the system catalog, while the collected data itself resides in a separate table called pg_statistic_ext_data. This separation is used to implement access control for sensitive information. v. 12

Extended expression statistics available to a particular user can be displayed in a more convenient format in a separate view:

[1] postgresql.org/docs/14/planner-stats#PLANNER-STATS-EXTENDED.html
backend/statistics/README

```
=> SELECT left(expr,50) || '...' AS expr,
  null_frac, avg_width, n_distinct,
  most_common_vals AS mcv,
  left(most_common_freqs::text,50) || '...' AS mcf,
  correlation
FROM pg_stats_ext_exprs
WHERE statistics_name = 'flights_expr' \gx
-[ RECORD 1 ]-----------------------------------------------------
expr        | EXTRACT(month FROM (scheduled_departure AT TIME ZO...
null_frac   | 0
avg_width   | 8
n_distinct  | 12
mcv         | {8,9,12,7,3,10,1,5,4,11,6,2}
mcf         | {0.12303333,0.11033333,0.080133334,0.0793,0.078966...
correlation | 0.08339994
```

Statistics for Expression Indexes

Another way to improve cardinality estimation is to use special statistics collected
p. 361 for expression indexes; these statistics are gathered automatically when such an index is created, just like it is done for a table. If the index is really needed, this approach turns out to be very convenient.

```
=> DROP STATISTICS flights_expr;

=> CREATE INDEX ON flights(extract(
  month FROM scheduled_departure AT TIME ZONE 'Europe/Moscow'
));

=> ANALYZE flights;

=> EXPLAIN SELECT * FROM flights
WHERE extract(
  month FROM scheduled_departure AT TIME ZONE 'Europe/Moscow'
) = 1;
                              QUERY PLAN
----------------------------------------------------------------------
 Bitmap Heap Scan on flights  (cost=310.98..3220.75 rows=16330 wi...
   Recheck Cond: (EXTRACT(month FROM (scheduled_departure AT TIME...
   -> Bitmap Index Scan on flights_extract_idx  (cost=0.00..306.9...
       Index Cond: (EXTRACT(month FROM (scheduled_departure AT TI...
(4 rows)
```

Statistics on expression indexes are stored in the same way as statistics on tables. For example, you can get the number of distinct values by specifying the index name as tablename when querying pg_stats:

```
=> SELECT n_distinct FROM pg_stats
WHERE tablename = 'flights_extract_idx';
 n_distinct
------------
         12
(1 row)
```

You can adjust the accuracy of index-related statistics using the ALTER INDEX command. If you do not know the column name that corresponds to the indexed expression, you have to first find it out. For example: v. 11

```
=> SELECT attname FROM pg_attribute
WHERE attrelid = 'flights_extract_idx'::regclass;
 attname
---------
 extract
(1 row)
=> ALTER INDEX flights_extract_idx
  ALTER COLUMN extract SET STATISTICS 42;
```

17.10 Multivariate Statistics

It is also possible to collect *multivariate statistics*, which span several table columns. As a prerequisite, you have to manually create the corresponding extended statistics using the CREATE STATISTICS command.

PostgreSQL implements three types of multivariate statistics.

Functional Dependencies Between Columns

v. 10

If values in one column depend (fully or partially) on values in another column and the filter conditions include both these columns, cardinality will be underestimated.

Let's consider a query with two filter conditions:

```
=> SELECT count(*) FROM flights
WHERE flight_no = 'PG0007' AND departure_airport = 'VKO';
 count
-------
   396
(1 row)
```

The value is hugely underestimated:

```
=> EXPLAIN SELECT * FROM flights
WHERE flight_no = 'PG0007' AND departure_airport = 'VKO';
                              QUERY PLAN
----------------------------------------------------------------------
 Bitmap Heap Scan on flights  (cost=10.49..816.84 rows=14 width=63)
   Recheck Cond: (flight_no = 'PG0007'::bpchar)
   Filter: (departure_airport = 'VKO'::bpchar)
   -> Bitmap Index Scan on flights_flight_no_scheduled_departure_key
       (cost=0.00..10.49 rows=276 width=0)
       Index Cond: (flight_no = 'PG0007'::bpchar)
(6 rows)
```

It is a well-known *problem of correlated predicates*. The planner assumes that predicates do not depend on each other, so the overall selectivity is estimated at the product of selectivities of filter conditions combined by logical AND. The plan above clearly illustrates this issue: the value estimated by the Bitmap Index Scan node for the condition on the flight_no column is significantly reduced once the Bitmap Heap Scan node filters the results by the condition on the departure_airport column.

p. 299

However, we do understand that airports are unambiguously defined by flight numbers: the second condition is virtually redundant (unless there is a mistake in the airport name, of course). In such cases, we can improve the estimation by applying extended statistics on functional dependencies.

Let's create an extended statistic on the functional dependency between the two columns:

```
=> CREATE STATISTICS flights_dep(dependencies)
ON flight_no, departure_airport FROM flights;
```

The next analysis run gathers this statistic, and the estimation improves:

```
=> ANALYZE flights;
=> EXPLAIN SELECT * FROM flights
WHERE flight_no = 'PG0007'
  AND departure_airport = 'VKO';
                              QUERY PLAN
----------------------------------------------------------------------
 Bitmap Heap Scan on flights  (cost=10.56..816.91 rows=276 width=63)
   Recheck Cond: (flight_no = 'PG0007'::bpchar)
   Filter: (departure_airport = 'VKO'::bpchar)
   -> Bitmap Index Scan on flights_flight_no_scheduled_departure_key
       (cost=0.00..10.49 rows=276 width=0)
       Index Cond: (flight_no = 'PG0007'::bpchar)
(6 rows)
```

The collected statistics is stored in the system catalog and can be accessed like this:

```
=> SELECT dependencies
FROM pg_stats_ext WHERE statistics_name = 'flights_dep';
               dependencies
-------------------------------------------
 {"2 => 5": 1.000000, "5 => 2": 0.010200}
(1 row)
```

Here 2 and 5 are column numbers stored in the pg_attribute table, whereas the corresponding values define the degree of functional dependency: from 0 (no dependency) to 1 (values in the second columns fully depend on values in the first column).

Multivariate Number of Distinct Values

v. 10

Statistics on the number of unique combinations of values stored in different columns improves cardinality estimation of a GROUP BY operation performed on several columns.

For example, here the estimated number of possible pairs of departure and arrival airports is the square of the total number of airports; however, the actual value is much smaller, as not all the pairs are connected by direct flights:

```
=> SELECT count(*)
FROM (
  SELECT DISTINCT departure_airport, arrival_airport FROM flights
) t;
 count
-------
   618
(1 row)
=> EXPLAIN SELECT DISTINCT departure_airport, arrival_airport
FROM flights;
                              QUERY PLAN
----------------------------------------------------------------------
 HashAggregate  (cost=5847.01..5955.16 rows=10816 width=8)
   Group Key: departure_airport, arrival_airport
   -> Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=8)
(3 rows)
```

Let's define and collect an extended statistic on distinct values:

```
=> CREATE STATISTICS flights_nd(ndistinct)
ON departure_airport, arrival_airport FROM flights;
=> ANALYZE flights;
```

The cardinality estimation has improved:

```
=> EXPLAIN SELECT DISTINCT departure_airport, arrival_airport
FROM flights;
                              QUERY PLAN
----------------------------------------------------------------------
 HashAggregate  (cost=5847.01..5853.19 rows=618 width=8)
   Group Key: departure_airport, arrival_airport
   -> Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=8)
(3 rows)
```

You can view the collected statistic in the system catalog:

```
=> SELECT n_distinct
FROM pg_stats_ext WHERE statistics_name = 'flights_nd';
  n_distinct
---------------
 {"5, 6": 618}
(1 row)
```

Multivariate MCV Lists

v. 12

If the distribution of values is non-uniform, it may be not enough to rely on the functional dependency alone, as the estimation accuracy will highly depend on a particular pair of values. For example, the planner underestimates the number of flights performed by Boeing 737 from Sheremetyevo airport:

```
=> SELECT count(*) FROM flights
WHERE departure_airport = 'SVO' AND aircraft_code = '733';
 count
-------
  2037
(1 row)
=> EXPLAIN SELECT * FROM flights
WHERE departure_airport = 'SVO' AND aircraft_code = '733';
                              QUERY PLAN
----------------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..5847.00 rows=708 width=63)
   Filter: ((departure_airport = 'SVO'::bpchar) AND (aircraft_cod...
(2 rows)
```

In this case, you can improve the estimation by collecting statistics on multivariate MCV lists:[1]

```
=> CREATE STATISTICS flights_mcv(mcv)
ON departure_airport, aircraft_code FROM flights;
=> ANALYZE flights;
```

The new cardinality estimation is much more accurate:

```
=> EXPLAIN SELECT * FROM flights
WHERE departure_airport = 'SVO' AND aircraft_code = '733';
                              QUERY PLAN
----------------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..5847.00 rows=2134 width=63)
   Filter: ((departure_airport = 'SVO'::bpchar) AND (aircraft_cod...
(2 rows)
```

[1] backend/statistics/README.mcv
backend/statistics/mcv.c

To get this estimation, the planner relies on the frequency values stored in the system catalog:

```
=> SELECT values, frequency
FROM pg_statistic_ext stx
  JOIN pg_statistic_ext_data stxd ON stx.oid = stxd.stxoid,
  pg_mcv_list_items(stxdmcv) m
WHERE stxname = 'flights_mcv'
AND values = '{SVO,773}';
  values   | frequency
-----------+-----------
 {SVO,773} |     0.005
(1 row)
```

100 Just like a regular MCV list, a multivariate list holds *default_statistics_target* values (if this parameter is also set at the column level, the largest of its values is used).

v. 13 If required, you can also change the size of the list, like it is done for extended expression statistics:

```
ALTER STATISTICS ... SET STATISTICS ...;
```

In all these examples, I have used only two columns, but you can collect multivariate statistics on a larger number of columns too.

To combine statistics of several types in one object, you can provide a comma-separated list of these types in its definition. If no type is specified, PostgreSQL will collect statistics of all the possible types for the specified columns.

v. 14 Apart from the actual column names, multivariate statistics can also use arbitrary expressions, just like expression statistics.

18

Table Access Methods

18.1 Pluggable Storage Engines

The data layout used by PostgreSQL is neither the only possible nor the best one for all load types. Following the idea of extensibility, PostgreSQL allows you to create and plug in various *table access methods* (pluggable storage engines), but there is only one available out of the box at the moment: v. 12

```
=> SELECT amname, amhandler FROM pg_am WHERE amtype = 't';
 amname |      amhandler
--------+----------------------
 heap   | heap_tableam_handler
(1 row)
```

You can specify the engine to use when creating a table (CREATE TABLE ... USING); otherwise, the default engine listed in the *default_table_access_method* parameter will be applied. heap

For the PostgreSQL core to work with various engines in the same way, table access methods must implement a special interface.[1] The function specified in the amhandler column returns the interface structure[2] that contains all the information required by the core.

The following core components can be used by all table access methods:

- transaction manager, including ACID and snapshot isolation support
- buffer manager

[1] postgresql.org/docs/14/tableam.html
[2] include/access/tableam.h

- I/O subsystem
- TOAST
- optimizer and executor
- index support

These components always remain at the disposal of the engine, even if it does not use them all.

In their turn, engines define:

- tuple format and data structure
- table scan implementation and cost estimation
- implementation of insert, delete, update, and lock operations
- visibility rules
- vacuum and analysis procedures

Historically, PostgreSQL used a single built-in data storage without any proper programming interface, so now it is very hard to come up with a good design that takes all the specifics of the standard engine into account and does not interfere with other methods.

> For example, it is still unclear how to deal with the WAL. New access methods may need to log their own operations that the core is unaware of. The existing generic WAL mechanism[1] is usually a bad choice, as it incurs too much overhead. You can add yet another interface for handling new types of WAL entries, but then crash recovery will depend on external code, which is highly undesirable. The only plausible solution so far is patching the core for each particular engine.

For this reason, I did not strive to provide any strict distinction between table access methods and the core. Many features described in the previous parts of the book formally belong to the heap access method rather than to the core itself. This method is likely to always remain the ultimate standard engine for PostgreSQL, while other methods will fill separate niches to address challenges of specific load types.

[1] postgresql.org/docs/14/generic-wal.html

Of all the new engines that are currently being developed, I would like to mention the following:

Zheap is aimed at fighting table bloating.[1] It implements in-place row updates and moves historic MVCC-related data into a separate undo storage. Such an engine will be useful for loads that involve frequent data updates.

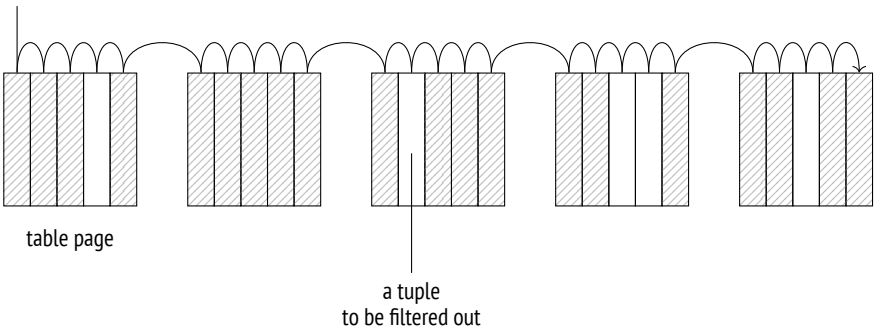
Zheap architecture will seem familiar to Oracle users, although it does have some nuances (for example, the interface of index access methods does not allow creating indexes with their own versioning). p. 354

Zedstore implements columnar storage,[2] which is likely to be most efficient with OLAP queries.

The stored data is structured as a B-tree of tuple IDs; each column is stored in its own B-tree associated with the main one. In the future, it might be possible to store several columns in one B-tree, thus getting a hybrid storage.

18.2 Sequential Scans

The storage engine defines the physical layout of table data and provides an access method to it. The only supported method is a sequential scan, which reads the file (or files) of the table's main fork in full. In each read page, the visibility of each tuple is checked; those tuples that do not satisfy the query are filtered out. p. 92



[1] github.com/EnterpriseDB/zheap
[2] github.com/greenplum-db/postgres/tree/zedstore

p. 179 A scanning process goes through the buffer cache; to ensure that large tables do not oust useful data, a small-sized buffer ring is employed. Other processes that are scanning the same table join this buffer ring, thus avoiding extra disk reads; such scans are called *synchronized*. Thus, scanning does not always have to begin at the start of the file.

Sequential scanning is the most efficient way to read the whole table or the best part of it. In other words, sequential scans bring the most value when the selectivity is low. (If the selectivity is high, meaning that the query has to select only a few rows, it is preferable to use an index.) p. 354

Cost Estimation

In the query execution plan, a sequential scan is represented by the Seq Scan node:

```
=> EXPLAIN SELECT *
FROM flights;
                            QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=63)
(1 row)
```

The estimated number of rows is provided as part of the basic statistics:

```
=> SELECT reltuples FROM pg_class WHERE relname = 'flights';
 reltuples
-----------
    214867
(1 row)
```

When estimating the cost, the optimizer takes the following two components into account: disk I/O and CPU resources.[1]

I/O cost is calculated by multiplying the number of pages in a table and the cost of reading a single page *assuming that pages are being read sequentially*. When

[1] backend/optimizer/path/costsize.c, cost_seqscan function

the buffer manager requests a page, the operating system actually reads more data from disk, so several subsequent pages are highly likely to be found in the operating system cache. For this reason, the cost of reading a single page using sequential scanning (which the planner estimates at *seq_page_cost*) is 1
lower than the random access cost (defined by the *random_page_cost* value). 4

The default settings work well for HDDs; if you are using SSDs, it makes sense to significantly reduce the *random_page_cost* value (the *seq_page_cost* parameter is usually left as is, serving as a reference value). Since the optimal ratio between these parameters depends on the hardware, they are usually set at the tablespace level (ALTER TABLESPACE ... SET).

```
=> SELECT relpages,
  current_setting('seq_page_cost') AS seq_page_cost,
  relpages * current_setting('seq_page_cost')::real AS total
FROM pg_class WHERE relname = 'flights';
 relpages | seq_page_cost | total
----------+---------------+-------
     2624 | 1             |  2624
(1 row)
```

These calculations clearly show the consequences of table bloating caused by untimely vacuuming: the larger the main fork of the table, the more pages have to be scanned, regardless of the number of live tuples they contain. *p. 156*

CPU resource estimation comprises the costs of processing each tuple (which the planner estimates at *cpu_tuple_cost*): 0.01

```
=> SELECT reltuples,
  current_setting('cpu_tuple_cost') AS cpu_tuple_cost,
  reltuples * current_setting('cpu_tuple_cost')::real AS total
FROM pg_class WHERE relname = 'flights';
 reltuples | cpu_tuple_cost |  total
-----------+----------------+---------
    214867 | 0.01           | 2148.67
(1 row)
```

The sum of these two estimates represents the total cost of the plan. The startup cost is zero because sequential scans have no prerequisites.

If the scanned table needs to be filtered, the applied filter conditions appear in
p. 308 the plan under the Filter section of the Seq Scan node. The estimated row count depends on the selectivity of these conditions, while the cost estimation includes the related computation expenses.

The EXPLAIN ANALYZE command displays both the actual number of returned rows and the number of rows that have been filtered out:

```
=> EXPLAIN (analyze, timing off, summary off)
SELECT * FROM flights
WHERE status = 'Scheduled';
                    QUERY PLAN
-----------------------------------------------
 Seq Scan on flights
   (cost=0.00..5309.84 rows=15383 width=63)
   (actual rows=15383 loops=1)
   Filter: ((status)::text = 'Scheduled'::text)
   Rows Removed by Filter: 199484
(5 rows)
```

Let's take a look at a more complex execution plan that uses aggregation:

```
=> EXPLAIN SELECT count(*) FROM seats;
                           QUERY PLAN
-------------------------------------------------------------
 Aggregate  (cost=24.74..24.75 rows=1 width=8)
   -> Seq Scan on seats  (cost=0.00..21.39 rows=1339 width=0)
(2 rows)
```

The plan consists of two nodes: the upper node (Aggregate), which computes the count function, pulls the data from the lower node (Seq Scan), which scans the table.

The startup cost of the Aggregate node includes the aggregation itself: it is impossible to return the first row (which is the only one in this case) without getting all the rows from the lower node. The aggregation cost is estimated based on the
0.0025 execution cost of a conditional operation (estimated at *cpu_operator_cost*) for each input row:[1]

[1] backend/optimizer/path/costsize.c, cost_agg function

```
=> SELECT reltuples,
  current_setting('cpu_operator_cost') AS cpu_operator_cost,
  round((
    reltuples * current_setting('cpu_operator_cost')::real
  )::numeric, 2) AS cpu_cost
FROM pg_class WHERE relname = 'seats';
 reltuples | cpu_operator_cost | cpu_cost
-----------+-------------------+----------
      1339 | 0.0025            |     3.35
(1 row)
```

The received estimate is added to the total cost of the Seq Scan node.

The total cost of the Aggregate node also includes the cost of processing a row to be returned, which is estimated at *cpu_tuple_cost*: 0.01

```
=> WITH t(cpu_cost) AS (
  SELECT round((
    reltuples * current_setting('cpu_operator_cost')::real
  )::numeric, 2)
  FROM pg_class WHERE relname = 'seats'
)
SELECT 21.39 + t.cpu_cost AS startup_cost,
  round((
    21.39 + t.cpu_cost +
    1 * current_setting('cpu_tuple_cost')::real
  )::numeric, 2) AS total_cost
FROM t;
 startup_cost | total_cost
--------------+------------
        24.74 |      24.75
(1 row)
```

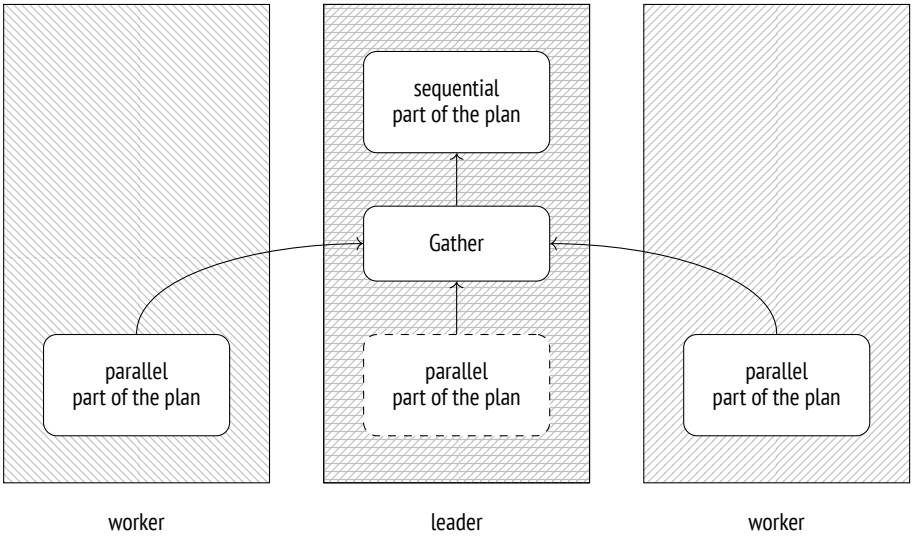
Thus, cost estimation dependencies can be pictured as follows:

```
                  QUERY PLAN
-------------------------------------------
 Aggregate
   (cost=24.74..24.75 rows=1 width=8)
   -> Seq Scan on seats
       (cost=0.00..21.39 rows=1339 width=0)
(4 rows)
```

18.3 Parallel Plans

v. 9.6 PostgreSQL supports parallel query execution.[1] The leading process that performs the query spawns (via postmaster) several worker processes that execute one and the same parallel part of the plan simultaneously. The results are passed to the leader, which puts them together in the Gather[2] node. When not accepting the data, the leader may also participate in the execution of the parallel part of the plan.

v. 11 on If required, you can forbid the leader's contributions to the parallel plan execution by turning off the *parallel_leader_participation* parameter.



Naturally, starting these processes and sending data between them is not free, so not all queries by far should be parallelized.

Besides, not all parts of the plan can be processed concurrently, even if parallel execution is allowed. Some of the operations are performed by the leader alone, in the sequential mode.

[1] postgresql.org/docs/14/parallel-query.html
backend/access/transam/README.parallel
[2] backend/executor/nodeGather.c

PostgreSQL does not support the other approach to parallel plan execution, which consists in performing data processing by several workers that virtually form an assembly line (roughly speaking, each plan node is performed by a separate process); this mechanism was deemed inefficient by PostgreSQL developers.

18.4 Parallel Sequential Scans

One of the nodes designed for parallel processing is the Parallel Seq Scan node, which performs a *parallel sequential scan*.

The name sounds a bit controversial (is the scan sequential or parallel after all?), but nevertheless, it reflects the essence of the operation. If we take a look at the file access, table pages are read sequentially, following the order in which they would have been read by a simple sequential scan. However, this operation is performed by several concurrent processes. To avoid scanning one and the same page twice, the executor synchronizes these processes via shared memory.

A subtle aspect here is that the operating system does not get the big picture typical of sequential scanning; instead, it sees several processes that perform random reads. Therefore, data prefetching that usually speeds up sequential scans becomes virtually useless. To minimize this unpleasant effect, PostgreSQL assigns each process not just one but several consecutive pages to read.[1] v. 14

As such, parallel scanning does not make much sense because the usual read costs are further increased by the overhead incurred by data transfer from process to process. However, if workers perform any post-processing on the fetched rows (such as aggregation), the total execution time may turn out to be much shorter.

Cost Estimation

Let's take a look at a simple query that performs aggregation on a large table. The execution plan is parallelized:

[1] backend/access/heap/heapam.c, table_block_parallelscan_startblock_init & table_block_parallelscan_nextpage functions

```
=> EXPLAIN SELECT count(*) FROM bookings;
                           QUERY PLAN
-----------------------------------------------------------------
 Finalize Aggregate  (cost=25442.58..25442.59 rows=1 width=8)
   -> Gather  (cost=25442.36..25442.57 rows=2 width=8)
       Workers Planned: 2
       -> Partial Aggregate
          (cost=24442.36..24442.37 rows=1 width=8)
          -> Parallel Seq Scan on bookings
             (cost=0.00..22243.29 rows=879629 width=0)
(7 rows)
```

All the nodes below Gather belong to the parallel part of the plan. They are executed by each of the workers (two of them are planned here) and possibly by the leader process (unless this functionality is turned off by the *parallel_leader_participation* parameter). The Gather node itself and all the nodes above it make the sequential part of the plan and are executed by the leader process alone.

The Parallel Seq Scan node represents a parallel heap scan. The rows field shows the estimated *average* number of rows to be processed by a *single* process. All in all, the execution must be performed by three processes (one leader and two workers), but the leader process will handle fewer rows: its share gets smaller as the number of workers grows.[1] In this particular case, the factor is 2.4.

```
=> SELECT reltuples::numeric, round(reltuples / 2.4) AS per_process
FROM pg_class WHERE relname = 'bookings';
 reltuples | per_process
-----------+-------------
   2111110 |      879629
(1 row)
```

The Parallel Seq Scan cost is calculated similar to that of a sequential scan. The received value is smaller, as each process handles fewer rows; the I/O part is included in full since the whole table still has to be read, page by page:

```
=> SELECT round((
  relpages        * current_setting('seq_page_cost')::real +
  reltuples / 2.4 * current_setting('cpu_tuple_cost')::real
)::numeric, 2)
FROM pg_class WHERE relname = 'bookings';
```

[1] backend/optimizer/path/costsize.c, get_parallel_divisor function

```
  round
----------
 22243.29
(1 row)
```

Next, the Partial Aggregate node performs aggregation of the fetched data; in this particular case, it counts the number of rows.

The aggregation cost is estimated in the usual manner and is added to the cost estimation of the table scan:

```
=> WITH t(startup_cost)
AS (
  SELECT 22243.29 + round((
    reltuples / 2.4 * current_setting('cpu_operator_cost')::real
  )::numeric, 2)
  FROM pg_class
  WHERE relname = 'bookings'
)
SELECT startup_cost,
  startup_cost + round((
    1 * current_setting('cpu_tuple_cost')::real
  )::numeric, 2) AS total_cost
FROM t;
 startup_cost | total_cost
--------------+------------
     24442.36 |   24442.37
(1 row)
```

The next node (Gather) is executed by the leader process. This node is responsible for launching workers and gathering the data they return.

For the purpose of planning, the cost estimation of starting processes (regardless of their number) is defined by the *parallel_setup_cost* parameter, while the cost of each row transfer between the processes is estimated at *parallel_tuple_cost*.

1000
0.1

In this example, the startup cost (spent on starting the processes) prevails; this value is added to the startup cost of the Partial Aggregate node. The total cost also includes the cost of transferring two rows; this value is added to the total cost of the Partial Aggregate node:[1]

[1] backend/optimizer/path/costsize.c, cost_gather function

```
=> SELECT
  24442.36 + round(
    current_setting('parallel_setup_cost')::numeric,
  2) AS setup_cost,
  24442.37 + round(
    current_setting('parallel_setup_cost')::numeric +
    2 * current_setting('parallel_tuple_cost')::numeric,
  2) AS total_cost;
 setup_cost | total_cost
------------+------------
   25442.36 |   25442.57
(1 row)
```

Last but not least, the Finalize Aggregate node aggregates all the partial results received by the Gather node from the parallel processes.

The final aggregation is estimated just like any other. Its startup cost is based on the cost of aggregating three rows; this value is added to the total cost of Gather (since all the rows are needed to compute the result). The total cost of Finalize Aggregate also includes the cost of returning one row.

```
=> WITH t(startup_cost) AS (
  SELECT 25442.57 + round((
    3 * current_setting('cpu_operator_cost')::real
  )::numeric, 2)
  FROM pg_class WHERE relname = 'bookings'
)
SELECT startup_cost,
  startup_cost + round((
    1 * current_setting('cpu_tuple_cost')::real
  )::numeric, 2) AS total_cost
FROM t;
 startup_cost | total_cost
--------------+------------
     25442.58 |   25442.59
(1 row)
```

Dependencies between cost estimations are determined by whether the node has to accumulate the data before passing the result to its parent node. Aggregation cannot return the result until it gets all the input rows, so its startup cost is based on the *total* cost of the lower node. The Gather node, on the contrary, starts sending rows upstream as soon as they are fetched. Therefore, the startup cost of this operation depends on the startup cost of the lower node, while its total cost is based on the lower node's total cost.

Here is the dependency graph:

```
                        QUERY PLAN
-------------------------------------------------------
 Finalize Aggregate
   (cost=25442.58..25442.59 rows=1 width=8)
   -> Gather
       (cost=25442.36..25442.57 rows=2 width=8)
       Workers Planned: 2
       -> Partial Aggregate
           (cost=24442.36..24442.37 rows=1 width=8)
           -> Parallel Seq Scan on bookings
               (cost=0.00..22243.29 rows=879629 width=0)
(9 rows)
```

18.5 Parallel Execution Limitations

Number of Background Workers

The number of processes is controlled by a hierarchy of three parameters. The maximal number of background workers running concurrently is defined by the *max_worker_processes* value. 8

However, parallel query execution is not the only operation that needs background workers. For example, they also participate in logical replication and can be used by extensions. The number of processes allocated specifically for parallel plan execution is limited to the *max_parallel_workers* value. 8

Out of this number, up to *max_parallel_workers_per_gather* processes can serve one leader. 2

The choice of these parameter values depends on the following factors:

- Hardware capabilities: the system must have free cores dedicated to parallel execution.
- Table sizes: the database must contain large tables.
- A typical load: there must be queries that potentially benefit from parallel execution.

These criteria are typically met by OLAP systems rather than OLTP ones.

The planner will not consider parallel execution at all if the estimated volume of
8MB heap data to be read does not exceed the *min_parallel_table_scan_size* value.

Unless the number of processes for a particular table is explicitly specified in the *parallel_workers* storage parameter, it will be calculated by the following formula:

$$1 + \left\lfloor \log_3 \left(\frac{\text{table size}}{min_parallel_table_scan_size} \right) \right\rfloor$$

It means that each time a table grows three times, PostgreSQL assigns one more parallel worker for its processing. The default settings give us these figures:

table, MB	number of processes
8	1
24	2
72	3
216	4
648	5
1944	6

In any case, the number of parallel workers cannot exceed the limit defined by the *max_parallel_workers_per_gather* parameter.

If we query a small table of 19 MB, only one worker will be planned and launched:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT count(*) FROM flights;


                              QUERY PLAN
----------------------------------------------------------------------
 Finalize Aggregate (actual rows=1 loops=1)
   -> Gather (actual rows=2 loops=1)
       Workers Planned: 1
       Workers Launched: 1
       -> Partial Aggregate (actual rows=1 loops=2)
           -> Parallel Seq Scan on flights (actual rows=107434 lo...
(6 rows)
```

A query on a table of 105 MB gets only two processes because it hits the limit of *max_parallel_workers_per_gather* workers: 2

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT count(*) FROM bookings;
                            QUERY PLAN
-------------------------------------------------------------------
 Finalize Aggregate (actual rows=1 loops=1)
   -> Gather (actual rows=3 loops=1)
       Workers Planned: 2
       Workers Launched: 2
       -> Partial Aggregate (actual rows=1 loops=3)
           -> Parallel Seq Scan on bookings (actual rows=703703 l...
(6 rows)
```

If we remove this limit, we will get the estimated three processes:

```
=> ALTER SYSTEM SET max_parallel_workers_per_gather = 4;
=> SELECT pg_reload_conf();
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT count(*) FROM bookings;
                            QUERY PLAN
-------------------------------------------------------------------
 Finalize Aggregate (actual rows=1 loops=1)
   -> Gather (actual rows=4 loops=1)
       Workers Planned: 3
       Workers Launched: 3
       -> Partial Aggregate (actual rows=1 loops=4)
           -> Parallel Seq Scan on bookings (actual rows=527778 l...
(6 rows)
```

If the number of slots that are free during query execution turns out to be smaller than the planned value, only the available number of workers will be launched.

Let's limit the total number of parallel processes to five and run two queries simultaneously:

```
=> ALTER SYSTEM SET max_parallel_workers = 5;
=> SELECT pg_reload_conf();
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT count(*) FROM bookings;
```

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT count(*) FROM bookings;
                           QUERY PLAN
----------------------------------------------------------------
 Finalize Aggregate (actual rows=1 loops=1)
   -> Gather (actual rows=3 loops=1)
       Workers Planned: 3
       Workers Launched: 2
       -> Partial Aggregate (actual rows=1 loops=3)
           -> Parallel Seq Scan on bookings (actual rows=7037...
(6 rows)
```

```
                           QUERY PLAN
----------------------------------------------------------------
 Finalize Aggregate (actual rows=1 loops=1)
   -> Gather (actual rows=4 loops=1)
       Workers Planned: 3
       Workers Launched: 3
       -> Partial Aggregate (actual rows=1 loops=4)
           -> Parallel Seq Scan on bookings (actual rows=527778 l...
(6 rows)
```

Although three processes were expected in both cases, one of the queries managed to get only two slots.

Let's restore the default settings:

```
=> ALTER SYSTEM RESET ALL;
=> SELECT pg_reload_conf();
```

Non-Parallelizable Queries

Not all queries can be parallelized.[1] In particular, parallel plans cannot be used for the following query types:

- Queries that modify or lock data (UPDATE, DELETE, SELECT FOR UPDATE, and the like).

[1] postgresql.org/docs/14/when-can-parallel-query-be-used.html

This restriction does not apply to subqueries within the following commands:

- CREATE TABLE AS, SELECT INTO, CREATE MATERIALIZED VIEW v. 11
- REFRESH MATERIALIZED VIEW v. 14

However, row insertion is still performed sequentially in all these cases.

- Queries that can be paused. It applies to queries run within cursors, including FOR loops in PL/pgSQL.
- Queries that call PARALLEL UNSAFE functions. By default, these are all user-defined functions and a few standard ones. You can get the full list of unsafe functions by querying the system catalog:

```
SELECT * FROM pg_proc WHERE proparallel = 'u';
```

- Queries within functions if these functions are called from a parallelized query (to avoid recursive growth of the number of workers).

Some of these limitations may be removed in the future versions of PostgreSQL. For example, the ability to parallelize queries at the Serializable isolation level is already there. v. 12

Parallel insertion of rows using such commands as INSERT and COPY is currently under development.[1]

A query may remain unparallelized for several reasons:

- This type of a query does not support parallelization at all.
- Parallel plan usage is forbidden by the server configuration (for example, because of the imposed table size limit).
- A parallel plan is more expensive than a sequential one.

To check whether a query can be parallelized at all, you can temporarily switch on the *force_parallel_mode* parameter. Then the planner will build parallel plans whenever possible: off

[1] commitfest.postgresql.org/32/2844
commitfest.postgresql.org/32/2841
commitfest.postgresql.org/32/2610

```
=> EXPLAIN SELECT * FROM flights;
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=63)
(1 row)

=> SET force_parallel_mode = on;
=> EXPLAIN SELECT * FROM flights;
                             QUERY PLAN
-------------------------------------------------------------------
 Gather  (cost=1000.00..27259.37 rows=214867 width=63)
   Workers Planned: 1
   Single Copy: true
   -> Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=63)
(4 rows)
```

Parallel Restricted Queries

The bigger is the parallel part of the plan, the more performance gains can be potentially achieved. However, certain operations are executed strictly sequentially by the leader process alone,[1] even though they do not interfere with parallelization as such. In other words, they cannot appear in the plan tree below the Gather node.

Non-expandable subqueries. The most obvious example of a non-expandable subquery[2] is scanning a CTE result (represented in the plan by the CTE Scan node):

```
=> EXPLAIN (costs off)
WITH t AS MATERIALIZED (
  SELECT * FROM flights
)
SELECT count(*) FROM t;
         QUERY PLAN
----------------------------
 Aggregate
   CTE t
     -> Seq Scan on flights
   -> CTE Scan on t
(4 rows)
```

[1] postgresql.org/docs/14/parallel-safety.html
[2] backend/optimizer/plan/subselect.c

If a CTE is not materialized, the plan does not contain the CTE Scan node, so this limitation does not apply. v. 12

Note, however, that a CTE itself can be computed in the parallel mode if it turns out to be less expensive:

```
=> EXPLAIN (costs off)
WITH t AS MATERIALIZED (
  SELECT count(*) FROM flights
)
SELECT * FROM t;
                   QUERY PLAN
-------------------------------------------------
 CTE Scan on t
   CTE t
     -> Finalize Aggregate
         -> Gather
             Workers Planned: 1
             -> Partial Aggregate
                 -> Parallel Seq Scan on flights
(7 rows)
```

Another example of a non-expandable subquery is shown under by the SubPlan node in the plan below:

```
=> EXPLAIN (costs off)
SELECT * FROM flights f
WHERE f.scheduled_departure > ( -- SubPlan
  SELECT min(f2.scheduled_departure)
  FROM flights f2
  WHERE f2.aircraft_code = f.aircraft_code
);
                       QUERY PLAN
-------------------------------------------------------
 Seq Scan on flights f
   Filter: (scheduled_departure > (SubPlan 1))
   SubPlan 1
     -> Aggregate
         -> Seq Scan on flights f2
             Filter: (aircraft_code = f.aircraft_code)
(6 rows)
```

The first two rows represent the plan of the main query: the flights table is scanned sequentially, and each of its rows is checked against the provided filter. The filter

condition includes a subquery; the plan of this subquery starts on the third row. So the SubPlan node is executed several times, once for each row fetched by sequential scanning in this case.

The upper Seq Scan node of this plan cannot participate in parallel execution because it relies on the data returned by the SubPlan node.

Last but not least, here is one more non-expandable subquery represented by the InitPlan node:

```
=> EXPLAIN (costs off)
SELECT * FROM flights f
WHERE f.scheduled_departure > ( -- SubPlan
  SELECT min(f2.scheduled_departure)
  FROM flights f2
  WHERE EXISTS ( -- InitPlan
    SELECT *
    FROM ticket_flights tf
    WHERE tf.flight_id = f.flight_id
  )
);
                        QUERY PLAN
-------------------------------------------------------
 Seq Scan on flights f
   Filter: (scheduled_departure > (SubPlan 2))
   SubPlan 2
     -> Finalize Aggregate
         InitPlan 1 (returns $1)
           -> Seq Scan on ticket_flights tf
                Filter: (flight_id = f.flight_id)
         -> Gather
             Workers Planned: 1
             Params Evaluated: $1
             -> Partial Aggregate
                 -> Result
                     One-Time Filter: $1
                     -> Parallel Seq Scan on flights f2
(14 rows)
```

Unlike the SubPlan node, InitPlan is evaluated only once (in this particular example, once per each execution of the SubPlan 2 node).

The parent node of InitPlan cannot participate in parallel execution (but those nodes that receive the result of the InitPlan evaluation can, like in this example).

Temporary tables. Temporary tables do not support parallel scanning, as they can be accessed exclusively by the process that has created them. Their pages are processed in the local buffer cache. Making the local cache accessible to several processes would require a locking mechanism like in the shared cache, which would make its other benefits less prominent.

p. 187
p. 274

```
=> CREATE TEMPORARY TABLE flights_tmp AS SELECT * FROM flights;
=> EXPLAIN (costs off)
SELECT count(*) FROM flights_tmp;
          QUERY PLAN
------------------------------
 Aggregate
   -> Seq Scan on flights_tmp
(2 rows)
```

Parallel restricted functions. Functions defined as PARALLEL RESTRICTED are allowed only in the sequential part of the plan. You can get the list of such functions from the system catalog by running the following query:

```
SELECT * FROM pg_proc WHERE proparallel = 'r';
```

Only label your functions as PARALLEL RESTRICTED (to say nothing of PARALLEL SAFE) if you are fully aware of all the implications and have carefully studied all the imposed restrictions.[1]

[1] postgresql.org/docs/14/parallel-safety#PARALLEL-LABELING.html

19

Index Access Methods

19.1 Indexes and Extensibility

Indexes are database objects that mainly serve the purpose of accelerating data access. These are auxiliary structures: any index can be deleted and recreated based on heap data. In addition to data access speedup, indexes are also used to enforce some integrity constraints.

The PostgreSQL core provides six built-in index access methods (index types):

```
=> SELECT amname FROM pg_am WHERE amtype = 'i';
 amname
--------
 btree
 hash
 gist
 gin
 spgist
 brin
(6 rows)
```

v. 9.6 PostgreSQL's extensibility implies that new access methods can be added without modifying the core. One such extension (the bloom method) is included into the standard set of modules.

Despite all the differences between various index types, all of them eventually
p. 387 match a key (such as a value of an indexed column) against heap tuples that contain this key. Tuples are referred to by six-byte *tuple IDs*, or TIDs. Knowing the key or some information about the key, it is possible to quickly read the tuples that are likely to contain the required data without scanning the whole table.

To ensure that a new access method can be added as an extension, PostgreSQL implements a common *indexing engine*. Its main objective is to retrieve and process TIDs returned by a particular access method:

- read data from the corresponding heap tuples
- check tuple visibility against a particular snapshot *p. 92*
- recheck conditions if their evaluation by the method is indecisive

The indexing engine also participates in execution of plans built at the optimization stage. When assessing various execution paths, the optimizer needs to know the properties of all potentially applicable access methods: can the method return the data in the required order, or do we need a separate sorting stage? is it possible to return several first values right away, or do we have to wait for the whole result set to be fetched? and so on.

It is not only the optimizer that needs to know specifics of the access method. Index creation poses more questions to answer: does the access method support multicolumn indexes? can this index guarantee uniqueness?

The indexing engine allows using a variety of access methods; in order to be supported, an access method must implement a particular interface to declare its features and properties.

Access methods are used to address the following tasks:

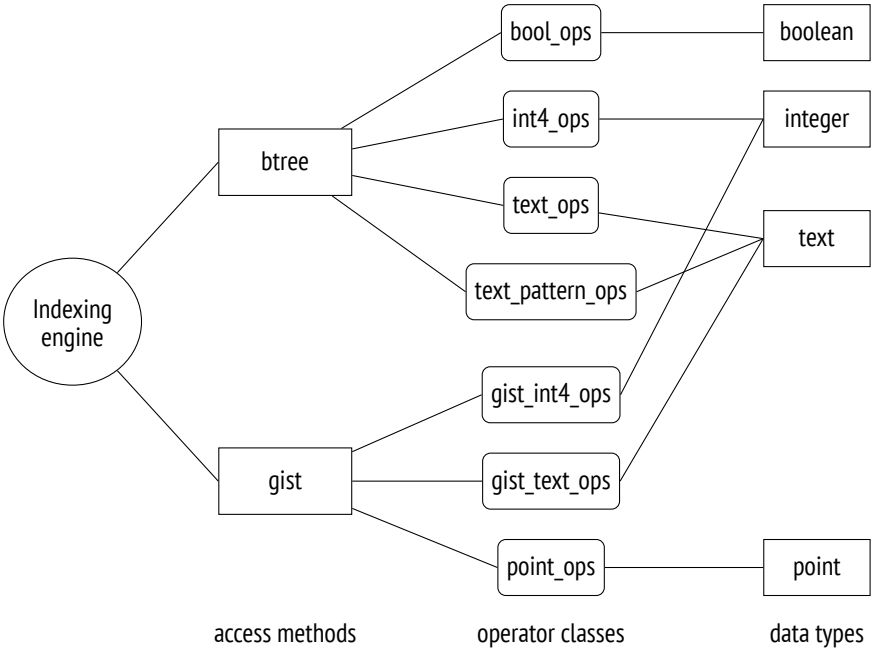
- implement algorithms for building indexes, as well as inserting and deleting index entries
- distribute index entries between pages (to be further handled by the buffer cache manager) *p. 169*
- implement the algorithm of vacuuming *p. 118*
- acquire locks to ensure correct concurrent operation *p. 274*
- generate WAL entries *p. 189*
- search indexed data by the key
- estimate index scan costs

Extensibility also manifests itself as the ability to add new data types, which the access method knows nothing of in advance. Therefore, access methods have to define their own interfaces for plugging in arbitrary data types.

To enable usage of a new data type with a particular access method, you have to implement the corresponding interface—that is, provide operators that can be used with an index, and possibly some auxiliary *support functions*. Such a set of operators and functions is called an *operator class*.

The indexing logic is partially implemented by the access method itself, but some of it is outsourced to operator classes. This distribution is rather arbitrary: while B-trees have all the logic wired into the access method, some other methods may provide only the main framework, leaving all the implementation details at the discretion of particular operator classes. One and the same data type is often supported by several operator classes, and the user can select the one with the most suitable behavior.

Here is a small fraction of the overall picture:



19.2 Operator Classes and Families

Operator Classes

An access method interface[1] is implemented by an *operator class*,[2] which is a set of operators and support functions applied by the access method to a particular data type.

Classes of operators are stored in the pg_opclass table in the system catalog. The following query returns the complete data for the above illustration:

```
=> SELECT amname, opcname, opcintype::regtype
FROM pg_am am
  JOIN pg_opclass opc ON opcmethod = am.oid;
 amname |           opcname            |          opcintype
--------+------------------------------+----------------------------
 btree  | array_ops                    | anyarray
 hash   | array_ops                    | anyarray
 btree  | bit_ops                      | bit
 btree  | bool_ops                     | boolean
 ...
 brin   | pg_lsn_minmax_multi_ops      | pg_lsn
 brin   | pg_lsn_bloom_ops             | pg_lsn
 brin   | box_inclusion_ops            | box
(177 rows)
```

In most cases, we do not have to know anything about operator classes. We simply create an index that uses some operator class by default.

For example, here are B-tree operator classes that support the text type. One of the classes is always marked as the default one:

```
=> SELECT opcname, opcdefault
FROM pg_am am
  JOIN pg_opclass opc ON opcmethod = am.oid
WHERE amname = 'btree'
  AND opcintype = 'text'::regtype;
```

[1] postgresql.org/docs/14/xindex.html
[2] postgresql.org/docs/14/indexes-opclass.html

```
       opcname       | opcdefault
---------------------+------------
 text_ops            | t
 varchar_ops         | f
 text_pattern_ops    | f
 varchar_pattern_ops | f
(4 rows)
```

A typical command for index creation looks as follows:

```
CREATE INDEX ON aircrafts(model, range);
```

But it is just a shorthand notation that expands to the following syntax:

```
CREATE INDEX ON aircrafts
USING btree -- the default access method
(
  model text_ops, -- the default operator class for text
  range int4_ops  -- the default operator class for integer
);
```

If you would like to use an index of a different type or achieve some custom behavior, you have to specify the desired access method or operator class explicitly.

Each operator class defined for a particular access method and data type must contain a set of operators that take parameters of this type and implement the semantics of this access method.

For example, the btree access method defines five mandatory comparison operators. Any btree operator class must contain all the five:

```
=> SELECT opcname, amopstrategy, amopopr::regoperator
FROM pg_am am
  JOIN pg_opfamily opf ON opfmethod = am.oid
  JOIN pg_opclass opc ON opcfamily = opf.oid
  JOIN pg_amop amop ON amopfamily = opcfamily
WHERE amname = 'btree'
  AND opcname IN ('text_ops', 'text_pattern_ops')
  AND amoplefttype = 'text'::regtype
  AND amoprighttype = 'text'::regtype
ORDER BY opcname, amopstrategy;
```

```
     opcname      | amopstrategy |     amopopr
------------------+--------------+-----------------
 text_ops         |            1 | <(text,text)
 text_ops         |            2 | <=(text,text)
 text_ops         |            3 | =(text,text)
 text_ops         |            4 | >=(text,text)
 text_ops         |            5 | >(text,text)
 text_pattern_ops |            1 | ~<~(text,text)
 text_pattern_ops |            2 | ~<=~(text,text)
 text_pattern_ops |            3 | =(text,text)
 text_pattern_ops |            4 | ~>=~(text,text)
 text_pattern_ops |            5 | ~>~(text,text)
(10 rows)
```

The semantics of an operator implied by the access method is reflected by the strategy number shown as amopstrategy.[1] For example, strategy 1 for btree means "less than," 2 denotes "less than or equal to," and so on. Operators themselves can have arbitrary names.

The example above shows two kinds of operators. The difference between regular operators and those with a tilde is that the latter do not take *collation*[2] into account and perform bitwise comparison of strings. Nevertheless, both flavors implement the same logical operations of comparison.

The text_pattern_ops operator class is designed to address the limitation in support of the ~~ operator (which corresponds to the LIKE operator). In a database using any collation other than C, this operator cannot use a regular index on a text field:

```
=> SHOW lc_collate;
 lc_collate
-------------
 en_US.UTF-8
(1 row)
=> CREATE INDEX ON tickets(passenger_name);
=> EXPLAIN (costs off)
SELECT * FROM tickets WHERE passenger_name LIKE 'ELENA%';
```

[1] postgresql.org/docs/14/xindex#XINDEX-STRATEGIES.html
[2] postgresql.org/docs/14/collation.html
postgresql.org/docs/14/indexes-collations.html

```
                    QUERY PLAN
---------------------------------------------
 Seq Scan on tickets
   Filter: (passenger_name ~~ 'ELENA%'::text)
(2 rows)
```

An index with the text_pattern_ops operator class behaves differently:

```
=> CREATE INDEX tickets_passenger_name_pattern_idx
ON tickets(passenger_name text_pattern_ops);
=> EXPLAIN (costs off)
SELECT * FROM tickets WHERE passenger_name LIKE 'ELENA%';
                           QUERY PLAN
---------------------------------------------------------------
 Bitmap Heap Scan on tickets
   Filter: (passenger_name ~~ 'ELENA%'::text)
   -> Bitmap Index Scan on tickets_passenger_name_pattern_idx
       Index Cond: ((passenger_name ~>=~ 'ELENA'::text) AND
       (passenger_name ~<~ 'ELENB'::text))
(5 rows)
```

Note how the filter expression has changed in the Index Cond condition. The search now uses only the template's prefix before %, while false-positive hits are filtered out during a recheck based on the Filter condition. The operator class for the btree access method does not provide an operator for comparing templates, and the only way to apply a B-tree here is to rewrite this condition using comparison operators. The operators of the text_pattern_ops class do not take collation into account, which gives us an opportunity to use an equivalent condition instead.[1]

An index can be used to speed up access by a filter condition if the following two prerequisites are met:

1. the condition is written as "*indexed-column operator expression*" (if the operator has a commuting counterpart specified,[2] the condition can also have the form of "*expression operator indexed-column*")[3]
2. and the *operator* belongs to the operator class specified for the *indexed-column* in the index declaration.

[1] backend/utils/adt/like_support.c
[2] postgresql.org/docs/14/xoper-optimization#id-1.8.3.18.6.html
[3] backend/optimizer/path/indxpath.c, match_clause_to_indexcol function

For example, the following query can use an index:

```
=> EXPLAIN (costs off)
SELECT * FROM tickets WHERE 'ELENA BELOVA' = passenger_name;
                        QUERY PLAN
-------------------------------------------------------
 Index Scan using tickets_passenger_name_idx on tickets
   Index Cond: (passenger_name = 'ELENA BELOVA'::text)
(2 rows)
```

Note the position of arguments in the Index Cond condition: at the execution stage, the indexed field must be on the left. When the arguments are permuted, the operator is replaced by a commuting one; in this particular case, it is the same operator because the equality relation is commutative.

In the next query, it is technically impossible to use a regular index because the column name in the condition is replaced by a function call:

```
=> EXPLAIN (costs off)
SELECT * FROM tickets WHERE initcap(passenger_name) = 'Elena Belova';
                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on tickets
   Filter: (initcap(passenger_name) = 'Elena Belova'::text)
(2 rows)
```

Here you can use an *expression index*,[1] which has an arbitrary expression specified in its declaration instead of a column:

```
=> CREATE INDEX ON tickets( (initcap(passenger_name)) );
=> EXPLAIN (costs off)
SELECT * FROM tickets WHERE initcap(passenger_name) = 'Elena Belova';
                              QUERY PLAN
-------------------------------------------------------------------
 Bitmap Heap Scan on tickets
   Recheck Cond: (initcap(passenger_name) = 'Elena Belova'::text)
   -> Bitmap Index Scan on tickets_initcap_idx
       Index Cond: (initcap(passenger_name) = 'Elena Belova'::text)
(4 rows)
```

An index expression can depend only on heap tuple values and must be affected by neither other data stored in the database nor configuration parameters (such

[1] postgresql.org/docs/14/indexes-expressional.html

as locale settings). In other words, if the expression contains any function calls, these functions must be IMMUTABLE,[1] and they must *observe* this volatility category. Otherwise, an index scan and a heap scan may return different results for the same query.

Apart from regular operators, an operator class can provide *support functions*[2] required by the access method. For example, the btree access method defines five support functions;[3] the first one (which compares two values) is mandatory, while all the rest can be absent:

```
=> SELECT amprocnum, amproc::regproc
FROM pg_am am
  JOIN pg_opfamily opf ON opfmethod = am.oid
  JOIN pg_opclass opc ON opcfamily = opf.oid
  JOIN pg_amproc amproc ON amprocfamily = opcfamily
WHERE amname = 'btree'
  AND opcname = 'text_ops'
  AND amproclefttype = 'text'::regtype
  AND amprocrighttype = 'text'::regtype
ORDER BY amprocnum;
 amprocnum |       amproc
-----------+--------------------
         1 | bttextcmp
         2 | bttextsortsupport
         4 | btvarstrequalimage
(3 rows)
```

Operator Family

Each operator class always belongs to some *operator family*[4] (listed in the system catalog in the pg_opfamily table). A family can comprise several classes that handle similar data types in the same way.

For example, the integer_ops family includes several classes for integral data types that have the same semantics but differ in size:

[1] postgresql.org/docs/14/xfunc-volatility.html
[2] postgresql.org/docs/14/xindex#XINDEX-SUPPORT.html
[3] postgresql.org/docs/14/btree-support-funcs.html
[4] postgresql.org/docs/14/xindex#XINDEX-OPFAMILY.html

```
=> SELECT opcname, opcintype::regtype
FROM pg_am am
  JOIN pg_opfamily opf ON opfmethod = am.oid
  JOIN pg_opclass opc ON opcfamily = opf.oid
WHERE amname = 'btree'
  AND opfname = 'integer_ops';
 opcname  | opcintype
----------+-----------
 int2_ops | smallint
 int4_ops | integer
 int8_ops | bigint
(3 rows)
```

The datetime_ops family comprises operator classes that process dates:

```
=> SELECT opcname, opcintype::regtype
FROM pg_am am
  JOIN pg_opfamily opf ON opfmethod = am.oid
  JOIN pg_opclass opc ON opcfamily = opf.oid
WHERE amname = 'btree'
  AND opfname = 'datetime_ops';
     opcname     |          opcintype
-----------------+----------------------------
 date_ops        | date
 timestamptz_ops | timestamp with time zone
 timestamp_ops   | timestamp without time zone
(3 rows)
```

While each operator class supports a single data type, a family can comprise operator classes for different data types:

```
=> SELECT opcname, amopopr::regoperator
FROM pg_am am
  JOIN pg_opfamily opf ON opfmethod = am.oid
  JOIN pg_opclass opc ON opcfamily = opf.oid
  JOIN pg_amop amop ON amopfamily = opcfamily
WHERE amname = 'btree'
  AND opfname = 'integer_ops'
  AND amoplefttype = 'integer'::regtype
  AND amopstrategy = 1
ORDER BY opcname;
```

```
 opcname  |       amopopr
----------+---------------------
 int2_ops | <(integer,bigint)
 int2_ops | <(integer,smallint)
 int2_ops | <(integer,integer)
 int4_ops | <(integer,bigint)
 int4_ops | <(integer,smallint)
 int4_ops | <(integer,integer)
 int8_ops | <(integer,bigint)
 int8_ops | <(integer,smallint)
 int8_ops | <(integer,integer)
(9 rows)
```

Thanks to such grouping of various operators into a single family, the planner can do without type casting when an index is used for conditions involving values of different types.

19.3 Indexing Engine Interface

v. 9.6 Just like for table access methods, the amhandler column of the pg_am table contains the name of the function that implements the interface:[1]

```
=> SELECT amname, amhandler FROM pg_am WHERE amtype = 'i';
 amname |  amhandler
--------+-------------
 btree  | bthandler
 hash   | hashhandler
 gist   | gisthandler
 gin    | ginhandler
 spgist | spghandler
 brin   | brinhandler
(6 rows)
```

This function fills placeholders in the interface structure[2] with actual values. Some of them are functions responsible for separate tasks related to index access (for example, they can perform an index scan and return heap tuple IDs), while others are index method properties that the indexing engine must be aware of.

[1] postgresql.org/docs/14/indexam.html
[2] include/access/amapi.h

All properties are grouped into three categories:[1]

- access method properties
- properties of a particular index
- column-level properties of an index

The distinction between access method and index-level properties is provided with a view to the future: right now, all the indexes based on a particular access method always have the same properties at these two levels.

Access Method Properties

The following five properties are defined at the access method level (shown for the B-tree method here): v. 11

```
=> SELECT a.amname, p.name, pg_indexam_has_property(a.oid, p.name)
FROM pg_am a, unnest(array[
  'can_order', 'can_unique', 'can_multi_col',
  'can_exclude', 'can_include'
]) p(name)
WHERE a.amname = 'btree';
 amname |     name      | pg_indexam_has_property
--------+---------------+-------------------------
 btree  | can_order     | t
 btree  | can_unique    | t
 btree  | can_multi_col | t
 btree  | can_exclude   | t
 btree  | can_include   | t
(5 rows)
```

Can Order The ability to receive sorted data.[2] This property is currently supported only by B-trees.

To get the results in the required order, you can always scan the table and then sort the fetched data:

[1] backend/utils/adt/amutils.c, indexam_property function
[2] postgresql.org/docs/14/indexes-ordering.html

```
=> EXPLAIN (costs off)
SELECT * FROM seats ORDER BY seat_no;
       QUERY PLAN
------------------------
 Sort
   Sort Key: seat_no
   -> Seq Scan on seats
(3 rows)
```

But if there is an index that supports this property, the data can be returned in the desired order at once:

```
=> EXPLAIN (costs off)
SELECT * FROM seats ORDER BY aircraft_code;
              QUERY PLAN
-------------------------------------
 Index Scan using seats_pkey on seats
(1 row)
```

Can Unique Support for unique and primary key constraints.[1] This property applies only to B-trees.

Each time a unique or primary key constraint is declared, PostgreSQL automatically creates a unique index to support this constraint.

```
=> INSERT INTO bookings(book_ref, book_date, total_amount)
VALUES ('000004', now(), 100.00);
ERROR:  duplicate key value violates unique constraint
"bookings_pkey"
DETAIL:  Key (book_ref)=(000004) already exists.
```

That said, if you simply create a unique index without explicitly declaring an integrity constraint, the effect will seem to be exactly the same: the indexed column will not allow duplicates. So what is the difference?

An integrity constraint defines the property that must never be violated, while an index is just a mechanism to guarantee it. In theory, a constraint could be imposed using other means.

For example, PostgreSQL does not support global indexes for partitioned tables, but nevertheless, you can create a unique constraint on such tables (if it

[1] postgresql.org/docs/14/indexes-unique.html

includes the partition key). In this case, the global uniqueness is ensured by local unique indexes of each partition, as different partitions cannot have the same partition keys.

Can Multi Col The ability to build a *multicolumn index*.[1]

A multicolumn index can speed up search by several conditions imposed on different table columns. For example, the ticket_flights table has a composite primary key, so the corresponding index is built on more than one column:

```
=> \d ticket_flights_pkey
     Index "bookings.ticket_flights_pkey"
  Column   |     Type      | Key? | Definition
-----------+---------------+------+------------
 ticket_no | character(13) | yes  | ticket_no
 flight_id | integer       | yes  | flight_id
primary key, btree, for table "bookings.ticket_flights"
```

A flight search by a ticket number and a flight ID is performed using an index:

```
=> EXPLAIN (costs off)
SELECT * FROM ticket_flights
WHERE ticket_no = '0005432001355'
  AND flight_id = 51618;
                        QUERY PLAN
-------------------------------------------------------
 Index Scan using ticket_flights_pkey on ticket_flights
   Index Cond: ((ticket_no = '0005432001355'::bpchar) AND
   (flight_id = 51618))
(3 rows)
```

As a rule, a multicolumn index can speed up search even if filter conditions involve only some of its columns. In the case of a B-tree, the search will be efficient if the filter condition spans a range of columns that appear first in the index declaration:

```
=> EXPLAIN (costs off)
SELECT *
FROM ticket_flights
WHERE ticket_no = '0005432001355';
```

[1] postgresql.org/docs/14/indexes-multicolumn.html

```
                          QUERY PLAN
-------------------------------------------------------
 Index Scan using ticket_flights_pkey on ticket_flights
   Index Cond: (ticket_no = '0005432001355'::bpchar)
(2 rows)
```

In all other cases (for example, if the condition includes only flights_id), search will be virtually limited to the initial columns (if the query includes the corresponding conditions), while other conditions will only be used to filter out the returned results. Indexes of other types may behave differently though.

Can Exclude Support for EXCLUDE constraints.[1]

An EXCLUDE constraint guarantees that a condition defined by an operator will not be satisfied for any pair of table rows. To impose this constraint, PostgreSQL automatically creates an index; there must be an operator class that contains the operator used in the constraint's condition.

It is the intersection operator && that usually serves this purpose. For instance, you can use it to explicitly declare that a conference room cannot be booked twice for the same time, or that buildings on a map cannot overlap.

With the equality operator, the exclusion constraint takes on the meaning of uniqueness: the table is forbidden to have two rows with the same key values. Nevertheless, it is not the same as a UNIQUE constraint: in particular, the exclusion constraint key cannot be referred to from foreign keys, and neither can it be used in the ON CONFLICT clause.

v. 11 **Can Include** The ability to add non-key columns to an index, which make this index covering.
p. 384

Using this property, you can extend a unique index with additional columns. Such an index still guarantees that all the key column values are unique, while data retrieval from the included columns incurs no heap access:

```
=> CREATE UNIQUE INDEX ON flights(flight_id) INCLUDE (status);
=> EXPLAIN (costs off)
SELECT status FROM flights
WHERE flight_id = 51618;
```

[1] postgresql.org/docs/14/ddl-constraints#DDL-CONSTRAINTS-EXCLUSION.html

```
                          QUERY PLAN
-----------------------------------------------------------------
 Index Only Scan using flights_flight_id_status_idx on flights
   Index Cond: (flight_id = 51618)
(2 rows)
```

Index Properties

Here are the properties related to an index (shown for an existing one):

```
=>  SELECT p.name, pg_index_has_property('seats_pkey', p.name)
FROM unnest(array[
  'clusterable', 'index_scan', 'bitmap_scan', 'backward_scan'
]) p(name);
     name      | pg_index_has_property
---------------+-----------------------
 clusterable   | t
 index_scan    | t
 bitmap_scan   | t
 backward_scan | t
(4 rows)
```

Clusterable The ability to physically move heap tuples in accordance with the order in which their IDs are returned by an index scan. *p. 323*

This property shows whether the CLUSTER command is supported.

Index Scan *Index scan* support. *p. 373*

This property implies that the access method can return TIDs one by one. Strange as it may seem, some indexes do not provide this functionality.

Bitmap Scan *Bitmap scan* support. *p. 385*

This property defines whether the access method can build and return a bitmap of all TIDs at once.

Backward Scan The ability to return results in reverse order as compared to the one specified at index creation.

This property makes sense only if the access method supports index scans.

Column Properties

And finally, let's take a look at the column properties:

```
=> SELECT p.name,
  pg_index_column_has_property('seats_pkey', 1, p.name)
FROM unnest(array[
  'asc', 'desc', 'nulls_first', 'nulls_last', 'orderable',
  'distance_orderable', 'returnable', 'search_array', 'search_nulls'
]) p(name);
        name        | pg_index_column_has_property
--------------------+------------------------------
 asc                | t
 desc               | f
 nulls_first        | f
 nulls_last         | t
 orderable          | t
 distance_orderable | f
 returnable         | t
 search_array       | t
 search_nulls       | t
(9 rows)
```

Asc, Desc, Nulls First, Nulls Last Ordering column values.

These properties define whether column values should be stored in ascending or descending order, and whether NULL values should appear before or after regular values. All these properties are applicable only to B-trees.

p. 312

Orderable The ability to sort column values using the ORDER BY clause.

This property is applicable only to B-trees.

p. 370

Distance Orderable Support for *ordering operators*.[1]

Unlike regular indexing operators that return logical values, ordering operators return a real number that denotes the "distance" from one argument to another. Indexes support such operators specified in the ORDER BY clause of a query.

For example, the ordering operator <-> can find the airports located at the shortest distance to the specified point:

[1] postgresql.org/docs/14/xindex#XINDEX-ORDERING-OPS.html

```
=> CREATE INDEX ON airports_data USING gist(coordinates);
=> EXPLAIN (costs off)
SELECT * FROM airports
ORDER BY coordinates <-> point (43.578,57.593)
LIMIT 3;
                          QUERY PLAN
-----------------------------------------------------------------
 Limit
   -> Index Scan using airports_data_coordinates_idx on airpo...
       Order By: (coordinates <-> '(43.578,57.593)'::point)
(3 rows)
```

Returnable The ability to return data without accessing the table (*index-only scan support*). *p. 381*

This property defines whether an index structure allows retrieving indexed values. It is not always possible: for example, some indexes may store hash codes rather than actual values. In this case, the Can Include property will not be available either.

Search Array Support for searching several elements in an array.

An explicit use of arrays is not the only case when it might be necessary. For example, the planner transforms the in (list) expression into an array scan:

```
=> EXPLAIN (costs off)
SELECT * FROM bookings
WHERE book_ref IN ('C7C821', 'A5D060', 'DDE1BB');
                 QUERY PLAN
-------------------------------------------
 Index Scan using bookings_pkey on bookings
   Index Cond: (book_ref = ANY
   ('{C7C821,A5D060,DDE1BB}'::bpchar[]))
(3 rows)
```

If the index method does not support such operators, the executor may have to perform several iterations to find particular values (which can make the index scan less efficient).

Search Nulls Search for is null and is not null conditions.

Should we index null values? On the one hand, it allows us to perform index scans for conditions like is [not] null, as well as use the index as a covering

one if no filter conditions are provided (in this case, the index has to return the data of all the heap tuples, including those that contain NULL values). But on the other hand, skipping NULL values can reduce the index size.

The decision remains at the discretion of access method developers, but more often than not NULL values do get indexed.

If you do not need NULL values in an index, you can exclude them by building a *partial index*[1] that covers only those rows that are required. For example:

```
=> CREATE INDEX ON flights(actual_arrival)
WHERE actual_arrival IS NOT NULL;
=> EXPLAIN (costs off)
SELECT * FROM flights
WHERE actual_arrival = '2017-06-13 10:33:00+03';
                           QUERY PLAN
-----------------------------------------------------------------
 Index Scan using flights_actual_arrival_idx on flights
   Index Cond: (actual_arrival = '2017-06-13 10:33:00+03'::ti...
(2 rows)
```

A partial index is smaller than the full one, and it gets updated only if the modified row is indexed, which can sometimes lead to tangible performance gains. Obviously, apart from NULL checks, the WHERE clause can provide any condition (that can be used with immutable functions).

The ability to build partial indexes is provided by the indexing engine, so it does not depend on the access method.

Naturally, the interface includes only those properties of index methods that must be known in advance for a correct decision to be taken. For example, it does not list any properties that enable such features as support for predicate locks or non-blocking index creation (CONCURRENTLY). Such properties are defined in the code of the functions that implement the interface.

[1] postgresql.org/docs/14/indexes-partial.html

20

Index Scans

20.1 Regular Index Scans

There are two basic ways of accessing TIDs provided by an index. The first one is to perform an *index scan*. Most of the index access methods (but not all of them) have the INDEX SCAN property to support this operation. *p. 369*

Index scans are represented in the plan by the Index Scan[1] node:

```
=> EXPLAIN SELECT * FROM bookings
WHERE book_ref = '9AC0C6' AND total_amount = 48500.00;
                 QUERY PLAN
---------------------------------------------
 Index Scan using bookings_pkey on bookings
   (cost=0.43..8.45 rows=1 width=21)
   Index Cond: (book_ref = '9AC0C6'::bpchar)
   Filter: (total_amount = 48500.00)
(4 rows)
```

During an index scan, the access method returns TIDs one by one.[2] Upon receiving a TID, the indexing engine accesses the heap page this TID refers to, gets the corresponding tuple, and, if the visibility rules are met, returns the requested set of fields of this tuple. This process continues until the access method runs out of TIDs that matches the query.

The Index Cond line includes only those filter conditions that can be checked using an index. Other conditions that have to be rechecked against the heap are listed separately in the Filter line.

[1] backend/executor/nodeIndexscan.c
[2] backend/access/index/indexam.c, index_getnext_tid function

As this example shows, both index and heap access operations are handled by a common Index Scan node rather by two different ones. But there is also a separate Tid Scan node,[1] which fetches tuples from the heap if their IDs are known in advance:

```
=> EXPLAIN SELECT * FROM bookings WHERE ctid = '(0,1)'::tid;
                        QUERY PLAN
-------------------------------------------------------
 Tid Scan on bookings  (cost=0.00..4.01 rows=1 width=21)
   TID Cond: (ctid = '(0,1)'::tid)
(2 rows)
```

Cost Estimation

Cost estimation of an index scan comprises the estimated costs of index access operations and heap page reads.

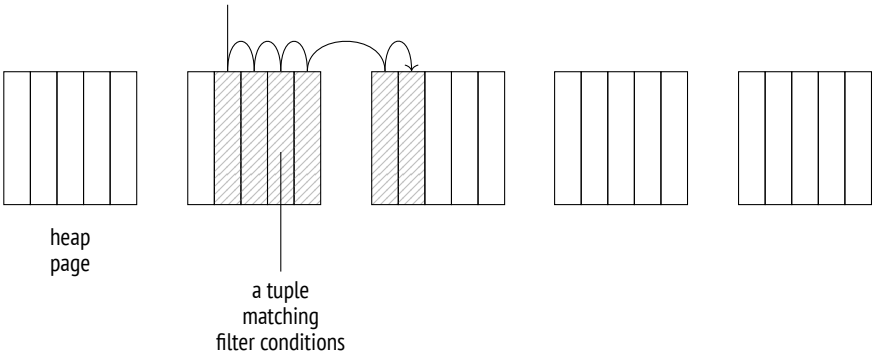
Obviously, the index-related part of the estimation fully depends on the particular access method. For B-trees, the cost is mostly incurred by fetching index pages and processing their entries. The number of pages and rows to be read can be
p. 298 determined by the total volume of data and the selectivity of the applied filters. Index pages are accessed at random (pages that follow each other in the logical structure are physically scattered on disk). The estimation is further increased by CPU resources spent on getting from the root to the leaf node and computing all the required expressions.[2]

The heap-related part of the estimation includes the cost of heap page access and the CPU time required to process all the fetched tuples. It is important to note that I/O estimation depends on both the index scan selectivity and the *correlation* between the physical order of tuples on disk and the order in which the access method returns their IDs.

[1] backend/executor/nodeTidscan.c
[2] backend/utils/adt/selfuncs.c, btcostestimate function
postgresql.org/docs/14/index-cost-estimation.html

Good Scenario: High Correlation

If the physical order of tuples on disk has a perfect correlation with the logical order of TIDs in the index, each page will be accessed *only once*: the Index Scan node will *sequentially* go from one page to another, reading the tuples one by one.



PostgreSQL collects statistics on correlation: p. 323

```
=> SELECT attname, correlation
FROM pg_stats WHERE tablename = 'bookings'
ORDER BY abs(correlation) DESC;
   attname    | correlation
--------------+--------------
 book_ref     |            1
 total_amount | 0.0026738467
 book_date    |  8.02188e-05
(3 rows)
```

The correlation is high if the corresponding absolute value is close to one (like in the case of book_ref); values that are close to zero are a sign of chaotic data distribution.

> In this particular case, high correlation in the book_ref column is of course due to the fact that the data has been loaded into the table in ascending order based on this column, and there have been no updates yet. We would see the same picture if we executed the CLUSTER command for the index created on this column.

However, the perfect correlation does not guarantee that all queries will be returning results in ascending order of book_ref values. First of all, any row update moves the resulting tuple to the end of the table. Second, the plan that relies on an index scan based on some other column returns the results in a different order. And even a sequential scan may not start at the beginning of the table. So if you need a particular order, you should explicitly define it in the ORDER BY clause.

p. 179

Here is an example of an index scan that processes a large number of rows:

```
=> EXPLAIN SELECT * FROM bookings WHERE book_ref < '100000';
                  QUERY PLAN
---------------------------------------------
 Index Scan using bookings_pkey on bookings
   (cost=0.43..4638.91 rows=132999 width=21)
   Index Cond: (book_ref < '100000'::bpchar)
(3 rows)
```

The condition's selectivity is estimated as follows:

```
=> SELECT round(132999::numeric/reltuples::numeric, 4)
FROM pg_class WHERE relname = 'bookings';
 round
--------
 0.0630
(1 row)
```

p. 318

This value is close to $\frac{1}{16}$, which we could have guessed knowing that book_ref values range from 000000 to FFFFFF.

For B-trees, the index-related part of the I/O cost estimation includes the cost of reading all the required pages. Index entries that satisfy any condition supported by B-trees are stored in pages bound into an ordered list, so the number of index pages to be read is estimated at the index size multiplied by the selectivity. But since these pages are not physically ordered, reading happens in a *random* fashion.

CPU resources are spent on processing all the index entries that are read (the cost of processing a single entry is estimated at the *cpu_index_tuple_cost* value) and computing the condition for each of these entries (in this case, the condition contains a single operator; its cost is estimated at the *cpu_operator_cost* value).

0.005

0.0025

Table access is regarded as *sequential* reading of the required number of pages. In the case of a perfect correlation, heap tuples will follow each other on disk, so the number of pages is estimated at the size of the table multiplied by the selectivity.

The I/O cost is further extended by the expenses incurred by tuple processing; they are estimated at the *cpu_tuple_cost* value per tuple. 0.01

```
=> WITH costs(idx_cost, tbl_cost) AS (
  SELECT
    (
      SELECT round(
        current_setting('random_page_cost')::real * pages +
        current_setting('cpu_index_tuple_cost')::real * tuples +
        current_setting('cpu_operator_cost')::real * tuples
      )
      FROM (
        SELECT relpages * 0.0630 AS pages, reltuples * 0.0630 AS tuples
        FROM pg_class WHERE relname = 'bookings_pkey'
      ) c
    ),
    (
      SELECT round(
        current_setting('seq_page_cost')::real * pages +
        current_setting('cpu_tuple_cost')::real * tuples
      )
      FROM (
        SELECT relpages * 0.0630 AS pages, reltuples * 0.0630 AS tuples
        FROM pg_class WHERE relname = 'bookings'
      ) c
    )
)
SELECT idx_cost, tbl_cost, idx_cost + tbl_cost AS total
FROM costs;
 idx_cost | tbl_cost | total
----------+----------+-------
     2457 |     2177 |  4634
(1 row)
```

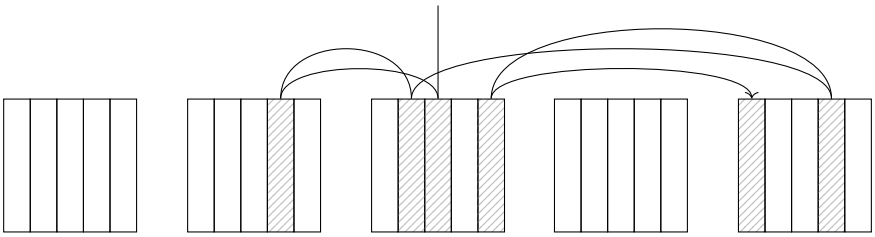
These calculations illustrate the logic behind the cost estimation, so the result is aligned with the estimation provided by the planner, even if it is approximated. Getting the exact value would require taking other details into account, which we are not going to discuss here.

Bad Scenario: Low Correlation

Everything changes if the correlation is low. Let's create an index on the book_date column, which has almost zero correlation with this index, and then take a look at the query that selects almost the same fraction of rows as in the previous example. Index access turns out to be so expensive that the planner chooses it only if all the other alternatives are explicitly forbidden:

```
=> CREATE INDEX ON bookings(book_date);
=> SET enable_seqscan = off;
=> SET enable_bitmapscan = off;
=> EXPLAIN SELECT * FROM bookings
WHERE book_date < '2016-08-23 12:00:00+03';
                              QUERY PLAN
----------------------------------------------------------------------
 Index Scan using bookings_book_date_idx on bookings
   (cost=0.43..56957.48 rows=132403 width=21)
   Index Cond: (book_date < '2016-08-23 12:00:00+03'::timestamp w...
(3 rows)
```

The thing is that low correlation increases the chances of the next tuple returned by the access method to be located in a different page. Therefore, the Index Scan node has to hop between pages instead of reading them sequentially; in the worst-case scenario, the number of page accesses can reach the number of fetched tuples.



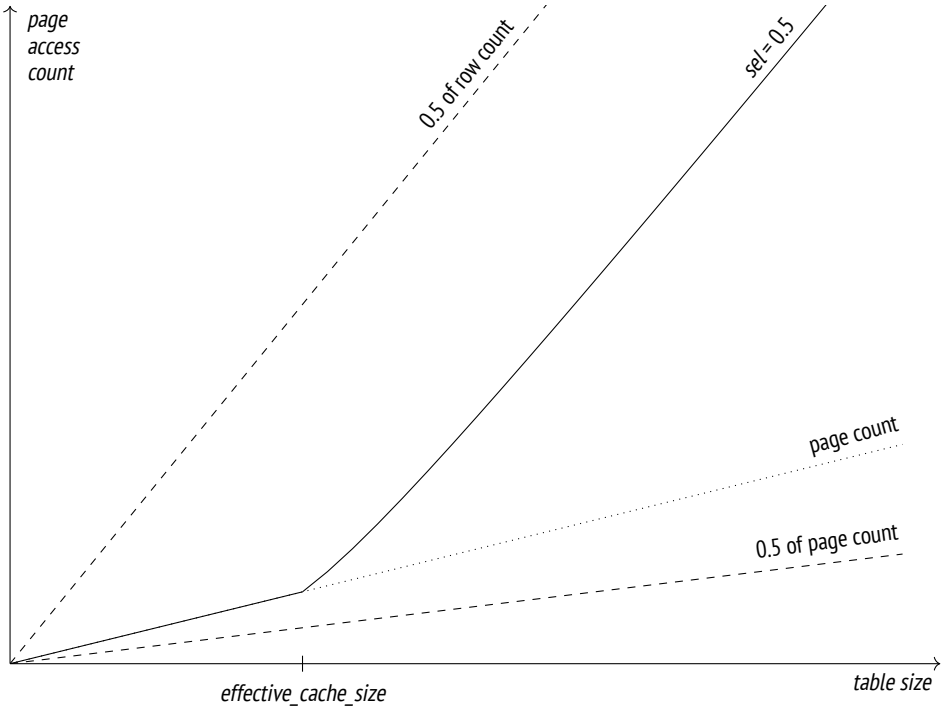
However, we cannot simply replace *seq_page_cost* with *random_page_cost* and relpages with reltuples in the good-scenario calculations. The cost that we see in the plan is much lower than the value we would have estimated this way:

```
=> WITH costs(idx_cost, tbl_cost) AS (
  SELECT
    ( SELECT round(
        current_setting('random_page_cost')::real * pages +
        current_setting('cpu_index_tuple_cost')::real * tuples +
        current_setting('cpu_operator_cost')::real * tuples
      )
      FROM (
        SELECT relpages * 0.0630 AS pages, reltuples * 0.0630 AS tuples
        FROM pg_class WHERE relname = 'bookings_pkey'
      ) c
    ),
    ( SELECT round(
        current_setting('random_page_cost')::real * tuples +
        current_setting('cpu_tuple_cost')::real * tuples
      )
      FROM (
        SELECT relpages * 0.0630 AS pages, reltuples * 0.0630 AS tuples
        FROM pg_class WHERE relname = 'bookings'
      ) c
    )
)
SELECT idx_cost, tbl_cost, idx_cost + tbl_cost AS total FROM costs;
 idx_cost | tbl_cost | total
----------+----------+--------
     2457 |   533330 | 535787
(1 row)
```

The reason is that the model takes caching into account. Frequently used pages are kept in the buffer cache (and in the OS cache), so the bigger the cache size, the more chances to find the required page in it, thus avoiding an extra disk access operation. For planning purposes, the cache size is defined by the *effective_cache_size* parameter. The smaller its value, the more pages are expected to be read. 4GB

The graph that follows shows the dependency between the estimation of the number of pages to be read and the table size (for the selectivity of $\frac{1}{2}$ and the page containing 10 rows).[1] The dashed lines show the access count in the best scenario possible (half of the page count if the correlation is perfect) and in the worst scenario (half of the row count if there is zero correlation and no cache).

[1] backend/optimizer/path/costsize.c, index_pages_fetched function



It is assumed that the *effective_cache_size* value indicates the total volume of memory that can be used for caching (including both the PostgreSQL buffer cache and OS cache). But since this parameter is used solely for estimation purposes and does not affect memory allocation itself, you do not have to take actual figures into account when changing this setting.

If you reduce *effective_cache_size* to the minimum, the plan estimation will be close to the low-end value shown above for the no-caching case:

```
=> SET effective_cache_size = '8kB';
=> EXPLAIN SELECT * FROM bookings
WHERE book_date < '2016-08-23 12:00:00+03';
                              QUERY PLAN
----------------------------------------------------------------------
 Index Scan using bookings_book_date_idx on bookings
   (cost=0.43..532745.48 rows=132403 width=21)
   Index Cond: (book_date < '2016-08-23 12:00:00+03'::timestamp w...
(3 rows)
```

```
=> RESET effective_cache_size;
=> RESET enable_seqscan;
=> RESET enable_bitmapscan;
```

The planner calculates the table I/O cost for both worst-case and best-case scenarios and then takes an intermediate value based on the actual correlation.[1]

Thus, an index scan can be a good choice if only a fraction of rows has to be read. If heap tuples are correlated with the order in which the access method returns their IDs, this fraction can be quite substantial. However, if the correlation is low, index scanning becomes much less attractive for queries with low selectivity.

20.2 Index-Only Scans

If an index contains all the heap data required by the query, it is called a *covering index* for this particular query. If such an index is available, extra table access can be avoided: instead of TIDs, the access method can return the actual data directly. Such a type of an index scan is called an *index-only scan*.[2] It can be used by those access methods that support the RETURNABLE property. *p. 371*

In the plan, this operation is represented by the Index Only Scan[3] node:

```
=> EXPLAIN SELECT book_ref FROM bookings WHERE book_ref < '100000';
                    QUERY PLAN
-----------------------------------------------
 Index Only Scan using bookings_pkey on bookings
   (cost=0.43..3791.91 rows=132999 width=7)
   Index Cond: (book_ref < '100000'::bpchar)
(3 rows)
```

The name suggests that this node never has to access the heap, but it is not so. In PostgreSQL, indexes contain no information on tuple visibility, so the access method returns the data of *all* the heap tuples that satisfy the filter condition, even *p. 84*

[1] backend/optimizer/path/costsize.c, cost_index function
[2] postgresql.org/docs/14/indexes-index-only-scans.html
[3] backend/executor/nodeIndexonlyscan.c

if the current transaction cannot see them. Their visibility is then checked by the indexing engine.

However, if this method had to access the table to check visibility of each tuple, it would not be any different from a regular index scan. Instead, it employs the *visibility map* provided for tables, in which the vacuum process marks the pages that contain only all-visible tuples (that is, those tuples that are accessible to all transactions, regardless of the snapshot used). If the TID returned by the index access method belongs to such a page, there is no need to check its visibility.

p. 29

The cost estimation of an index-only scan depends on the fraction of all-visible pages in the heap. PostgreSQL collects such statistics:

```
=> SELECT relpages, relallvisible
FROM pg_class WHERE relname = 'bookings';
 relpages | relallvisible
----------+---------------
    13447 |         13446
(1 row)
```

The cost estimation of an index-only scan differs from that of a regular index scan: its I/O cost related to table access is taken in proportion to the fraction of pages that do not appear in the visibility map. (The cost estimation of tuple processing is the same.)

Since in this particular example all pages contain only all-visible tuples, the cost of heap I/O is in fact excluded from the cost estimation:

```
=> WITH costs(idx_cost, tbl_cost) AS (
  SELECT
    (
      SELECT round(
        current_setting('random_page_cost')::real * pages +
        current_setting('cpu_index_tuple_cost')::real * tuples +
        current_setting('cpu_operator_cost')::real * tuples
      )
      FROM (
        SELECT relpages * 0.0630 AS pages,
              reltuples * 0.0630 AS tuples
        FROM pg_class WHERE relname = 'bookings_pkey'
      ) c
    ) AS idx_cost,
```

```
    (
      SELECT round(
        (1 - frac_visible) * -- fraction of non-all-visible pages
        current_setting('seq_page_cost')::real * pages +
        current_setting('cpu_tuple_cost')::real * tuples
      )
      FROM (
        SELECT relpages * 0.0630 AS pages,
              reltuples * 0.0630 AS tuples,
          relallvisible::real/relpages::real AS frac_visible
        FROM pg_class WHERE relname = 'bookings'
      ) c
    ) AS tbl_cost
)
SELECT idx_cost, tbl_cost, idx_cost + tbl_cost AS total
FROM costs;
 idx_cost | tbl_cost | total
----------+----------+-------
     2457 |     1330 |  3787
(1 row)
```

Any unvacuumed changes that have not disappeared behind the database horizon yet increase the estimated cost of the plan (and, consequently, make this plan less attractive to the optimizer). The EXPLAIN ANALYZE command can show the actual heap access count. p. 101

In a newly created table, PostgreSQL has to check visibility of all the tuples:

```
=> CREATE TEMP TABLE bookings_tmp
WITH (autovacuum_enabled = off) AS
  SELECT * FROM bookings
  ORDER BY book_ref;
=> ALTER TABLE bookings_tmp ADD PRIMARY KEY(book_ref);
=> ANALYZE bookings_tmp;
=> EXPLAIN (analyze, timing off, summary off)
SELECT book_ref FROM bookings_tmp WHERE book_ref < '100000';
                              QUERY PLAN
----------------------------------------------------------------------
 Index Only Scan using bookings_tmp_pkey on bookings_tmp
   (cost=0.43..4638.91 rows=132999 width=7) (actual rows=132109 l...
   Index Cond: (book_ref < '100000'::bpchar)
   Heap Fetches: 132109
(4 rows)
```

But once the table has been vacuumed, such a check becomes redundant and is not performed as long as all the pages remain all-visible.

```
=> VACUUM bookings_tmp;
=> EXPLAIN (analyze, timing off, summary off)
SELECT book_ref FROM bookings_tmp WHERE book_ref < '100000';
                              QUERY PLAN
----------------------------------------------------------------------
 Index Only Scan using bookings_tmp_pkey on bookings_tmp
   (cost=0.43..3787.91 rows=132999 width=7) (actual rows=132109 l...
   Index Cond: (book_ref < '100000'::bpchar)
   Heap Fetches: 0
(4 rows)
```

Indexes with the Include Clause

It is not always possible to extend an index with all the columns required by a query:

- Adding a new column to a unique index would compromise the uniqueness of the original key columns.
- Index access methods may not provide an operator class for the data type of the column to be added.

v. 11 In this case, you can still include columns into an index without making them a part of the index key. It will of course be impossible to perform an index scan based on the included columns, but if a query references these columns, the index will function as a covering one.

The following example shows how to replace an automatically created primary key index by another index with an included column:

```
=> CREATE UNIQUE INDEX ON bookings(book_ref) INCLUDE (book_date);

=> BEGIN;
=> ALTER TABLE bookings
  DROP CONSTRAINT bookings_pkey CASCADE;
```

```
NOTICE:  drop cascades to constraint tickets_book_ref_fkey on table
tickets
ALTER TABLE

=> ALTER TABLE bookings ADD CONSTRAINT bookings_pkey PRIMARY KEY
  USING INDEX bookings_book_ref_book_date_idx; -- a new index
NOTICE:  ALTER TABLE / ADD CONSTRAINT USING INDEX will rename index
"bookings_book_ref_book_date_idx" to "bookings_pkey"
ALTER TABLE
=> ALTER TABLE tickets
  ADD FOREIGN KEY (book_ref) REFERENCES bookings(book_ref);
=> COMMIT;

=> EXPLAIN SELECT book_ref, book_date
FROM bookings WHERE book_ref < '100000';
                             QUERY PLAN
--------------------------------------------------------------------
 Index Only Scan using bookings_pkey on bookings  (cost=0.43..437...
   Index Cond: (book_ref < '100000'::bpchar)
(2 rows)
```

Such indexes are often called *covering*, but it is not quite correct. An index is considered covering if the set of its columns *covers* all the columns required by a particular query. It does not matter whether it involves any columns added by the INCLUDE clause, or only key columns are being used. Moreover, one and the same index can be covering for one query but not for the other.

20.3 Bitmap Scans

The efficiency of an index scan is limited: as the correlation decreases, the number of accesses to heap pages rises, and scanning becomes random rather than sequential. To overcome this limitation, PostgreSQL can fetch *all* the TIDs before accessing the table and sort them in ascending order based on their page numbers.[1] This is exactly how *bitmap scanning* works, which is yet another common approach to processing TIDs. It can be used by those access methods that support the BITMAP SCAN property. *p. 369*

[1] backend/access/index/indexam.c, index_getbitmap function

Unlike a regular index scan, this operation is represented in the query plan by two nodes:

```
=> CREATE INDEX ON bookings(total_amount);
=> EXPLAIN
SELECT * FROM bookings WHERE total_amount = 48500.00;
                            QUERY PLAN
-------------------------------------------------------------------
 Bitmap Heap Scan on bookings  (cost=54.63..7040.42 rows=2865 wid...
   Recheck Cond: (total_amount = 48500.00)
   -> Bitmap Index Scan on bookings_total_amount_idx
       (cost=0.00..53.92 rows=2865 width=0)
       Index Cond: (total_amount = 48500.00)
(5 rows)
```

The Bitmap Index Scan[1] node gets the *bitmap* of all TIDs[2] from the access method.

The bitmap consists of separate segments, each corresponding to a single heap page. All these segments have the same size, which is enough for all the page tuples, no matter how many of them are present. This number is limited because a tuple header is quite large; a standard-size page can accommodate 256 tuples at the most, which fit 32 bytes.[3]

Then the Bitmap Heap Scan[4] traverses the bitmap segment by segment, reads the corresponding pages, and checks all their tuples that are marked all-visible. Thus, pages are read in ascending order based on their numbers, and each of them is read exactly once.

That said, this process is not the same as sequential scanning since the accessed pages rarely follow each other. Regular prefetching performed by the operating system does not help in this case, so the Bitmap Heap Scan node implements its
1 own prefetching by asynchronously reading *effective_io_concurrency* pages—and it is the only node that does it. This mechanism relies on the posix_fadvise function implemented by some operating systems. If your system supports this function, it makes sense to configure the *effective_io_concurrency* parameter at the tablespace level in accordance with the hardware capabilities.

[1] backend/executor/nodeBitmapIndexscan.c
[2] backend/access/index/indexam.c, index_getbitmap function
[3] backend/nodes/tidbitmap.c
[4] backend/executor/nodeBitmapHeapscan.c

Asynchronous prefetching is also used by some other internal processes:

- for index pages when heap rows are being deleted[1] v. 13
- for heap pages during analysis (ANALYZE)[2] v. 14

The prefetch depth is defined by the *maintenance_io_concurrency*. 10

Bitmap Accuracy

The more pages contain the tuples that satisfy the filter condition of the query, the bigger is the bitmap. It is built in the local memory of the backend, and its size is limited by the *work_mem* parameter. Once the maximum allowed size is reached, 4MB some bitmap segments become lossy: each bit of a lossy segment corresponds to a whole page, while the segment itself comprises a range of pages.[3] As a result, the size of the bitmap becomes smaller at the expense of its accuracy.

The EXPLAIN ANALYZE command shows the accuracy of the built bitmap:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM bookings WHERE total_amount > 150000.00;
                              QUERY PLAN
----------------------------------------------------------------------
 Bitmap Heap Scan on bookings (actual rows=242691 loops=1)
   Recheck Cond: (total_amount > 150000.00)
   Heap Blocks: exact=13447
   -> Bitmap Index Scan on bookings_total_amount_idx (actual rows...
       Index Cond: (total_amount > 150000.00)
(5 rows)
```

Here we have enough memory for an exact bitmap.

If we decrease the *work_mem* value, some of the bitmap segments become lossy:

```
=> SET work_mem = '512kB';
```

[1] backend/access/heap/heapam.c, index_delete_prefetch_buffer function
[2] backend/commands/analyze.c, acquire_sample_rows function
[3] backend/nodes/tidbitmap.c, tbm_lossify function

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM bookings WHERE total_amount > 150000.00;
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on bookings (actual rows=242691 loops=1)
   Recheck Cond: (total_amount > 150000.00)
   Rows Removed by Index Recheck: 1145721
   Heap Blocks: exact=5178 lossy=8269
   -> Bitmap Index Scan on bookings_total_amount_idx (actual rows...
       Index Cond: (total_amount > 150000.00)
(6 rows)
=> RESET work_mem;
```

When reading a heap page that corresponds to a lossy bitmap segment, PostgreSQL has to recheck the filter condition for each tuple in the page. The condition to be rechecked is always displayed in the plan as Recheck Cond, even if this recheck is not performed. The number of tuples filtered out during a recheck is displayed separately (as Rows Removed by Index Recheck).

> If the size of the result set is too big, the bitmap may not fit the *work_mem* memory chunk, even if all its segments are lossy. Then this limit is ignored, and the bitmap takes as much space as required. PostgreSQL neither further reduces the bitmap accuracy nor flushes any of its segments to disk.

Operations on Bitmaps

If the query applies conditions to several table columns that have separate indexes created on them, a bitmap scan can use several indexes together.[1] All these indexes have their own bitmaps built on the fly; the bitmaps are then combined together bit by bit, using either logical conjunction (if the expressions are connected by AND) or logical disjunction (if the expressions are connected by OR). For example:

```
=> EXPLAIN (costs off)
SELECT * FROM bookings
WHERE book_date < '2016-08-28'
  AND total_amount > 250000;
```

[1] postgresql.org/docs/14/indexes-ordering.html

```
                              QUERY PLAN
----------------------------------------------------------------------
 Bitmap Heap Scan on bookings
   Recheck Cond: ((total_amount > '250000'::numeric) AND (book_da...
   -> BitmapAnd
       -> Bitmap Index Scan on bookings_total_amount_idx
           Index Cond: (total_amount > '250000'::numeric)
       -> Bitmap Index Scan on bookings_book_date_idx
           Index Cond: (book_date < '2016-08-28 00:00:00+03'::tim...
(7 rows)
```

Here the BitmapAnd node combines two bitmaps using the bitwise AND operation.

As two bitmaps are being merged into one,[1] exact segments remain exact when merged together (if the new bitmap fits the *work_mem* memory chunk), but if any segment in a pair is lossy, the resulting segment will be lossy too.

Cost Estimation

Let's take a look at the query that uses a bitmap scan:

```
=> EXPLAIN
SELECT * FROM bookings WHERE total_amount = 28000.00;
                              QUERY PLAN
----------------------------------------------------------------------
 Bitmap Heap Scan on bookings  (cost=599.48..14444.96 rows=31878 ...
   Recheck Cond: (total_amount = 28000.00)
   -> Bitmap Index Scan on bookings_total_amount_idx
       (cost=0.00..591.51 rows=31878 width=0)
       Index Cond: (total_amount = 28000.00)
(5 rows)
```

The approximate selectivity of the condition used by the planner equals

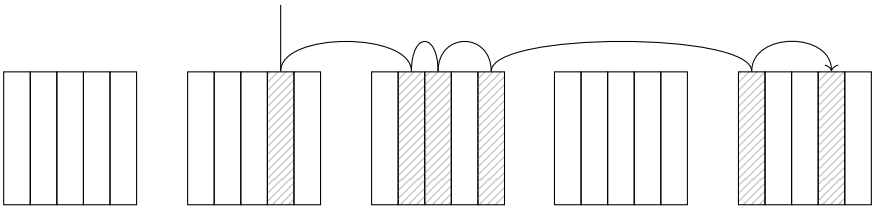
```
=> SELECT round(31878::numeric/reltuples::numeric, 4)
FROM pg_class WHERE relname = 'bookings';
 round
--------
 0.0151
(1 row)
```

[1] backend/nodes/tidbitmap.c, tbm_union & tbm_intersect functions

The total cost of the Bitmap Index Scan node is estimated in the same way as the cost of a regular index scan that does not take heap access into account:

```
=> SELECT round(
  current_setting('random_page_cost')::real * pages +
  current_setting('cpu_index_tuple_cost')::real * tuples +
  current_setting('cpu_operator_cost')::real * tuples
)
FROM (
  SELECT relpages * 0.0151 AS pages, reltuples * 0.0151 AS tuples
  FROM pg_class WHERE relname = 'bookings_total_amount_idx'
) c;
 round
-------
   589
(1 row)
```

The I/O cost estimation for the Bitmap Heap Scan node differs from that for a perfect-correlation case of a regular index scan. A bitmap allows reading heap pages in ascending order based on their numbers, without getting back to one and the same page, but the tuples that satisfy the filter condition do not follow each other anymore. Instead of reading a strictly sequential page range that is quite compact, PostgreSQL is likely to access far more pages.



The number of pages to be read is estimated by the following formula:[1]

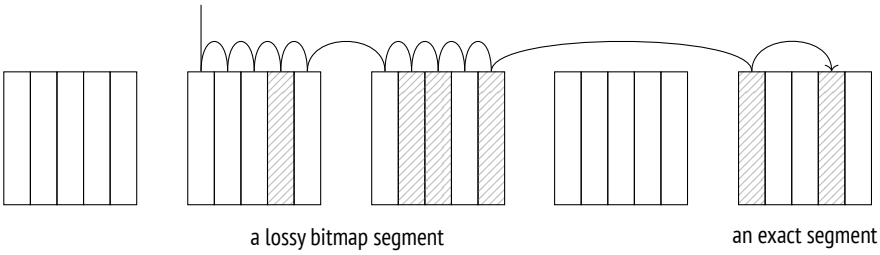
$$\min\left(\frac{2\ \text{relpages}\cdot\text{reltuples}\cdot \mathit{sel}}{2\ \text{relpages}+\text{reltuples}\cdot \mathit{sel}},\ \text{relpages}\right)$$

The estimated cost of reading a single page falls between *seq_page_cost* and *random_page_cost*, depending on the ratio of the fraction of fetched pages to the total number of pages in the table:

[1] backend/optimizer/path/costsize.c, compute_bitmap_pages function

```
=> WITH t AS (
  SELECT relpages,
    least(
      (2 * relpages * reltuples * 0.0151) /
      (2 * relpages + reltuples * 0.0151),
      relpages
    ) AS pages_fetched,
    round(reltuples * 0.0151) AS tuples_fetched,
    current_setting('random_page_cost')::real AS rnd_cost,
    current_setting('seq_page_cost')::real AS seq_cost
  FROM pg_class WHERE relname = 'bookings'
)
SELECT pages_fetched,
  rnd_cost - (rnd_cost - seq_cost) *
  sqrt(pages_fetched / relpages) AS cost_per_page,
  tuples_fetched
FROM t;
 pages_fetched | cost_per_page | tuples_fetched
---------------+---------------+----------------
         13447 |             1 |          31878
(1 row)
```

As usual, the I/O estimation is increased by the cost of processing each fetched tuple. If an exact bitmap is used, the number of tuples is estimated at the total number of tuples in the table multiplied by the selectivity of filter conditions. But if any bitmap segments are lossy, PostgreSQL has to access the corresponding pages to recheck all their tuples.



Thus, the estimation takes into account the expected fraction of lossy bitmap segments (which can be calculated based on the total number of selected rows and the bitmap size limit defined by *work_mem*).[1] v. 11

[1] backend/optimizer/path/costsize.c, compute_bitmap_pages function

The total cost of condition rechecks also increases the estimation (regardless of the bitmap accuracy).

The startup cost estimation of the Bitmap Heap Scan node is based on the total cost of the Bitmap Index Scan node, which is extended by the cost of bitmap processing:

```
                      QUERY PLAN
-----------------------------------------------------
 Bitmap Heap Scan on bookings
   (cost=599.48..14444.96 rows=31878 width=21)
   Recheck Cond: (total_amount = 28000.00)
   -> Bitmap Index Scan on bookings_total_amount_idx
       (cost=0.00..591.51 rows=31878 width=0)
       Index Cond: (total_amount = 28000.00)
(6 rows)
```

Here the bitmap is exact, and the cost is estimated roughly as follows:[1]

```
=> WITH t AS (
  SELECT 1 AS cost_per_page,
         13447 AS pages_fetched,
         31878 AS tuples_fetched
),
costs(startup_cost, run_cost) AS (
  SELECT
    ( SELECT round(
        589 /* cost estimation for the child node */ +
        0.1 * current_setting('cpu_operator_cost')::real *
        reltuples * 0.0151
      )
      FROM pg_class WHERE relname = 'bookings_total_amount_idx'
    ),
    ( SELECT round(
        cost_per_page * pages_fetched +
        current_setting('cpu_tuple_cost')::real * tuples_fetched +
        current_setting('cpu_operator_cost')::real * tuples_fetched
      )
      FROM t
    )
)
SELECT startup_cost, run_cost,
  startup_cost + run_cost AS total_cost
FROM costs;
```

[1] backend/optimizer/path/costsize.c, cost_bitmap_heap_scan function

```
 startup_cost | run_cost | total_cost
--------------+----------+------------
          597 |    13845 |      14442
(1 row)
```

If the query plan combines several bitmaps, the sum of the costs of separate index scans is increased by a (small) cost of merging them together.[1]

20.4 Parallel Index Scans

All the index scanning modes—a regular index scan, an index-only scan, and a bitmap scan—have their own flavors for parallel plans. v. 9.6 p. 340

The cost of parallel execution is estimated in the same way as that of sequential one, but (just like in the case of a parallel sequential scan) CPU resources are distributed between all parallel processes, thus reducing the total cost. The I/O component of the cost is not distributed because processes are synchronized to perform page access sequentially.

Now let me show you several examples of parallel plans without breaking down their cost estimation.

A parallel index scan:

```
=> EXPLAIN SELECT sum(total_amount)
FROM bookings WHERE book_ref < '400000';
                              QUERY PLAN
-----------------------------------------------------------------------
 Finalize Aggregate  (cost=19192.81..19192.82 rows=1 width=32)
   -> Gather  (cost=19192.59..19192.80 rows=2 width=32)
       Workers Planned: 2
       -> Partial Aggregate  (cost=18192.59..18192.60 rows=1 widt...
           -> Parallel Index Scan using bookings_pkey on bookings
               (cost=0.43..17642.82 rows=219907 width=6)
               Index Cond: (book_ref < '400000'::bpchar)
(7 rows)
```

[1] backend/optimizer/path/costsize.c, cost_bitmap_and_node & cost_bitmap_or_node functions

While a parallel scan of a B-tree is in progress, the ID of the current index page is kept in the server's shared memory. The initial value is set by the process that starts the scan: it traverses the tree from the root to the first suitable leaf page and saves its ID. Workers access subsequent index pages as needed, replacing the saved ID. Having fetched a page, the worker iterates through all its suitable entries and reads the corresponding heap tuples. The scanning completes when the worker has read the whole range of values that satisfy the query filter.

A parallel index-only scan:

```
=> EXPLAIN SELECT sum(total_amount)
FROM bookings WHERE total_amount < 50000.00;
                              QUERY PLAN
----------------------------------------------------------------------
 Finalize Aggregate  (cost=23370.60..23370.61 rows=1 width=32)
   -> Gather  (cost=23370.38..23370.59 rows=2 width=32)
       Workers Planned: 2
       -> Partial Aggregate  (cost=22370.38..22370.39 rows=1 widt...
           -> Parallel Index Only Scan using bookings_total_amoun...
               (cost=0.43..21387.27 rows=393244 width=6)
               Index Cond: (total_amount < 50000.00)
(7 rows)
```

A parallel index-only scan skips heap access for all-visible pages; it is the only difference it has from a parallel index scan.

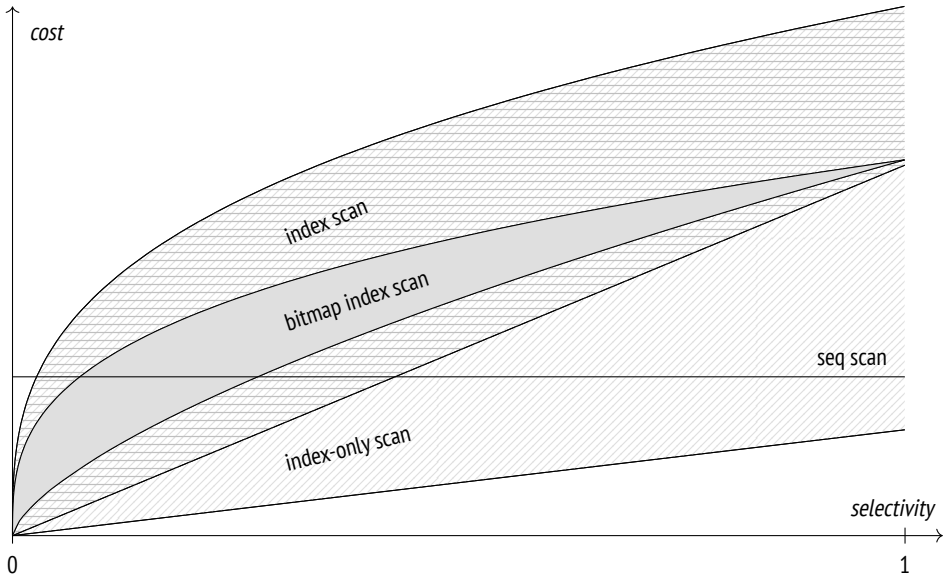
A parallel bitmap scan:

```
=> EXPLAIN SELECT sum(total_amount)
FROM bookings WHERE book_date < '2016-10-01';
                              QUERY PLAN
----------------------------------------------------------------------
 Finalize Aggregate  (cost=21492.21..21492.22 rows=1 width=32)
   -> Gather  (cost=21491.99..21492.20 rows=2 width=32)
       Workers Planned: 2
       -> Partial Aggregate  (cost=20491.99..20492.00 rows=1 widt...
           -> Parallel Bitmap Heap Scan on bookings
               (cost=4891.17..20133.01 rows=143588 width=6)
               Recheck Cond: (book_date < '2016-10-01 00:00:00+03...
               -> Bitmap Index Scan on bookings_book_date_idx
                   (cost=0.00..4805.01 rows=344611 width=0)
                   Index Cond: (book_date < '2016-10-01 00:00:00+...
(10 rows)
```

A bitmap scan implies that a bitmap is always built sequentially, by a single leader process; for this reason, the name of the Bitmap Index Scan node does not contain the word Parallel. When the bitmap is ready, the Parallel Bitmap Heap Scan node starts a parallel heap scan. Workers access subsequent heap pages and process them concurrently.

20.5 Comparison of Various Access Methods

The following illustration shows how costs of various access methods depend on selectivity of filter conditions:



It is a qualitative diagram; the actual figures are of course dependent on the particular table and server configuration.

Sequential scanning does not depend on selectivity, and starting from a certain fraction of selected rows, it is usually more efficient than other methods.

The cost of an index scan is affected by the correlation between the physical order of tuples and the order in which their IDs are returned by the access method. If the correlation is perfect, an index scan can be quite efficient even if the fraction

of selected rows is rather high. However, for low correlation (which is much more common) it can quickly become even more expensive than a sequential scan. That said, index scanning is still an absolute leader when it comes to selecting a single row using an index (typically a unique one).

If applicable, index-only scans can show great performance and beat sequential scans even if all the rows are selected. However, their performance is highly dependent on the visibility map, and in the worst-case scenario an index-only scan can degrade to a regular index scan.

The cost of a bitmap scan is affected by the size of available memory, but to a much lesser extent than an index scan cost depends on correlation. If the correlation is low, the bitmap scan turns out to be much cheaper.

Each access method has its own perfect usage scenarios; there is no such method that always outperforms other methods. The planner has to do extensive calculations to estimate the efficiency of each method in each particular case. Clearly, the accuracy of these estimations highly depends on the accuracy of the collected statistics.

21

Nested Loop

21.1 Join Types and Methods

Joins are a key feature of the SQL language; they serve as the foundation for its power and flexibility. Sets of rows (either retrieved from tables directly or received as the result of some other operations) are always joined pairwise.

There are several *types* of joins:

Inner joins. An *inner join* (specified as INNER JOIN, or simply JOIN) comprises those pairs of rows of two sets that satisfy a particular *join condition*. The join condition combines some columns of one set of rows with some columns of the other set; all the columns involved constitute the *join key*.

If the join condition demands that join keys of two sets be equal, such a join is called an *equi-join*; this is the most common join type.

A *Cartesian product* (CROSS JOIN) of two sets comprises all the possible pairs of rows of these sets—it is a special case of an inner join with a true condition.

Outer joins. A *left outer join* (specified as LEFT OUTER JOIN, or simply LEFT JOIN) extends the result of an inner join by those rows of the left set that have no match in the right set (the corresponding right-side columns are filled with NULL values).

The same is also true for a *right outer join* (RIGHT JOIN), down to the permutation of sets.

A *full outer join* (specified as FULL JOIN) comprises left and right outer joins, adding both right-side and left-side rows for which no match has been found.

Anti-Joins and Semi-Joins. A *semi-join* looks a lot like an inner join, but it includes only those rows of the left set that have a match in the right set (a row is included only once even if there are several matches).

An *anti-join* includes those rows of a set that have no match in the other set.

The SQL language has no explicit semi- and anti-joins, but the same outcome can be achieved using predicates like EXISTS and NOT EXISTS.

All these joins are logical operations. For example, an inner join is often described as a Cartesian product that has been cleared of the rows that do not satisfy the join condition. But at the physical level, an inner join is typically achieved via less expensive means.

PostgreSQL provides several join *methods*:

- a nested loop join
- a hash join
- a merge join

Join methods are algorithms that implement logical operations of SQL joins. These basic algorithms often have special flavors tailored for particular join types, even though they may support only some of them. For example, a nested loop supports an inner join (represented in the plan by a Nested Loop node) and a left outer join (represented by a Nested Loop Left Join node), but it cannot be used for full joins.

Some flavors of the same algorithms can also be used by other operations, such as aggregation.

Different join methods perform best in different conditions; it is the job of the planner to choose the most cost-effective one.

21.2 Nested Loop Joins

The basic algorithm of the nested loop join functions as follows. The outer loop traverses all the rows of the first set (called the *outer* set). For each of these rows,

the nested loop goes through the rows of the second set (called the *inner* set) to find the ones that satisfy the join condition. Each found pair is returned immediately as part of the query result.[1]

The algorithm accesses the inner set as many times as there are rows in the outer set. Therefore, the efficiency of nested loop joins depends on several factors:

- cardinality of the outer set of rows
- availability of an access method that can efficiently fetch the needed rows of the inner set
- recurrent access to the same rows of the inner set

Cartesian Product

A nested loop join is the most efficient way to find a Cartesian product, regardless of the number of rows in the sets:

```
=> EXPLAIN SELECT * FROM aircrafts_data a1
  CROSS JOIN aircrafts_data a2
WHERE a2.range > 5000;
                     QUERY PLAN
-----------------------------------------------------
 Nested Loop  (cost=0.00..2.78 rows=45 width=144)
   -> Seq Scan on aircrafts_data a1                  outer set
      (cost=0.00..1.09 rows=9 width=72)
   -> Materialize  (cost=0.00..1.14 rows=5 width=72)
      -> Seq Scan on aircrafts_data a2               inner set
         (cost=0.00..1.11 rows=5 width=72)
         Filter: (range > 5000)
(7 rows)
```

The Nested Loop node performs a join using the algorithm described above. It always has two child nodes: the one that is displayed higher in the plan corresponds to the outer set of rows, while the lower one represents the inner set.

[1] backend/executor/nodeNestloop.c

In this example, the inner set is represented by the Materialize node.[1] This node returns the rows received from its child node, having saved them for future use (the
4MB rows are accumulated in memory until their total size reaches *work_mem*; then PostgreSQL starts spilling them into a temporary file on disk). If accessed again, the node reads the accumulated rows without calling the child node. Thus, the executor can avoid scanning the full table again and read only those rows that satisfy the condition.

A similar plan can also be built for a query that uses a regular equi-join:

```
=> EXPLAIN SELECT *
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
WHERE t.ticket_no = '0005432000284';
                           QUERY PLAN
-----------------------------------------------------------------
 Nested Loop  (cost=0.99..25.05 rows=3 width=136)
   -> Index Scan using tickets_pkey on tickets t
       (cost=0.43..8.45 rows=1 width=104)
       Index Cond: (ticket_no = '0005432000284'::bpchar)
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
       (cost=0.56..16.58 rows=3 width=32)
       Index Cond: (ticket_no = '0005432000284'::bpchar)
(7 rows)
```

Having recognized the equality of the two values, the planner replaces the join condition tf.ticket_no = t.ticket_no by the tf.ticket_no = *constant* condition, virtually reducing an equi-join to a Cartesian product.[2]

Cardinality estimation. The cardinality of a Cartesian product is estimated at the product of cardinalities of the joined data sets: $3 = 1 \times 3$.

Cost estimation. The startup cost of the join operation combines the startup costs of all child nodes.

[1] backend/executor/nodeMaterial.c
[2] backend/optimizer/path/equivclass.c

The full cost of the join includes the following components:

- the cost of fetching all the rows of the outer set
- the cost of a single retrieval of all the rows of the inner set (since the cardinality estimation of the outer set equals one)
- the cost of processing each row to be returned

Here is a dependency graph for the cost estimation:

```
                              QUERY PLAN
-----------------------------------------------------------------
 Nested Loop  (cost=0.99..25.05 rows=3 width=136)
   -> Index Scan using tickets_pkey on tickets t
       (cost=0.43..8.45 rows=1 width=104)
       Index Cond: (ticket_no = '0005432000284'::bpchar)
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
       (cost=0.56..16.58 rows=3 width=32)
       Index Cond: (ticket_no = '0005432000284'::bpchar)
(7 rows)
```

×1

The cost of the join is calculated as follows:

```
=> SELECT 0.43 + 0.56 AS startup_cost,
  round((
    8.45 + 16.57 +
    3 * current_setting('cpu_tuple_cost')::real
  )::numeric, 2) AS total_cost;
 startup_cost | total_cost
--------------+------------
         0.99 |      25.05
(1 row)
```

Now let's get back to the previous example:

```
=> EXPLAIN SELECT *
FROM aircrafts_data a1
  CROSS JOIN aircrafts_data a2
WHERE a2.range > 5000;
```

```
                      QUERY PLAN
-----------------------------------------------------
 Nested Loop  (cost=0.00..2.78 rows=45 width=144)
   -> Seq Scan on aircrafts_data a1
       (cost=0.00..1.09 rows=9 width=72)
   -> Materialize  (cost=0.00..1.14 rows=5 width=72)
       -> Seq Scan on aircrafts_data a2
           (cost=0.00..1.11 rows=5 width=72)
           Filter: (range > 5000)
(7 rows)
```

The plan now contains the Materialize node; having once accumulated the rows received from its child node, Materialize returns them much faster for all the subsequent calls.

In general, the total cost of a join comprises the following expenses:[1]

- the cost of fetching all the rows of the outer set
- the cost of the initial fetch of all the rows of the inner set (during which materialization is performed)
- $(N-1)$-fold cost of repeat fetches of rows of the inner set (here N is the number of rows in the outer set)
- the cost of processing each row to be returned

The dependency graph here is as follows:

```
                      QUERY PLAN
-----------------------------------------------------
 Nested Loop  (cost=0.00..2.78 rows=45 width=144)
   -> Seq Scan on aircrafts_data a1
       (cost=0.00..1.09 rows=9 width=72)
                                  ×9
   -> Materialize
       (cost=0.00..1.14 rows=5 width=72)
       -> Seq Scan on aircrafts_data a2
           (cost=0.00..1.11 rows=5 width=72)
           Filter: (range > 5000)
(8 rows)
```

[1] backend/optimizer/path/costsize.c, initial_cost_nestloop andfinal_cost_nestloop function

In this example, materialization reduces the cost of repeat data fetches. The cost of the first Materialize call is shown in the plan, but all the subsequent calls are not listed. I will not provide any calculations here,[1] but in this particular case the estimation is 0.0125.

Thus, the cost of the join performed in this example is calculated as follows:

```
=> SELECT 0.00 + 0.00 AS startup_cost,
  round((
    1.09 + (1.14 + 8 * 0.0125) +
    45 * current_setting('cpu_tuple_cost')::real
  )::numeric, 2) AS total_cost;
 startup_cost | total_cost
--------------+------------
         0.00 |       2.78
(1 row)
```

Parameterized Joins

Now let's consider a more common example that does not boil down to a Cartesian product:

```
=> CREATE INDEX ON tickets(book_ref);
=> EXPLAIN SELECT *
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
WHERE t.book_ref = '03A76D';
                            QUERY PLAN
-----------------------------------------------------------------
 Nested Loop  (cost=0.99..45.68 rows=6 width=136)
   -> Index Scan using tickets_book_ref_idx on tickets t
       (cost=0.43..12.46 rows=2 width=104)
       Index Cond: (book_ref = '03A76D'::bpchar)
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
       (cost=0.56..16.58 rows=3 width=32)
       Index Cond: (ticket_no = t.ticket_no)
(7 rows)
```

[1] backend/optimizer/path/costsize.c, cost_rescan function

Here the Nested Loop node traverses the rows of the outer set (tickets), and for each of these rows it searches for the corresponding rows of the inner set (flights), passing the ticket number (t.ticket_no) to the condition *as a parameter*. When the inner node (Index Scan) is called, it has to deal with the condition ticket_no = *constant*.

Cardinality estimation. The planner estimates that the filter condition by a booking number is satisfied by two rows of the outer set (rows=2), and each of these rows matches three rows of the inner set on average (rows=3).

Join selectivity is a fraction of the Cartesian product of the two sets that remains after the join. It is obvious that we must exclude those rows of both sets that contain NULL values in the join key since the equality condition will never be satisfied for them.

The estimated cardinality equals the cardinality of the Cartesian product (that is, the product of cardinalities of the two sets) multiplied by the selectivity.[1]

Here the estimated cardinality of the first (outer) set is two rows. Since no conditions are applied to the second (inner) set except for the join condition itself, the cardinality of the second set is taken as the cardinality of the ticket_flights table.

Since the joined tables are connected by a foreign key, the selectivity estimation relies on the fact that each row of the child table has exactly one matching row in the parent table. So the selectivity is taken as the inverse of the size of the table referred to by the foreign key.[2]

Thus, for the case when the ticket_no columns contain no NULL values, the estimation is as follows:

```
=> SELECT round(2 * tf.reltuples * (1.0 / t.reltuples)) AS rows
FROM pg_class t, pg_class tf
WHERE t.relname = 'tickets'
  AND tf.relname = 'ticket_flights';
 rows
------
    6
(1 row)
```

[1] backend/optimizer/path/costsize.c, calc_joinrel_size_estimate function
[2] backend/optimizer/path/costsize.c, get_foreign_key_join_selectivity function

Clearly, tables can be also joined without using foreign keys. Then the selectivity will be taken as the estimated selectivities of the particular join conditions.[1]

For the equi-join in this example, the generic formula for selectivity estimation that assumes uniform distribution of values looks as follows: $\min\left(\frac{1}{nd_1}, \frac{1}{nd_2}\right)$, where nd_1 and nd_2 represent the number of distinct values of the join key in the first and second set, respectively.[2] p. 313

Statistics on distinct values show that ticket numbers in the tickets table are unique (which is only to be expected, as the ticket_no column is the primary key), and the ticket_flights has about three matching rows for each ticket:

```
=> SELECT t.n_distinct, tf.n_distinct
FROM pg_stats t, pg_stats tf
WHERE t.tablename = 'tickets' AND t.attname = 'ticket_no'
  AND tf.tablename = 'ticket_flights' AND tf.attname = 'ticket_no';
 n_distinct | n_distinct
------------+-------------
         -1 | -0.30347472
(1 row)
```

The result would match the estimation for the join with the foreign key:

```
=> SELECT round(2 * tf.reltuples *
  least(1.0/t.reltuples, 1.0/tf.reltuples/0.30347472)
) AS rows
FROM pg_class t, pg_class tf
WHERE t.relname = 'tickets' AND tf.relname = 'ticket_flights';
 rows
------
    6
(1 row)
```

The planner tries to refine this baseline estimation whenever possible. It cannot use histograms at the moment, but it takes MCV lists into account if such statistics have been collected on the join key for both tables.[3] The selectivity of the rows that appear in the list can be estimated more accurately, and only the remaining rows will have to rely on calculations that are based on uniform distribution. p. 315

[1] backend/optimizer/path/clausesel.c, clauselist_selectivity function
[2] backend/utils/adt/selfuncs.c, eqjoinsel function
[3] backend/utils/adt/selfuncs.c, eqjoinsel function

In general, join selectivity estimation is likely to be more accurate if the foreign key is defined. It is especially true for composite join keys, as the selectivity is often largely underestimated in this case.

Using the EXPLAIN ANALYZE command, you can view not only the actual number of rows, but also the number of times the inner loop has been executed:

```
=> EXPLAIN (analyze, timing off, summary off) SELECT *
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
WHERE t.book_ref = '03A76D';
                            QUERY PLAN
-------------------------------------------------------------------
 Nested Loop  (cost=0.99..45.68 rows=6 width=136)
   (actual rows=8 loops=1)
   -> Index Scan using tickets_book_ref_idx on tickets t
       (cost=0.43..12.46 rows=2 width=104) (actual rows=2 loops=1)
       Index Cond: (book_ref = '03A76D'::bpchar)
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
       (cost=0.56..16.58 rows=3 width=32) (actual rows=4 loops=2)
       Index Cond: (ticket_no = t.ticket_no)
(8 rows)
```

The outer set contains two rows (actual rows=2); the estimation has been correct. So the Index Scan node was executed twice (loops=2), and each time it selected four rows on average (actual rows=4). Hence the total number of found rows: actual rows=8.

> I do not show the execution time of each stage of the plan (TIMING OFF) for the output to fit the limited width of the page; besides, on some platforms an output with timing enabled can significantly slow down query execution. But if we did include it, PostgreSQL would display an average value, just like for the row count. To get the total execution time, you should multiply this value by the number of iterations (loops).

Cost estimation. The cost estimation formula here is the same as in the previous examples.

Let's recall our query plan:

```
=> EXPLAIN SELECT *
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
WHERE t.book_ref = '03A76D';
                           QUERY PLAN
-----------------------------------------------------------------
 Nested Loop  (cost=0.99..45.68 rows=6 width=136)
   -> Index Scan using tickets_book_ref_idx on tickets t
       (cost=0.43..12.46 rows=2 width=104)
       Index Cond: (book_ref = '03A76D'::bpchar)
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
       (cost=0.56..16.58 rows=3 width=32)
       Index Cond: (ticket_no = t.ticket_no)
(7 rows)
```

In this case, the cost of each subsequent scan of the inner set is the same as that of the first scan. So we ultimately get the following figures:

```
=> SELECT 0.43 + 0.56 AS startup_cost,
  round((
    12.46 + 2 * 16.57 +
    6 * current_setting('cpu_tuple_cost')::real
  )::numeric, 2) AS total_cost;
 startup_cost | total_cost
--------------+------------
         0.99 |      45.66
(1 row)
```

Caching Rows (Memoization)

v. 14

If the inner set is repeatedly scanned with the same parameter values (thus giving the same results), it may turn out to be beneficial to cache the rows of this set.

Such caching is performed by the Memoize[1] node. Being similar to the Materialize node, it is designed to handle parameterized joins and has a much more complex implementation:

[1] backend/executor/nodeMemoize.c

- The Materialize node simply materializes all the rows returned by its child node, while Memoize ensures that the rows returned for different parameter values are kept separately.
- In the event of an overflow, the Materialize storage starts spilling rows to disk, while Memoize keeps all the rows in memory (there would otherwise be no point in caching).

Here is an example of a query that uses Memoize:

```
=> EXPLAIN SELECT *
FROM flights f
  JOIN aircrafts_data a ON f.aircraft_code = a.aircraft_code
WHERE f.flight_no = 'PG0003';
                              QUERY PLAN
----------------------------------------------------------------------
 Nested Loop  (cost=5.44..387.10 rows=113 width=135)
   -> Bitmap Heap Scan on flights f
       (cost=5.30..382.22 rows=113 width=63)
       Recheck Cond: (flight_no = 'PG0003'::bpchar)
       -> Bitmap Index Scan on flights_flight_no_scheduled_depart...
           (cost=0.00..5.27 rows=113 width=0)
           Index Cond: (flight_no = 'PG0003'::bpchar)
   -> Memoize  (cost=0.15..0.27 rows=1 width=72)
       Cache Key: f.aircraft_code
       Cache Mode: logical
       -> Index Scan using aircrafts_pkey on aircrafts_data a
           (cost=0.14..0.26 rows=1 width=72)
           Index Cond: (aircraft_code = f.aircraft_code)
(13 rows)
```

4MB 1.0 The size of the memory chunk used to store cached rows equals *work_mem* × × *hash_mem_multiplier*. As implied by the second parameter's name, cached rows are stored in a hash table (with open addressing).[1] The hash key (shown as Cache Key in the plan) is the parameter value (or several values if there are more than one parameter).

All the hash keys are bound into a list; one of its ends is considered cold (since it contains the keys that have not been used for a long time), while the other is hot (it stores recently used keys).

[1] include/lib/simplehash.h

If a call on the Memoize node shows that the passed parameter values correspond to the already cached rows, these rows will be passed on to the parent node (Nested Loop) without checking the child node. The used hash key is then moved to the hot end of the list.

If the cache does not contain the required rows, the Memoize node pulls them from its child node, caches them, and passes them on to the node above. The corresponding hash key also becomes hot.

As new data is being cached, it can fill all the available memory. To free some space, the rows that correspond to cold keys get evicted. This eviction algorithm differs from the one used in the buffer cache but serves the same purpose. *p. 177*

Some parameter values may turn out to have so many matching rows that they do not fit into the allocated memory chunk, even if all the other rows are already evicted. Such parameters are skipped—it makes no sense to cache only some of the rows since the next call will still have to get all the rows from the child node.

Cost and cardinality estimations. These calculations are quite similar to what we have already seen above. We just have to bear in mind that the cost of the Memoize node shown in the plan has nothing to do with its actual cost: it is simply the cost of its child node increased by the *cpu_tuple_cost* value.[1] 0.01

We have already come across a similar situation for the Materialize node: its cost is only calculated for *subsequent scans*[2] and is not reflected in the plan.

Clearly, it only makes sense to use Memoize if it is cheaper than its child node. The cost of each subsequent Memoize scan depends on the expected cache access profile and the size of the memory chunk that can be used for caching. The calculated value is highly dependent on the accurate estimation of the number of distinct parameter values to be used in the scans of the inner set of rows.[3] Based on this number, you can weigh the probabilities of the rows to be cached and to be evicted from the cache. The expected hits reduce the estimated cost, while potential evictions increase it. We will skip the details of these calculations here.

[1] backend/optimizer/util/pathnode.c, create_memoize_path function
[2] backend/optimizer/path/costsize.c, cost_memoize_rescan function
[3] backend/utils/adt/selfuncs.c, estimate_num_groups function

To figure out what is actually going on during query execution, we will use the EXPLAIN ANALYZE command, as usual:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM flights f
  JOIN aircrafts_data a ON f.aircraft_code = a.aircraft_code
WHERE f.flight_no = 'PG0003';
                              QUERY PLAN
----------------------------------------------------------------------
 Nested Loop (actual rows=113 loops=1)
   -> Bitmap Heap Scan on flights f
       (actual rows=113 loops=1)
       Recheck Cond: (flight_no = 'PG0003'::bpchar)
       Heap Blocks: exact=2
       -> Bitmap Index Scan on flights_flight_no_scheduled_depart...
           (actual rows=113 loops=1)
           Index Cond: (flight_no = 'PG0003'::bpchar)
   -> Memoize (actual rows=1 loops=113)
       Cache Key: f.aircraft_code
       Cache Mode: logical
       Hits: 112  Misses: 1  Evictions: 0  Overflows: 0  Memory
       Usage: 1kB
       -> Index Scan using aircrafts_pkey on aircrafts_data a
           (actual rows=1 loops=1)
           Index Cond: (aircraft_code = f.aircraft_code)
(16 rows)
```

This query selects the flights that follow the same route and are performed by aircraft of a particular type, so all the calls on the Memoize node use the same hash key. The first row has to be fetched from the table (Misses: 1), but all the subsequent rows are found in the cache (Hits: 112). The whole operation takes just 1 kB of memory.

The other two displayed values are zero: they represent the number of evictions and the number of cache overflows when it was impossible to cache all the rows related to a particular set of parameters. Large figures would indicate that the allocated cache is too small, which might be caused by inaccurate estimation of the number of distinct parameter values. Then the use of the Memoize node can turn out to be quite expensive. In the extreme case, you can forbid the planner to
on use caching by turning off the *enable_memoize* parameter.

Outer Joins

The nested loop join can be used to perform the *left outer join*:

```
=> EXPLAIN SELECT *
FROM ticket_flights tf
  LEFT JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no
                               AND bp.flight_id = tf.flight_id
WHERE tf.ticket_no = '0005434026720';
                              QUERY PLAN
----------------------------------------------------------------------
 Nested Loop Left Join  (cost=1.12..33.35 rows=3 width=57)
   Join Filter: ((bp.ticket_no = tf.ticket_no) AND (bp.flight_id =
   tf.flight_id))
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
       (cost=0.56..16.58 rows=3 width=32)
       Index Cond: (ticket_no = '0005434026720'::bpchar)
   -> Materialize  (cost=0.56..16.62 rows=3 width=25)
       -> Index Scan using boarding_passes_pkey on boarding_passe...
           (cost=0.56..16.61 rows=3 width=25)
           Index Cond: (ticket_no = '0005434026720'::bpchar)
(10 rows)
```

Here the join operation is represented by the Nested Loop Left Join node. The planner has chosen a non-parameterized join with a filter: it performs identical scans of the inner set of rows (so this set is hidden behind the Materialize node) and returns the rows that satisfy the filter condition (Join Filter).

The cardinality of the outer join is estimated just like the one of the inner join, except that the calculated estimation is compared with the cardinality of the outer set of rows, and the bigger value is taken as the final result.[1] In other words, the outer join never reduces the number of rows (but can increase it).

The cost estimation is similar to that of the inner join.

We must also keep in mind that the planner can select different plans for inner and outer joins. Even this simple example will have a different Join Filter if the planner is forced to use a nested loop join:

[1] backend/optimizer/path/costsize.c, calc_joinrel_size_estimate function

```
=> SET enable_mergejoin = off;
=> EXPLAIN SELECT *
FROM ticket_flights tf
  JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no
                         AND bp.flight_id = tf.flight_id
WHERE tf.ticket_no = '0005434026720';
                            QUERY PLAN
----------------------------------------------------------------------
 Nested Loop  (cost=1.12..33.33 rows=3 width=57)
   Join Filter: (tf.flight_id = bp.flight_id)
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
       (cost=0.56..16.58 rows=3 width=32)
       Index Cond: (ticket_no = '0005434026720'::bpchar)
   -> Materialize  (cost=0.56..16.62 rows=3 width=25)
       -> Index Scan using boarding_passes_pkey on boarding_passe...
           (cost=0.56..16.61 rows=3 width=25)
           Index Cond: (ticket_no = '0005434026720'::bpchar)
(9 rows)
=> RESET enable_mergejoin;
```

A slight difference in the total cost is caused by the fact that the outer join must also check ticket numbers to get the correct result if there is no match in the outer set of rows.

Right joins are not supported,[1] as the nested loop algorithm treats the inner and outer sets differently. The outer set is scanned in full; as for the inner set, the index access allows reading only those rows that satisfy the join condition, so some of its rows may be skipped altogether.

A full join is not supported for the same reason.

Anti- and Semi-joins

Anti-joins and semi-joins are similar in the sense that for each row of the first (outer) set it is enough to find only *one* matching row in the second (inner) set.

An *anti-join* returns the rows of the first set only if they have no match in the second set: as soon as the executor finds the first matching row in the second set, it can

[1] backend/optimizer/path/joinpath.c, match_unsorted_outer function

exit the current loop: the corresponding row of the first set must be excluded from the result.

Anti-joins can be used to compute the NOT EXISTS predicate.

For example, let's find aircraft models with undefined cabin configuration. The corresponding plan contains the Nested Loop Anti Join node:

```
=> EXPLAIN SELECT *
FROM aircrafts a
WHERE NOT EXISTS (
  SELECT * FROM seats s WHERE s.aircraft_code = a.aircraft_code
);
                              QUERY PLAN
----------------------------------------------------------------------
 Nested Loop Anti Join  (cost=0.28..4.65 rows=1 width=40)
   -> Seq Scan on aircrafts_data ml  (cost=0.00..1.09 rows=9 widt...
   -> Index Only Scan using seats_pkey on seats s
       (cost=0.28..5.55 rows=149 width=4)
       Index Cond: (aircraft_code = ml.aircraft_code)
(5 rows)
```

An alternative query without the NOT EXISTS predicate will have the same plan:

```
=> EXPLAIN SELECT a.*
FROM aircrafts a
  LEFT JOIN seats s ON a.aircraft_code = s.aircraft_code
WHERE s.aircraft_code IS NULL;
                              QUERY PLAN
----------------------------------------------------------------------
 Nested Loop Anti Join  (cost=0.28..4.65 rows=1 width=40)
   -> Seq Scan on aircrafts_data ml  (cost=0.00..1.09 rows=9 widt...
   -> Index Only Scan using seats_pkey on seats s
       (cost=0.28..5.55 rows=149 width=4)
       Index Cond: (aircraft_code = ml.aircraft_code)
(5 rows)
```

A *semi-join* returns those rows of the first set that have at least one match in the second set (again, there is no need to check the set for other matches—the result is already known).

A semi-join can be used to compute the EXISTS predicate. Let's find the aircraft models with seats installed in the cabin:

```
=> EXPLAIN SELECT *
FROM aircrafts a
WHERE EXISTS (
  SELECT * FROM seats s
  WHERE s.aircraft_code = a.aircraft_code
);
                              QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop Semi Join  (cost=0.28..6.67 rows=9 width=40)
   -> Seq Scan on aircrafts_data ml  (cost=0.00..1.09 rows=9 widt...
   -> Index Only Scan using seats_pkey on seats s
       (cost=0.28..5.55 rows=149 width=4)
       Index Cond: (aircraft_code = ml.aircraft_code)
(5 rows)
```

The Nested Loop Semi Join node represents the same-name join method. This plan (just like the anti-join plans above) provides the basic estimation of the number of rows in the seats table (rows=149), although it is enough to retrieve only one of them. The actual query execution stops after fetching the first row, of course:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM aircrafts a
WHERE EXISTS (
  SELECT * FROM seats s
  WHERE s.aircraft_code = a.aircraft_code
);
                          QUERY PLAN
-------------------------------------------------------------
 Nested Loop Semi Join (actual rows=9 loops=1)
   -> Seq Scan on aircrafts_data ml (actual rows=9 loops=1)
   -> Index Only Scan using seats_pkey on seats s
       (actual rows=1 loops=9)
       Index Cond: (aircraft_code = ml.aircraft_code)
       Heap Fetches: 0
(6 rows)
```

Cardinality estimation. The selectivity of a semi-join is estimated in the usual manner, except that the cardinality of the inner set is taken as one. For anti-joins, the estimated selectivity is subtracted from one, just like for negation.[1]

[1] backend/optimizer/path/costsize.c, calc_joinrel_size_estimate function

Cost estimation. For anti- and semi-joins, the cost estimation reflects the fact that the scan of the second set stops as soon as the first matching row is found.[1]

Non-Equi-joins

The nested loop algorithm allows joining sets of rows based on any join condition.

Obviously, if the inner set is a base table with an index created on it, and the join condition uses an operator that belongs to an operator class of this index, the access to the inner set can be quite efficient. But it is always possible to perform the join by calculating a Cartesian product of rows filtered by some condition—which can be absolutely arbitrary in this case. Like in the following query, which selects pairs of airports that are located close to each other: p. 357

```
=> CREATE EXTENSION earthdistance CASCADE;
=> EXPLAIN (costs off) SELECT *
FROM airports a1
  JOIN airports a2 ON a1.airport_code != a2.airport_code
                  AND a1.coordinates <@> a2.coordinates < 100;
                              QUERY PLAN
----------------------------------------------------------------------
 Nested Loop
   Join Filter: ((ml.airport_code <> ml_1.airport_code) AND
   ((ml.coordinates <@> ml_1.coordinates) < '100'::double precisi...
   -> Seq Scan on airports_data ml
   -> Materialize
       -> Seq Scan on airports_data ml_1
(6 rows)
```

Parallel Mode

v. 9.6

A nested loop join can participate in parallel plan execution.[2] p. 340

It is only the outer set that can be processed in parallel, as it can be scanned by several workers simultaneously. Having fetched an outer row, each worker then has to search for the matching rows in the inner set, which is done sequentially.

[1] backend/optimizer/path/costsize.c, final_cost_nestloop function
[2] backend/optimizer/path/joinpath.c, consider_parallel_nestloop function

The query shown below includes several joins; it searches for passengers that have tickets for a particular flight:

```
=> EXPLAIN (costs off) SELECT t.passenger_name
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
  JOIN flights f ON f.flight_id = tf.flight_id
WHERE f.flight_id = 12345;
                            QUERY PLAN
----------------------------------------------------------------------
 Nested Loop
   -> Index Only Scan using flights_flight_id_status_idx on fligh...
       Index Cond: (flight_id = 12345)
   -> Gather
       Workers Planned: 2
       -> Nested Loop
           -> Parallel Seq Scan on ticket_flights tf
               Filter: (flight_id = 12345)
           -> Index Scan using tickets_pkey on tickets t
               Index Cond: (ticket_no = tf.ticket_no)
(10 rows)
```

At the upper level, the nested loop join is performed sequentially. The outer set consists of a single row of the flights table fetched by a unique key, so the use of a nested loop is justified even for a large number of inner rows.

p. 341 The inner set is retrieved using a parallel plan. Each of the workers scans its own share of rows of the ticket_flights table and joins them with tickets using the nested loop algorithm.

22

Hashing

22.1 Hash Joins

One-Pass Hash Joins

A hash join searches for matching rows using a pre-built hash table. Here is an example of a plan with such a join:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no;
                QUERY PLAN
-----------------------------------------
 Hash Join
   Hash Cond: (tf.ticket_no = t.ticket_no)
   -> Seq Scan on ticket_flights tf
   -> Hash
       -> Seq Scan on tickets t
(5 rows)
```

At the **first stage**, the Hash Join node[1] calls the Hash node,[2] which pulls the whole inner set of rows from its child node and places it into a *hash table*.

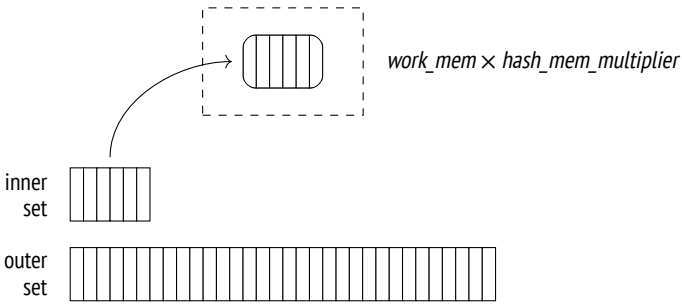
Storing pairs of *hash keys* and *values*, the hash table enables fast access to a value by its key; the search time does not depend on the size of the hash table, as hash keys are distributed more or less uniformly between a limited number of *buckets*. The bucket to which a given key goes is determined by the *hash function* of the hash key; since the number of buckets is always a power of two, it is enough to take the required number of bits of the computed value.

[1] backend/executor/nodeHashjoin.c
[2] backend/executor/nodeHash.c

p. 172 Just like the buffer cache, this implementation uses a dynamically extendible hash table that resolves hash collisions by chaining.[1]

At the **first stage** of a join operation, the inner set is scanned, and the hash function is computed for each of its rows. The columns referenced in the join condition (Hash Cond) serve as the hash key, while the hash table itself stores all the queried fields of the inner set.

v. 13 A hash join is most efficient if the whole hash table can be accommodated in RAM, as the executor manages to process the data in one batch in this case. The size
4MB of the memory chunk allocated for this purpose is limited by the *work_mem* ×
1.0 × *hash_mem_multiplier* value.



Let's run EXPLAIN ANALYZE to take a look at statistics on memory usage of a query:

```
=> SET work_mem = '256MB';
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM bookings b
  JOIN tickets t ON b.book_ref = t.book_ref;
                          QUERY PLAN
-----------------------------------------------------------------
 Hash Join (actual rows=2949857 loops=1)
   Hash Cond: (t.book_ref = b.book_ref)
   -> Seq Scan on tickets t (actual rows=2949857 loops=1)
   -> Hash (actual rows=2111110 loops=1)
       Buckets: 4194304  Batches: 1  Memory Usage: 145986kB
       -> Seq Scan on bookings b (actual rows=2111110 loops=1)
(6 rows)
```

[1] backend/utils/hash/dynahash.c

Unlike a nested loop join, which treats inner and outer sets differently, a hash join can swap them around. The smaller set is usually used as the inner one, as it results in a smaller hash table.

In this example, the whole table fits into the allocated cache: it takes about 143 MB (Memory Usage) and contains $4\text{ M} = 2^{22}$ buckets. So the join is performed in one pass (Batches).

But if the query referred to only one column, the hash table would fit 111 MB:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT b.book_ref
FROM bookings b
  JOIN tickets t ON b.book_ref = t.book_ref;
                          QUERY PLAN
----------------------------------------------------------------
 Hash Join (actual rows=2949857 loops=1)
   Hash Cond: (t.book_ref = b.book_ref)
   -> Index Only Scan using tickets_book_ref_idx on tickets t
       (actual rows=2949857 loops=1)
       Heap Fetches: 0
   -> Hash (actual rows=2111110 loops=1)
       Buckets: 4194304  Batches: 1  Memory Usage: 113172kB
       -> Seq Scan on bookings b (actual rows=2111110 loops=1)
(8 rows)
=> RESET work_mem;
```

It is yet another reason to avoid referring to superfluous fields in a query (which can happen if you are using an asterisk, to give one example).

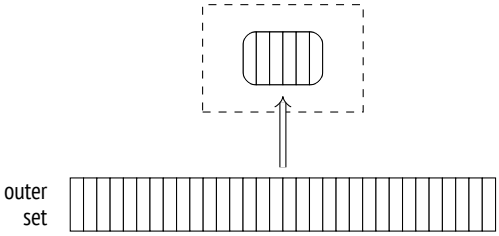
The chosen number of buckets should guarantee that each bucket holds only one row on average when the hash table is completely filled with data. Higher density would increase the rate of hash collisions, making the search less efficient, while a less compact hash table would take up too much memory. The estimated number of buckets is increased up to the nearest power of two.[1]

(If the estimated hash table size exceeds the memory limit based on the average width of a single row, two-pass hashing will be applied.)

A hash join cannot start returning results until the hash table is fully built.

[1] backend/executor/nodeHash.c, ExecChooseHashTableSize function

At the **second stage** (the hash table is already built by this time), the Hash Join node calls on its second child node to get the outer set of rows. For each scanned row, the hash table is searched for a match. It requires calculating the hash key for the columns of the outer set that are included into the join condition.



The found matches are returned to the parent node.

p. 404 **Cost estimation.** We have already covered cardinality estimation; since it does not depend on the join method, I will now focus on cost estimation.

The cost of the Hash node is represented by the total cost of its child node. It is a dummy number that simply fills the slot in the plan.[1] All the actual estimations are included into the cost of the Hash Join node.[2]

Here is an example:

```
=> EXPLAIN (analyze, timing off, summary off)
SELECT * FROM flights f
  JOIN seats s ON s.aircraft_code = f.aircraft_code;
                              QUERY PLAN
----------------------------------------------------------------------
 Hash Join  (cost=38.13..278507.28 rows=16518865 width=78)
   (actual rows=16518865 loops=1)
   Hash Cond: (f.aircraft_code = s.aircraft_code)
   -> Seq Scan on flights f  (cost=0.00..4772.67 rows=214867 widt...
       (actual rows=214867 loops=1)
   -> Hash  (cost=21.39..21.39 rows=1339 width=15)
       (actual rows=1339 loops=1)
       Buckets: 2048  Batches: 1  Memory Usage: 79kB
       -> Seq Scan on seats s  (cost=0.00..21.39 rows=1339 width=15)
           (actual rows=1339 loops=1)
(10 rows)
```

[1] backend/optimizer/plan/createplan.c, create_hashjoin_plan function
[2] backend/optimizer/path/costsize.c, initial_cost_hashjoin and final_cost_hashjoin functions

The startup cost of the join reflects primarily the cost of hash table creation and includes the following components:

- the total cost of fetching the inner set, which is required to build the hash table
- the cost of calculating the hash function of all the columns included into the join key, for each row of the inner set (estimated at *cpu_operator_cost* per operation) 0.0025
- the cost of insertion of all the inner rows into the hash table (estimated at *cpu_tuple_cost* per inserted row) 0.01
- the startup cost of fetching the outer set of rows, which is required to start the join operation

The total cost comprises the startup cost and the cost of the join itself, namely:

- the cost of computing the hash function of all the columns included into the join key, for each row of the outer set (*cpu_operator_cost*)
- the cost of join condition rechecks, which are required to address possible hash collisions (estimated at *cpu_operator_cost* per each checked operator)
- the processing cost for each resulting row (*cpu_tuple_cost*)

The number of required rechecks is the hardest to estimate. It is calculated by multiplying the number of rows of the outer set by some fraction of the inner set (stored in the hash table). To estimate this fraction, the planner has to take into account that data distribution may be non-uniform. I will spare you the details of these computations;[1] in this particular case, this fraction is estimated at 0.150112.

Thus, the cost of our query is estimated as follows:

```
=> WITH cost(startup) AS (
  SELECT round((
    21.39 +
    current_setting('cpu_operator_cost')::real * 1339 +
    current_setting('cpu_tuple_cost')::real * 1339 +
    0.00
  )::numeric, 2)
)
```

[1] backend/utils/adt/selfuncs.c, estimate_hash_bucket_stats function

```
SELECT startup,
  startup + round((
    4772.67 +
    current_setting('cpu_operator_cost')::real * 214867 +
    current_setting('cpu_operator_cost')::real * 214867 * 1339 *
      0.150112 +
    current_setting('cpu_tuple_cost')::real * 16518865
  )::numeric, 2) AS total
FROM cost;
 startup |   total
---------+-----------
   38.13 | 278507.26
(1 row)
```

And here is the dependency graph:

```
                    QUERY PLAN
--------------------------------------------------
 Hash Join
   (cost=38.13..278507.28 rows=16518865 width=78)
   Hash Cond: (f.aircraft_code = s.aircraft_code)
   -> Seq Scan on flights f
       (cost=0.00..4772.67 rows=214867 width=63)
   -> Hash
       (cost=21.39..21.39 rows=1339 width=15)
       -> Seq Scan on seats s
           (cost=0.00..21.39 rows=1339 width=15)
(9 rows)
```

Two-Pass Hash Joins

If the planner's estimations show that the hash table will not fit the allocated memory, the inner set of rows is split into *batches* to be processed separately. The number of batches (just like the number of buckets) is always a power of two; the batch to use is determined by the corresponding number of bits of the hash key.[1]

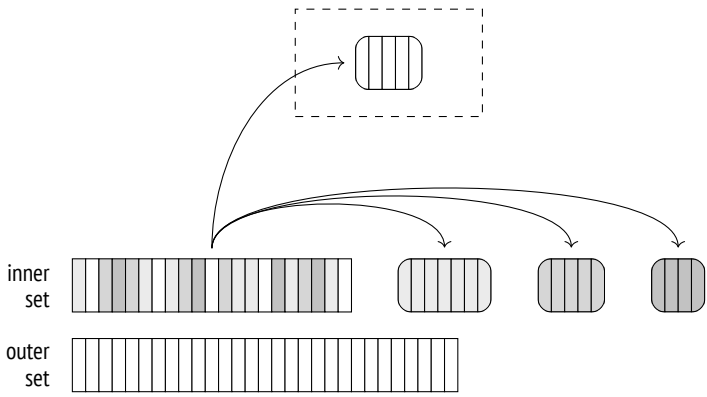
Any two matching rows belong to one and the same batch: rows placed into different batches cannot have the same hash code.

[1] backend/executor/nodeHash.c, ExecHashGetBucketAndBatch function

All batches hold an equal number of hash keys. If the data is distributed uniformly, batch sizes will also be roughly the same. The planner can control memory consumption by choosing an appropriate number of batches.[1]

At the **first stage**, the executor scans the inner set of rows to build the hash table. If the scanned row belongs to the first batch, it is added to the hash table and kept in RAM. Otherwise, it is written into a *temporary file* (there is a separate file for each batch).[2]

The total volume of temporary files that a session can store on disk is limited by the *temp_file_limit* parameter (temporary tables are not included into this limit). As soon as the session reaches this value, the query is aborted. -1

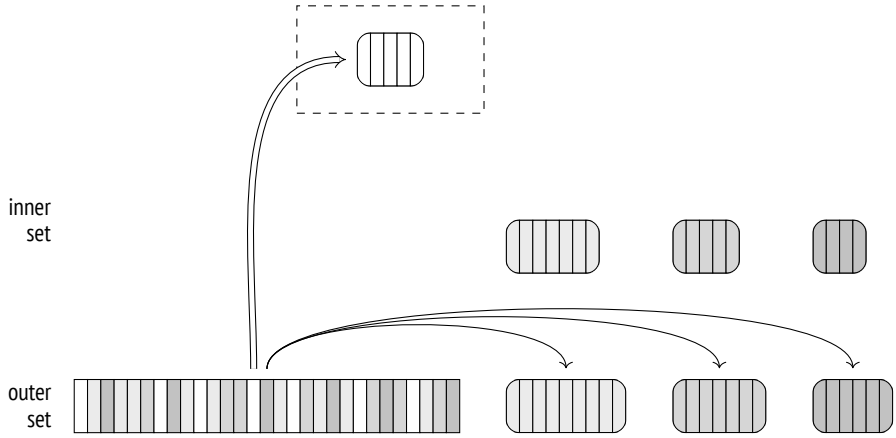


At the **second stage**, the outer set is scanned. If the row belongs to the first batch, it is matched against the hash table, which contains the first batch of rows of the inner set (there can be no matches in other batches anyway).

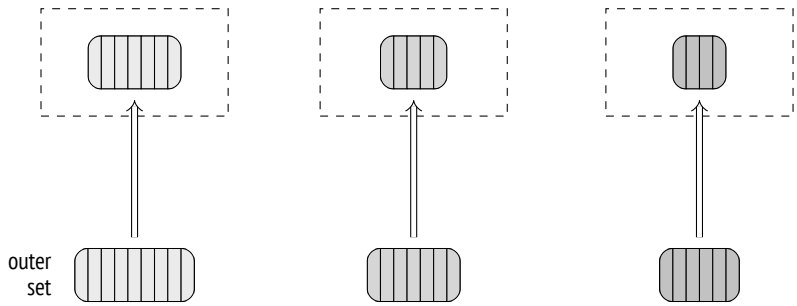
If the row belongs to a different batch, it is stored in a temporary file, which is again created separately for each batch. Thus, N batches can use $2(N-1)$ files (or fewer if some of the batches turn out to be empty).

Once the second stage is complete, the memory allocated for the hash table is freed. At this point, we already have the result of the join for one of the batches.

[1] backend/executor/nodeHash.c, ExecChooseHashTableSize function
[2] backend/executor/nodeHash.c, ExecHashTableInsert function



Both stages are repeated for each of the batches saved on disk: the rows of the inner set are transferred from the temporary file to the hash table; then the rows of the outer set related to the same batch are read from another temporary file and matched against this hash table. Once processed, temporary files get deleted.



Unlike a similar output for a one-pass join, the output of the EXPLAIN command for a two-pass join contains more than one batch. If run with the BUFFERS option, this command also displays statistics on disk access:

```
=> EXPLAIN (analyze, buffers, costs off, timing off, summary off)
SELECT *
FROM bookings b
  JOIN tickets t ON b.book_ref = t.book_ref;
```

```
                          QUERY PLAN
----------------------------------------------------------------
 Hash Join (actual rows=2949857 loops=1)
   Hash Cond: (t.book_ref = b.book_ref)
   Buffers: shared hit=7225 read=55637, temp read=55126
   written=55126
   -> Seq Scan on tickets t (actual rows=2949857 loops=1)
       Buffers: shared read=49415
   -> Hash (actual rows=2111110 loops=1)
       Buckets: 65536  Batches: 64  Memory Usage: 2277kB
       Buffers: shared hit=7225 read=6222, temp written=10858
       -> Seq Scan on bookings b (actual rows=2111110 loops=1)
           Buffers: shared hit=7225 read=6222
(11 rows)
```

I have already shown this query above with an increased *work_mem* setting. The default value of 4 MB is too small for the whole hash table to fit RAM; in this example, the data is split into 64 batches, and the hash table uses 64 K $= 2^{16}$ buckets. As the hash table is being built (the Hash node), the data is written into temporary files (temp written); at the join stage (the Hash Join node), temporary files are both read and written (temp read, written).

To collect more statistics on temporary files, you can set the *log_temp_files* parameter to zero. Then the server log will list all the temporary files and their sizes (as they appeared at the time of deletion). -1

Dynamic Adjustments

The planned course of events may be disrupted by two issues: inaccurate statistics and non-uniform data distribution.

If the distribution of values in the join key columns is non-uniform, different batches will have different sizes.

If some batch (except for the very first one) turns out to be too large, all its rows will have to be written to disk and then read from disk. It is the outer set that causes most of the trouble, as it is typically bigger. So if there are regular, non-multivariate statistics on MCVs of the outer set (that is, the outer set is represented p. 331

by a table, and the join is performed by a single column), rows with hash codes corresponding to MCVs are considered to be a part of the first batch.[1] This technique (called skew optimization) can reduce the I/O overhead of a two-pass join to some extent.

Because of these two factors, the size of some (or all) batches may exceed the estimation. Then the corresponding hash table will not fit the allocated memory chunk and will surpass the defined limits.

So if the hash table being built turns out too big, the number of batches is increased (doubled) on the fly. Each batch is virtually split into two new ones: about half of the rows (assuming that the distribution is uniform) is left in the hash table, while the other half is saved into a new temporary file.[2]

Such a split can happen even if a one-pass join has been originally planned. In fact, one- and two-pass joins use one and the same algorithm implemented by the same code; I single them out here solely for smoother narration.

The number of batches cannot be reduced. If it turns out that the planner has overestimated the data size, batches will not be merged together.

In the case of non-uniform distribution, increasing the number of batches may not help. For example, if the key column contains *one and the same* value in *all* its rows, they will be placed into the same batch since the hash function will be returning the same value over and over again. Unfortunately, the hash table will continue growing in this case, regardless of the imposed restrictions.

> In theory, this issue could be addressed by a multi-pass join, which would perform partial scans of the batch, but it is not supported.

To demonstrate a dynamic increase in the number of batches, we first have to perform some manipulations:

p. 311

```
=> CREATE TABLE bookings_copy (LIKE bookings INCLUDING INDEXES)
WITH (autovacuum_enabled = off);
=> INSERT INTO bookings_copy SELECT * FROM bookings;
INSERT 0 2111110
```

[1] backend/executor/nodeHash.c, ExecHashBuildSkewHash function
[2] backend/executor/nodeHash.c, ExecHashIncreaseNumBatches function

```
=> DELETE FROM bookings_copy WHERE random() < 0.9;
DELETE 1899264
=> ANALYZE bookings_copy;
=> INSERT INTO bookings_copy SELECT * FROM bookings
ON CONFLICT DO NOTHING;
INSERT 0 1899264
=> SELECT reltuples FROM pg_class WHERE relname = 'bookings_copy';
 reltuples
-----------
    211846
(1 row)
```

As a result, we get a new table called bookings_copy. It is an exact copy of the bookings table, but the planner underestimates the number of rows in it by ten times. A similar situation may occur if the hash table is generated for a set of rows produced by another join operation, so there is no reliable statistics available.

This miscalculation makes the planner think that 8 buckets are enough, but while the join is being performed, this number grows to 32:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM bookings_copy b
  JOIN tickets t ON b.book_ref = t.book_ref;
                            QUERY PLAN
-------------------------------------------------------------------
 Hash Join (actual rows=2949857 loops=1)
   Hash Cond: (t.book_ref = b.book_ref)
   -> Seq Scan on tickets t (actual rows=2949857 loops=1)
   -> Hash (actual rows=2111110 loops=1)
       Buckets: 65536 (originally 65536)  Batches: 32 (originally 8)
        Memory Usage: 4040kB
       -> Seq Scan on bookings_copy b (actual rows=2111110 loops=1)
(7 rows)
```

Cost estimation. I have already used this example to demonstrate cost estimation for a one-pass join, but now I am going to reduce the size of available memory to the minimum, so the planner will have to use two batches. It increases the cost of the join:

```
=> SET work_mem = '64kB';
=> EXPLAIN (analyze, timing off, summary off)
SELECT * FROM flights f
  JOIN seats s ON s.aircraft_code = f.aircraft_code;
                              QUERY PLAN
----------------------------------------------------------------------
 Hash Join  (cost=45.13..283139.28 rows=16518865 width=78)
   (actual rows=16518865 loops=1)
   Hash Cond: (f.aircraft_code = s.aircraft_code)
   -> Seq Scan on flights f  (cost=0.00..4772.67 rows=214867 widt...
       (actual rows=214867 loops=1)
   -> Hash  (cost=21.39..21.39 rows=1339 width=15)
       (actual rows=1339 loops=1)
       Buckets: 2048  Batches: 2  Memory Usage: 55kB
       -> Seq Scan on seats s  (cost=0.00..21.39 rows=1339 width=15)
           (actual rows=1339 loops=1)
(10 rows)
=> RESET work_mem;
```

The cost of the second pass is incurred by spilling rows into temporary files and reading them from these files.

The startup cost of a two-pass join is based on that of a one-pass join, which is increased by the estimated cost of writing as many pages as required to store all the necessary fields of *all* the rows of the inner set.[1] Although the first batch is not written to disk when the hash table is being built, the estimation does not take it into account and hence does not depend on the number of batches.

In its turn, the total cost comprises the total cost of a one-pass join and the estimated costs of reading the rows of the inner set previously stored on disk, as well as reading and writing the rows of the outer set.

Both writing and reading are estimated at *seq_page_cost* per page, as I/O operations are assumed to be sequential.

In this particular case, the number of pages required for the inner set is estimated at 7, while the data of the outer set is expected to fit 2309 pages. Having added these estimations to the one-pass join cost calculated above, we get the same figures as shown in the query plan:

[1] backend/optimizer/path/costsize.c, page_size function

```
=> SELECT 38.13 + -- startup cost of a one-pass join
  current_setting('seq_page_cost')::real * 7
  AS startup,
278507.28 + -- total cost of a one-pass join
  current_setting('seq_page_cost')::real * 2 * (7 + 2309)
  AS total;
 startup |   total
---------+-----------
   45.13 | 283139.28
(1 row)
```

Thus, if there is not enough memory, the join is performed in two passes and becomes less efficient. Therefore, it is important to observe the following points:

- The query must be composed in a way that excludes redundant fields from the hash table.
- The planner must choose the smaller of the two sets of rows when building the hash table.

Using Hash Joins in Parallel Plans

v. 9.6

The hash join algorithm described above can also be used in parallel plans. First, several parallel processes build their own (absolutely identical) hash tables for the inner set, independently of each other; then they start processing the outer set concurrently. The performance gain here is due to each process scanning only its own share of outer rows.

The following plan uses a regular one-pass hash join:

```
=> SET work_mem = '128MB';

=> SET enable_parallel_hash = off;

=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT count(*)
FROM bookings b
  JOIN tickets t ON t.book_ref = b.book_ref;
```

```
                              QUERY PLAN
----------------------------------------------------------------------
 Finalize Aggregate (actual rows=1 loops=1)
   -> Gather (actual rows=3 loops=1)
       Workers Planned: 2
       Workers Launched: 2
       -> Partial Aggregate (actual rows=1 loops=3)
           -> Hash Join (actual rows=983286 loops=3)
               Hash Cond: (t.book_ref = b.book_ref)
               -> Parallel Index Only Scan using tickets_book_ref...
                   Heap Fetches: 0
               -> Hash (actual rows=2111110 loops=3)
                   Buckets: 4194304  Batches: 1  Memory Usage:
                   113172kB
                   -> Seq Scan on bookings b (actual rows=2111110...
(13 rows)
=> RESET enable_parallel_hash;
```

Here each process hashes the bookings table, then retrieves its own share of outer rows via the Parallel Index Only Scan node, and matches these rows against the resulting hash table.

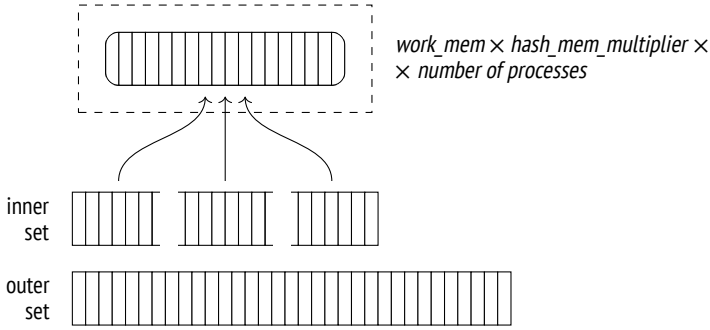
The hash table memory limit is applied to each parallel process separately, so the total size of memory allocated for this purpose will be three times bigger than indicated in the plan (Memory Usage).

v. 11 Parallel One-Pass Hash Joins

Even though a regular hash join can be quite efficient in parallel plans (especially for small inner sets, for which parallel processing does not make much sense), larger data sets are better handled by a special parallel hash join algorithm.

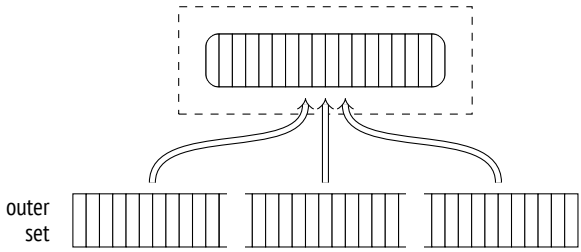
An important distinction of the parallel version of the algorithm is that the hash table is created in the *shared* memory, which is allocated dynamically and can be accessed by all parallel processes that contribute to the join operation. Instead of several separate hash tables, a single common one is built, which uses the total amount of memory dedicated to all the participating processes. It increases the chance of completing the join in one pass.

At the **first stage** (represented in the plan by the Parallel Hash node), all the parallel processes build a common hash table, taking advantage of the parallel access to the inner set.[1]



To move on from here, each parallel process must complete its share of first-stage processing.[2]

At the **second stage** (the Parallel Hash Join node), the processes are again run in parallel to match their shares of rows of the outer set against the hash table, which is already built by this time.[3]



Here is an example of such a plan:

```
=> SET work_mem = '64MB';
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT count(*)
FROM bookings b
  JOIN tickets t ON t.book_ref = b.book_ref;
```

[1] backend/executor/nodeHash.c, MultiExecParallelHash function
[2] backend/storage/ipc/barrier.c
[3] backend/executor/nodeHashjoin.c, ExecParallelHashJoin function

```
                              QUERY PLAN
----------------------------------------------------------------------
 Finalize Aggregate (actual rows=1 loops=1)
   -> Gather (actual rows=3 loops=1)
       Workers Planned: 2
       Workers Launched: 2
       -> Partial Aggregate (actual rows=1 loops=3)
           -> Parallel Hash Join (actual rows=983286 loops=3)
               Hash Cond: (t.book_ref = b.book_ref)
               -> Parallel Index Only Scan using tickets_book_ref...
                   Heap Fetches: 0
               -> Parallel Hash (actual rows=703703 loops=3)
                   Buckets: 4194304  Batches: 1  Memory Usage:
                   115392kB
                   -> Parallel Seq Scan on bookings b (actual row...
(13 rows)
=> RESET work_mem;
```

It is the same query that I showed in the previous section, but the parallel hash join
on was turned off by the *enable_parallel_hash* parameter at that time.

Although the available memory is down by half as compared to a regular hash join demonstrated before, the operation still completes in one pass because it uses the memory allocated for all the parallel processes (Memory Usage). The hash table gets a bit bigger, but since it is the only one we have now, the total memory usage has decreased.

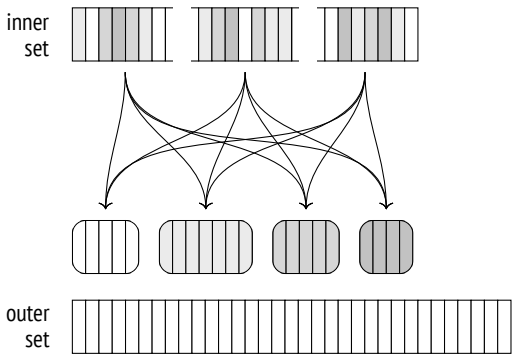
V. 11 Parallel Two-Pass Hash Joins

The consolidated memory of all the parallel processes may still be not enough to accommodate the whole hash table. It can become clear either at the planning stage or later, during query execution. The two-pass algorithm applied in this case is quite different from what we have seen so far.

The key distinction of this algorithm is that it creates several smaller hash tables instead of a single big one. Each process gets its own table and processes its own batches independently. (But since separate hash tables are still located in the shared memory, any process can get access to any of these tables.) If planning

shows that more than one batch will be required,[1] a separate hash table is built for each process right away. If the decision is taken at the execution stage, the hash table is rebuilt.[2]

Thus, at the **first stage** processes scan the inner set in parallel, splitting it into batches and writing them into temporary files.[3] Since each process reads only its own share of the inner set, none of them builds a full hash table for any of the batches (even for the first one). The full set of rows of any batch is only accumulated in the file written by all the parallel processes in a synchronized manner.[4] So unlike the non-parallel and one-pass parallel versions of the algorithm, the parallel two-pass hash join writes all the batches to disk, including the first one.



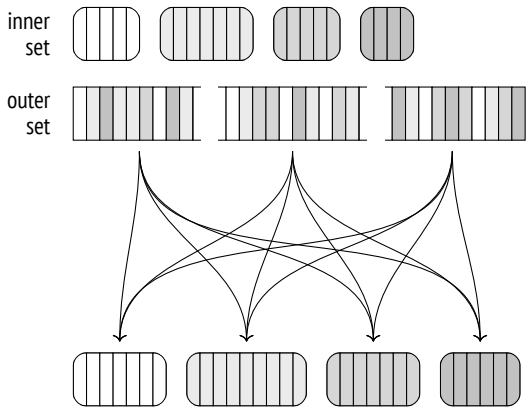
Once all the processes have completed hashing of the inner set, the **second stage** begins.[5]

If the non-parallel version of the algorithm were employed, the rows of the outer set that belong to the first batch would be matched against the hash table right away. But in the case of the parallel version, the memory does not contain the hash table yet, so the workers process the batches independently. Therefore, the second stage starts by a parallel scan of the outer set to distribute its rows into batches, and each batch is written into a separate temporary file.[6] The scanned

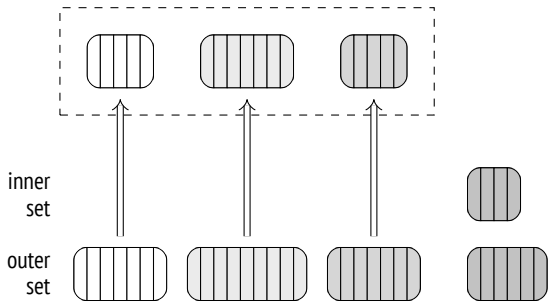
[1] backend/executor/nodeHash.c, ExecChooseHashTableSize function
[2] backend/executor/nodeHash.c, ExecParallelHashIncreaseNumBatches function
[3] backend/executor/nodeHash.c, MultiExecParallelHash function
[4] backend/utils/sort/sharedtuplestore.c
[5] backend/executor/nodeHashjoin.c, ExecParallelHashJoin function
[6] backend/executor/nodeHashjoin.c, ExecParallelHashJoinPartitionOuter function

rows are not inserted into the hash table (as it happens at the first stage), so the number of batches never rises.

Once all the processes have completed the scan of the outer set, we get $2N$ temporary files on disk; they contain the batches of the inner and outer sets.

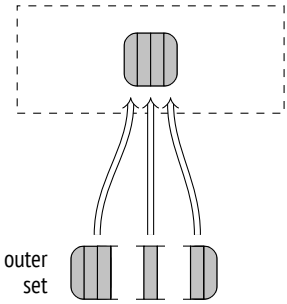


Then each process chooses one of the batches and performs the join: it loads the inner set of rows into a hash table in memory, scans the rows of the outer set, and matches them against the hash table. When the batch join is complete, the process chooses the next batch that has not been processed yet.[1]



If no more unprocessed batches are left, the process that has completed its own batch starts processing one of the batches that is currently being handled by another process; such concurrent processing is possible because all the hash tables are located in the shared memory.

[1] backend/executor/nodeHashjoin.c, ExecParallelHashJoinNewBatch function



This approach is more efficient than using a single big hash table for all the processes: it is easier to set up parallel processing, and synchronization is cheaper.

Modifications

The hash join algorithm supports any types of joins: apart from the inner join, it can also handle left, right, and full outer joins, as well as semi- and anti-joins. But as I have already mentioned, the join condition is limited to the equality operator.

We have already observed some of these operations when dealing with the nested loop join. Here is an example of the *right outer join*: *p. 411*

```
=> EXPLAIN (costs off) SELECT *
FROM bookings b
  LEFT OUTER JOIN tickets t ON t.book_ref = b.book_ref;
                QUERY PLAN
----------------------------------------
 Hash Right Join
   Hash Cond: (t.book_ref = b.book_ref)
   -> Seq Scan on tickets t
   -> Hash
       -> Seq Scan on bookings b
(5 rows)
```

Note that the logical left join specified in the SQL query got transformed into a physical operation of the right join in the execution plan.

At the logical level, bookings is the outer table (constituting the left side of the join operation), while the tickets table is the inner one. Therefore, bookings with no tickets must also be included into the join result.

At the physical level, inner and outer sets are assigned based on the cost of the join rather than their location in the query text. It usually means that the set with a smaller hash table will be used as the inner one. This is exactly what is happening here: the bookings table is used as the inner set, and the left join is changed to the right one.

And vice versa, if the query specifies the right outer join (to display the tickets that are not related to any bookings), the execution plan uses the left join:

```
=> EXPLAIN (costs off) SELECT *
FROM bookings b
  RIGHT OUTER JOIN tickets t ON t.book_ref = b.book_ref;
               QUERY PLAN
----------------------------------------
 Hash Left Join
   Hash Cond: (t.book_ref = b.book_ref)
   -> Seq Scan on tickets t
   -> Hash
       -> Seq Scan on bookings b
(5 rows)
```

To complete the picture, I will provide an example of a query plan with the full outer join:

```
=> EXPLAIN (costs off) SELECT *
FROM bookings b
  FULL OUTER JOIN tickets t ON t.book_ref = b.book_ref;
               QUERY PLAN
----------------------------------------
 Hash Full Join
   Hash Cond: (t.book_ref = b.book_ref)
   -> Seq Scan on tickets t
   -> Hash
       -> Seq Scan on bookings b
(5 rows)
```

Parallel hash joins are currently not supported for right and full joins.[1]

Note that the next example uses the bookings table as the outer set, but the planner would have preferred the right join if it were supported:

[1] commitfest.postgresql.org/33/2903

```
=> EXPLAIN (costs off) SELECT sum(b.total_amount)
FROM bookings b
  LEFT OUTER JOIN tickets t ON t.book_ref = b.book_ref;
                              QUERY PLAN
-----------------------------------------------------------------------
 Finalize Aggregate
   -> Gather
       Workers Planned: 2
       -> Partial Aggregate
           -> Parallel Hash Left Join
               Hash Cond: (b.book_ref = t.book_ref)
               -> Parallel Seq Scan on bookings b
               -> Parallel Hash
                   -> Parallel Index Only Scan using tickets_book...
(9 rows)
```

22.2 Distinct Values and Grouping

Algorithms that group values for aggregation and remove duplicates are very similar to join algorithms. One of the approaches they can use consists in building a hash table on the required columns. Values are included into the hash table only if it contains no such values yet. As a result, the hash table accumulates all the distinct values.

The node that performs hash aggregation is called HashAggregate.[1]

Let's consider some situations that may require this node.

The number of seats in each travel class (GROUP BY):

```
=> EXPLAIN (costs off) SELECT fare_conditions, count(*)
FROM seats
GROUP BY fare_conditions;
          QUERY PLAN
-----------------------------
 HashAggregate
   Group Key: fare_conditions
   -> Seq Scan on seats
(3 rows)
```

[1] backend/executor/nodeAgg.c

The list of travel classes (DISTINCT):

```
=> EXPLAIN (costs off) SELECT DISTINCT fare_conditions
FROM seats;
          QUERY PLAN
------------------------------
 HashAggregate
   Group Key: fare_conditions
   -> Seq Scan on seats
(3 rows)
```

Travel classes combined with one more value (UNION):

```
=> EXPLAIN (costs off) SELECT fare_conditions
FROM seats
UNION
SELECT NULL;
             QUERY PLAN
-----------------------------------
 HashAggregate
   Group Key: seats.fare_conditions
   -> Append
       -> Seq Scan on seats
       -> Result
(5 rows)
```

The Append node combines both sets but does not get rid of any duplicates, which must not appear in the UNION result. They have to be removed separately by the HashAggregate node.

4MB 1.0 The memory chunk allocated for the hash table is limited by the *work_mem* × × *hash_mem_multiplier* value, just like in the case of a hash join.

If the hash table fits the allocated memory, aggregation uses a single batch:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT DISTINCT amount FROM ticket_flights;
                          QUERY PLAN
----------------------------------------------------------------
 HashAggregate (actual rows=338 loops=1)
   Group Key: amount
   Batches: 1  Memory Usage: 61kB
   -> Seq Scan on ticket_flights (actual rows=8391852 loops=1)
(4 rows)
```

There are not so many distinct values in the amounts field, so the hash table takes only 61 kB (Memory Usage).

As soon as the hash table fills up the allocated memory, all the further values are spilled into temporary files and grouped into partitions based on several bits of their hash values. The number of partitions is a power of two and is chosen in such a way that each of their hash tables fits the allocated memory. The accuracy of the estimation is of course dependent on the quality of the collected statistics, so the received number is multiplied by 1.5 to further reduce partition sizes and raise the chances of processing each partition in one pass.[1] v. 13

Once the whole set is scanned, the node returns aggregation results for those values that have made it into the hash table.

Then the hash table is cleared, and each of the partitions saved into temporary files at the previous stage is scanned and processed just like any other set of rows. If the hash table still exceeds the allocated memory, the rows that are subject to overflow will be partitioned again and written to disk for further processing.

To avoid excessive I/O, the two-pass hash join algorithm moves MCVs into the first batch. Aggregation, however, does not require this optimization: those rows that fit the allocated memory will not be split into partitions, and MCVs are likely to occur early enough to get into RAM.

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT DISTINCT flight_id FROM ticket_flights;
                          QUERY PLAN
----------------------------------------------------------------
 HashAggregate (actual rows=150588 loops=1)
   Group Key: flight_id
   Batches: 5  Memory Usage: 4145kB  Disk Usage: 98184kB
   -> Seq Scan on ticket_flights (actual rows=8391852 loops=1)
(4 rows)
```

In this example, the number of distinct IDs is relatively high, so the hash table does not fit the allocated memory. It takes five batches to perform the query: one for the initial data set and four for the partitions written to disk.

[1] backend/executor/nodeAgg.c, hash_choose_num_partitions function

23

Sorting and Merging

23.1 Merge Joins

A merge join processes data sets sorted by the join key and returns the result that is sorted in a similar way. Input sets may come pre-sorted following an index scan; otherwise, the executor has to sort them before the actual merge begins.[1]

Merging Sorted Sets

Let's take a look at an example of a merge join; it is represented in the execution plan by the Merge Join node:[2]

```
=> EXPLAIN (costs off) SELECT *
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
ORDER BY t.ticket_no;
                           QUERY PLAN
-----------------------------------------------------------------
 Merge Join
   Merge Cond: (t.ticket_no = tf.ticket_no)
   -> Index Scan using tickets_pkey on tickets t
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
(4 rows)
```

The optimizer prefers this join method because it returns a sorted result, as defined by the ORDER BY clause. When choosing a plan, the optimizer notes the sort order of the data sets and does not perform any sorting unless it is really required. For

[1] backend/optimizer/path/joinpath.c, generate_mergejoin_paths function
[2] backend/executor/nodeMergejoin.c

example, if the data set produced by a merge join already has an appropriate sort order, it can be used in the subsequent merge join as is:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets t
  JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
  JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no
                         AND bp.flight_id = tf.flight_id
ORDER BY t.ticket_no;
                              QUERY PLAN
----------------------------------------------------------------------
 Merge Join
   Merge Cond: (tf.ticket_no = t.ticket_no)
   -> Merge Join
       Merge Cond: ((tf.ticket_no = bp.ticket_no) AND (tf.flight_...
       -> Index Scan using ticket_flights_pkey on ticket_flights tf
       -> Index Scan using boarding_passes_pkey on boarding_passe...
   -> Index Scan using tickets_pkey on tickets t
(7 rows)
```

The first tables to be joined are ticket_flights and boarding_passes; both of them have a composite primary key (ticket_no, flight_id), and the result is sorted by these two columns. The produced set of rows is then joined with the tickets table, which is sorted by the ticket_no column.

The join requires only one pass over both data sets and does not take any additional memory. It uses two pointers to the current rows (which are originally the first ones) of the inner and outer sets.

If the keys of the current rows do not match, one of the pointers (that references the row with the smaller key) is going to be advanced to the next row until it finds a match. The joined rows are returned to the upper node, and the pointer of the inner set is advanced by one place. The operation continues until one of the sets is over.

This algorithm copes with duplicates of the inner set, but the outer set can contain them too. Therefore, the algorithm has to be improved: if the key remains the same after the outer pointer is advanced, the inner pointer gets back to the first matching row. Thus, each row of the outer set will be matched to all the rows of the inner set with the same key.[1]

[1] backend/executor/nodeMergejoin.c, ExecMergeJoin function

For the outer join, the algorithm is further tweaked a bit, but it is still based on the same principle.

Merge join conditions can use only the equality operator, which means that only equi-joins are supported (although support for other condition types is currently under way too).[1]

Cost estimation. Let's take a closer look at the previous example:

```
=> EXPLAIN SELECT *
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
ORDER BY t.ticket_no;
                           QUERY PLAN
-----------------------------------------------------------------
 Merge Join  (cost=0.99..822334.21 rows=8391852 width=136)
   Merge Cond: (t.ticket_no = tf.ticket_no)
   -> Index Scan using tickets_pkey on tickets t
       (cost=0.43..139110.29 rows=2949857 width=104)
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
       (cost=0.56..570951.13 rows=8391852 width=32)
(6 rows)
```

The startup cost of the join includes at least the startup costs of all the child nodes.

In general, it may be required to scan some fraction of the outer or inner set before the first match is found. It is possible to estimate this fraction by comparing (based on the histogram) the smallest join keys in the two sets.[2] But in this particular case, the range of ticket numbers is the same in both tables.

p. 318

The total cost comprises the cost of fetching the data from the child nodes and the computation cost.

Since the join algorithm stops as soon as one of the sets is over (unless the outer join is performed, of course), the other set may be scanned only partially. To estimate the size of the scanned part, we can compare the maximal key values in the two sets. In this example, both sets will be read in full, so the total cost of the join includes the sum of the total costs of both child nodes.

[1] For example, see commitfest.postgresql.org/33/3160
[2] backend/utils/adt/selfuncs.c, mergejoinscansel function

Moreover, if there are any duplicates, some of the rows of the inner set may be scanned several times. The estimated number of repeat scans equals the difference between the cardinalities of the join result and the inner set.[1] In this query, these cardinalities are the same, which means that the sets contain no duplicates.

The algorithm compares join keys of the two sets. The cost of one comparison is estimated at the *cpu_operator_cost* value, while the estimated number of comparisons can be taken as the sum of rows of both sets (increased by the number of repeat reads caused by duplicates). The processing cost of each row included into the result is estimated at the *cpu_tuple_cost* value, as usual. 0.0025 0.01

Thus, in this example the cost of the join is estimated as follows:[2]

```
=> SELECT 0.43 + 0.56 AS startup,
  round((
    139110.29 + 570951.13 +
    current_setting('cpu_tuple_cost')::real * 8391852 +
    current_setting('cpu_operator_cost')::real * (2949857 + 8391852)
  )::numeric, 2) AS total;
 startup |   total
---------+-----------
    0.99 | 822334.21
(1 row)
```

Parallel Mode

v. 9.6

Although the merge join has no parallel flavor, it can still be used in parallel plans.[3]

The outer set can be scanned by several workers in parallel, but the inner set is always scanned by each worker in full.

Since the parallel hash join is almost always cheaper, I will turn it off for a while: *p. 430*

```
=> SET enable_hashjoin = off;
```

Here is an example of a parallel plan that uses a merge join:

[1] backend/optimizer/path/costsize.c, final_cost_mergejoin function
[2] backend/optimizer/path/costsize.c, initial_cost_mergejoin & final_cost_mergejoin functions
[3] backend/optimizer/path/joinpath.c, consider_parallel_mergejoin function

```
=> EXPLAIN (costs off)
SELECT count(*), sum(tf.amount)
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no;
                              QUERY PLAN
-----------------------------------------------------------------------
 Finalize Aggregate
   -> Gather
       Workers Planned: 2
       -> Partial Aggregate
           -> Merge Join
               Merge Cond: (tf.ticket_no = t.ticket_no)
               -> Parallel Index Scan using ticket_flights_pkey o...
               -> Index Only Scan using tickets_pkey on tickets t
(8 rows)
```

Full and right outer merge joins are not allowed in parallel plans.

Modifications

The merge join algorithm can be used with any types of joins. The only restriction is that join conditions of full and right outer joins must contain merge-compatible expressions (“*outer-column equals inner-column*” or “*column equals constant*”).[1] Inner and left outer joins simply filter the join result by irrelevant conditions, but for full and right joins such filtering is inapplicable.

Here is an example of a full join that uses the merge algorithm:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets t
  FULL JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
ORDER BY t.ticket_no;
                              QUERY PLAN
-----------------------------------------------------------------------
 Sort
   Sort Key: t.ticket_no
   -> Merge Full Join
       Merge Cond: (t.ticket_no = tf.ticket_no)
       -> Index Scan using tickets_pkey on tickets t
       -> Index Scan using ticket_flights_pkey on ticket_flights tf
(6 rows)
```

[1] backend/optimizer/path/joinpath.c, select_mergejoin_clauses function

Inner and left merge joins preserve the sort order. Full and right outer joins, however, cannot guarantee it because NULL values can be wedged in between the ordered values of the outer set, which breaks the sort order.[1] To restore the required order, the planner introduces the Sort node here. Naturally, it increases the cost of the plan, making the hash join more attractive, so the planner has selected this plan only because hash joins are currently disabled.

But the next example cannot do without a hash join: the nested loop does not allow full joins at all, while merging cannot be used because of an unsupported join condition. So the hash join is used regardless of the *enable_hashjoin* parameter value:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets t
  FULL JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
                              AND tf.amount > 0
ORDER BY t.ticket_no;
                  QUERY PLAN
-----------------------------------------------
 Sort
   Sort Key: t.ticket_no
   -> Hash Full Join
       Hash Cond: (tf.ticket_no = t.ticket_no)
       Join Filter: (tf.amount > '0'::numeric)
       -> Seq Scan on ticket_flights tf
       -> Hash
           -> Seq Scan on tickets t
(8 rows)
```

Let's restore the ability to use hash joins that we have previously disabled:

```
=> RESET enable_hashjoin;
```

23.2 Sorting

If one of the sets (or possibly both of them) is not sorted by the join key, it must be reordered before the join operation begins. This sorting operation is represented in the plan by the Sort node:[2]

[1] backend/optimizer/path/pathkeys.c, build_join_pathkeys function
[2] backend/executor/nodeSort.c

```
=> EXPLAIN (costs off)
SELECT * FROM flights f
  JOIN airports_data dep ON f.departure_airport = dep.airport_code
ORDER BY dep.airport_code;
                        QUERY PLAN
-------------------------------------------------------
 Merge Join
   Merge Cond: (f.departure_airport = dep.airport_code)
   -> Sort
       Sort Key: f.departure_airport
       -> Seq Scan on flights f
   -> Sort
       Sort Key: dep.airport_code
       -> Seq Scan on airports_data dep
(8 rows)
```

Such sorting can also be applied outside the context of joins if the ORDER BY clause is specified, both in a regular query and within a window function:

```
=> EXPLAIN (costs off)
SELECT flight_id,
  row_number() OVER (PARTITION BY flight_no ORDER BY flight_id)
FROM flights f;
              QUERY PLAN
-------------------------------------
 WindowAgg
   -> Sort
       Sort Key: flight_no, flight_id
       -> Seq Scan on flights f
(4 rows)
```

Here the WindowAgg node[1] computes a window function on the data set that has been pre-sorted by the Sort node.

The planner has several sort methods in its toolbox. The example that I have already shown uses two of them (Sort Method). These details can be displayed by the EXPLAIN ANALYZE command, as usual:

```
=> EXPLAIN (analyze,costs off,timing off,summary off)
SELECT * FROM flights f
  JOIN airports_data dep ON f.departure_airport = dep.airport_code
ORDER BY dep.airport_code;
```

[1] backend/executor/nodeWindowAgg.c

```
                              QUERY PLAN
-------------------------------------------------------------------
 Merge Join (actual rows=214867 loops=1)
   Merge Cond: (f.departure_airport = dep.airport_code)
   -> Sort (actual rows=214867 loops=1)
       Sort Key: f.departure_airport
       Sort Method: external merge  Disk: 17136kB
       -> Seq Scan on flights f (actual rows=214867 loops=1)
   -> Sort (actual rows=104 loops=1)
       Sort Key: dep.airport_code
       Sort Method: quicksort  Memory: 52kB
       -> Seq Scan on airports_data dep (actual rows=104 loops=1)
(10 rows)
```

Quicksort

If the data set to be sorted fits the *work_mem* chunk, the classic *quicksort* method is applied. This algorithm is described in all textbooks, so I am not going to explain it here. 4MB

As for the implementation, sorting is performed by a dedicated component[1] that chooses the most suitable algorithm depending on the amount of available memory and some other factors.

Cost estimation. Let's take a look at how a small table is sorted. In this case, sorting is performed in memory using the quicksort algorithm:

```
=> EXPLAIN SELECT *
FROM airports_data
ORDER BY airport_code;
                              QUERY PLAN
----------------------------------------------------------------------
 Sort  (cost=7.52..7.78 rows=104 width=145)
   Sort Key: airport_code
   -> Seq Scan on airports_data  (cost=0.00..4.04 rows=104 width=...
(3 rows)
```

[1] backend/utils/sort/tuplesort.c

The computational complexity of sorting n values is known to be $O(n \log_2 n)$. A
0.0025 single comparison operation is estimated at the doubled *cpu_operator_cost* value. Since the *whole* data set must be scanned and sorted before the result can be retrieved, the startup cost of sorting includes the total cost of the child node and all the expenses incurred by comparison operations.

The total cost of sorting also includes the cost of processing each row to be returned, which is estimated at *cpu_operator_cost* (and not at the usual *cpu_tuple_cost* value, as the overhead incurred by the Sort node is insignificant).[1]

For this example, the costs are calculated as follows:

```
=> WITH costs(startup) AS (
  SELECT 4.04 + round((
    current_setting('cpu_operator_cost')::real * 2 *
      104 * log(2, 104)
  )::numeric, 2)
)
SELECT startup,
  startup + round((
    current_setting('cpu_operator_cost')::real * 104
  )::numeric, 2) AS total
FROM costs;
 startup | total
---------+-------
    7.52 |  7.78
(1 row)
```

Top-N Heapsort

If a data set needs to be sorted only partially (as defined by the LIMIT clause), the *heapsort* method can be applied (it is represented in the plan as top-N heapsort). To be more exact, this algorithm is used if sorting reduces the number of rows at least by half, or if the allocated memory cannot accommodate the whole input set (while the output set fits it).

```
=> EXPLAIN (analyze, timing off, summary off)
SELECT * FROM seats
ORDER BY seat_no LIMIT 100;
```

[1] backend/optimizer/path/costsize.c, cost_sort function

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Limit  (cost=72.57..72.82 rows=100 width=15)
   (actual rows=100 loops=1)
   -> Sort  (cost=72.57..75.91 rows=1339 width=15)
       (actual rows=100 loops=1)
       Sort Key: seat_no
       Sort Method: top-N heapsort  Memory: 33kB
       -> Seq Scan on seats  (cost=0.00..21.39 rows=1339 width=15)
           (actual rows=1339 loops=1)
(8 rows)
```

To find k highest (or lowest) values out of n, the executor adds the first k rows into a data structure called heap. Then the rest of the rows get added one by one, and the smallest (or largest) value is removed from the heap after each iteration. Once all the rows are processed, the heap contains k sought-after values.

> The heap term here denotes a well-known data structure and has nothing to do with database tables, which are often referred to by the same name.

Cost estimation. The computational complexity of the algorithm is estimated at $O(n \log_2 k)$, but each particular operation is more expensive as compared to the quicksort algorithm. Therefore, the formula uses $n \log_2 2k$.[1]

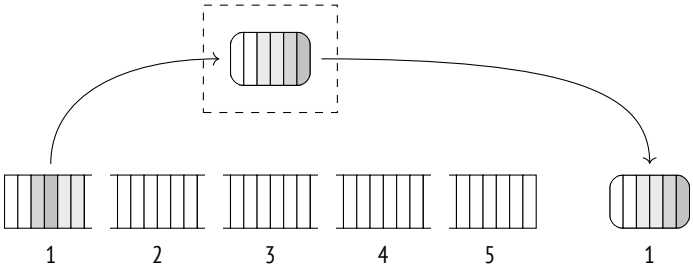
```
=> WITH costs(startup)
AS (
  SELECT 21.39 + round((
    current_setting('cpu_operator_cost')::real * 2 *
      1339 * log(2, 2 * 100)
  )::numeric, 2)
)
SELECT startup,
  startup + round((
    current_setting('cpu_operator_cost')::real * 100
  )::numeric, 2) AS total
FROM costs;
 startup | total
---------+-------
   72.57 | 72.82
(1 row)
```

[1] backend/optimizer/path/costsize.c, cost_sort function

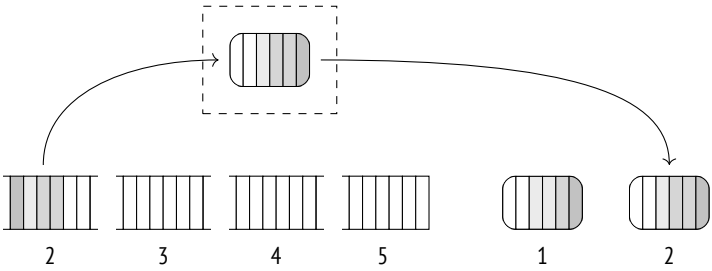
External Sorting

If the scan shows that the data set is too big to be sorted in memory, the sorting node switches over to *external merge sorting* (labeled as external merge in the plan).

The rows that are already scanned are sorted in memory by the quicksort algorithm and written into a temporary file.



Subsequent rows are then read into the freed memory, and this procedure is repeated until all the data is written into several pre-sorted files.



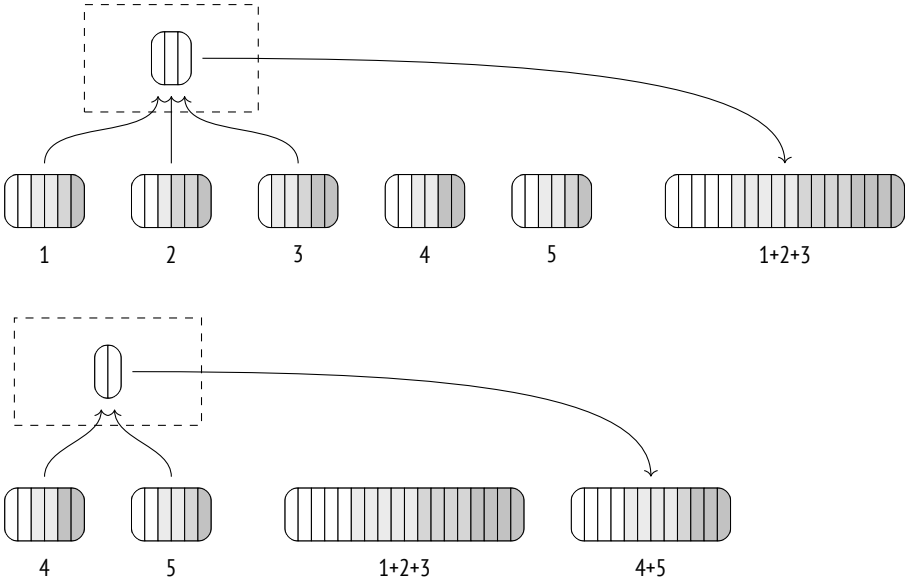
Next, these files are merged into one. This operation is performed by roughly the same algorithm that is used for merge joins; the main difference is that it can process more than two files at a time.

A merge operation does not need too much memory. In fact, it is enough to have room for one row per file. The first rows are read from each file, the row with the lowest value (or the highest one, depending on the sort order) is returned as a partial result, and the freed memory is filled with the next row fetched from the same file.

In practice, rows are read in batches of 32 pages rather than one by one, which reduces the number of I/O operations. The number of files that are merged in a single iteration depends on the available memory, but it is never smaller than six. The upper boundary is also limited (by 500) since efficiency suffers when there are too many files.[1]

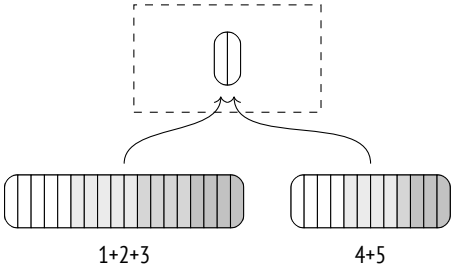
> Sorting algorithms have long-established terminology. External sorting was originally performed using magnetic tapes, and PostgreSQL keeps a similar name for the component that controls temporary files.[2] Partially sorted data sets are called "runs."[3] The number of runs participating in the merge is referred to as the "merge order." I did not use these terms, but they are worth knowing if you want to understand PostgreSQL code and comments.

If the sorted temporary files cannot be merged all at once, they have to be processed in several passes, their partial results being written into new temporary files. Each iteration increases the volume of data to be read and written, so the more RAM is available, the faster the external sorting completes.



[1] backend/utils/sort/tuplesort.c, tuplesort_merge_order function
[2] backend/utils/sort/logtape.c
[3] *Donald E. Knuth*. The Art of Computer Programming. Volume III. Sorting and Searching

The next iteration merges newly created temporary files.



The final merge is typically deferred and performed on the fly when the upper node pulls the data.

Let's run the EXPLAIN ANALYZE command to see how much disk space has been used by external sorting. The BUFFERS option displays buffer usage statistics for temporary files (temp read and written). The number of written buffers will be (roughly) the same as the number of read ones; converted to kilobytes, this value is shown as Disk in the plan:

```
=> EXPLAIN (analyze, buffers, costs off, timing off, summary off)
SELECT * FROM flights
ORDER BY scheduled_departure;
                        QUERY PLAN
-----------------------------------------------------------
 Sort (actual rows=214867 loops=1)
   Sort Key: scheduled_departure
   Sort Method: external merge  Disk: 17136kB
   Buffers: shared hit=2627, temp read=2142 written=2150
   -> Seq Scan on flights (actual rows=214867 loops=1)
       Buffers: shared hit=2624
(6 rows)
```

To print more details on using temporary files into the server log, you can enable the *log_temp_files* parameter.

Cost estimation. Let's take the same plan with external sorting as an example:

```
=> EXPLAIN SELECT *
FROM flights
ORDER BY scheduled_departure;
```

```
                              QUERY PLAN
----------------------------------------------------------------------
 Sort  (cost=31883.96..32421.12 rows=214867 width=63)
   Sort Key: scheduled_departure
   -> Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=63)
(3 rows)
```

Here the regular cost of comparisons (their number is the same as in the case of a quicksort operation in memory) is extended by the I/O cost.[1] All the input data has to be first written into temporary files on disk and then read from disk during the merge operation (possibly more than once if all the created files cannot be merged in one iteration).

It is assumed that three quarters of disk operations (both reads and writes) are sequential, while one quarter is random.

The volume of data written to disk depends on the number of rows to be sorted and the number of columns used in the query.[2] In this example, the query displays all the columns of the flights table, so the size of the data spilled to disk is almost the same as the size of the whole table if its tuple and page metadata are not taken into account (2309 pages instead of 2624).

Here sorting is completed in one iteration.

Therefore, the sorting cost is estimated in this plan as follows:

```
=> WITH costs(startup) AS (
  SELECT 4772.67 + round((
    current_setting('cpu_operator_cost')::real * 2 *
      214867 * log(2, 214867) +
    (current_setting('seq_page_cost')::real * 0.75 +
     current_setting('random_page_cost')::real * 0.25) *
    2 * 2309 * 1 -- one iteration
  )::numeric, 2)
)
SELECT startup,
  startup + round((
    current_setting('cpu_operator_cost')::real * 214867
  )::numeric, 2) AS total
FROM costs;
```

[1] backend/optimizer/path/costsize.c, cost_sort function
[2] backend/optimizer/path/costsize.c, relation_byte_size function

```
 startup  |  total
----------+----------
 31883.96 | 32421.13
(1 row)
```

v. 13 Incremental Sorting

If a data set has to be sorted by keys $K_1 \dots K_m \dots K_n$, and this data set is known to be already sorted by the first m keys, you do not have to re-sort it from scratch. Instead, you can split this set into groups by the same first keys $K_1 \dots K_m$ (values in these groups already follow the defined order), and then sort each of these groups separately by the remaining $K_{m+1} \dots K_n$ keys. This method is called the *incremental sort*.

Incremental sorting is less memory-intensive than other sorting algorithms, as it splits the set into several smaller groups; besides, it allows the executor to start returning results after the first group is processed, without waiting for the whole set to be sorted.

In PostgreSQL, the implementation is a bit more subtle:[1] while relatively big groups of rows are processed separately, smaller groups are combined together and are sorted in full. It reduces the overhead incurred by invoking the sorting procedure.[2]

The execution plan represents incremental sorting by the Incremental Sort node:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM bookings
ORDER BY total_amount, book_date;
                              QUERY PLAN
-----------------------------------------------------------------
 Incremental Sort (actual rows=2111110 loops=1)
   Sort Key: total_amount, book_date
   Presorted Key: total_amount
   Full-sort Groups: 2823  Sort Method: quicksort  Average
   Memory: 30kB  Peak Memory: 30kB
   Pre-sorted Groups: 2624  Sort Method: quicksort  Average
```

[1] backend/executor/nodeIncrementalSort.c
[2] backend/utils/sort/tuplesort.c

```
   Memory: 3152kB  Peak Memory: 3259kB
   -> Index Scan using bookings_total_amount_idx on bookings (ac...
(8 rows)
```

As the plan shows, the data set is pre-sorted by the total_amount field, as it is the result of an index scan run on this column (Presorted Key). The EXPLAIN ANALYZE command also displays run-time statistics. The Full-sort Groups row is related to small groups that were united to be sorted in full, while the Presorted Groups row displays the data on large groups with partially ordered data, which required incremental sorting by the book_date column only. In both cases, the in-memory quicksort method was applied. The difference in group sizes is due to non-uniform distribution of booking costs.

Incremental sorting can be used to compute window functions too: v. 14

```
=> EXPLAIN (costs off)
SELECT row_number() OVER (ORDER BY total_amount, book_date)
FROM bookings;
                           QUERY PLAN
-----------------------------------------------------------------
 WindowAgg
   -> Incremental Sort
       Sort Key: total_amount, book_date
       Presorted Key: total_amount
       -> Index Scan using bookings_total_amount_idx on bookings
(5 rows)
```

Cost estimation. Cost calculations for incremental sorting[1] are based on the expected number of groups[2] and the estimated sorting cost of an average-sized group (which we have already reviewed).

The startup cost reflects the cost estimation of sorting the first group, which allows the node to start returning sorted rows; the total cost includes the sorting cost of all groups.

We are not going to explore these calculations any further here.

[1] backend/optimizer/path/costsize.c, cost_incremental_sort function
[2] backend/utils/adt/selfuncs.c, estimate_num_groups function

v. 10 Parallel Mode

Sorting can also be performed concurrently. But although parallel workers do pre-sort their data shares, the Gather node knows nothing about their sort order and can only accumulate them on a first-come, first-serve basis. To preserve the sort order, the executor has to apply the Gather Merge node.[1]

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM flights
ORDER BY scheduled_departure
LIMIT 10;
                            QUERY PLAN
-------------------------------------------------------------------
 Limit (actual rows=10 loops=1)
   -> Gather Merge (actual rows=10 loops=1)
       Workers Planned: 1
       Workers Launched: 1
       -> Sort (actual rows=10 loops=2)
           Sort Key: scheduled_departure
           Sort Method: top-N heapsort  Memory: 27kB
           Worker 0:  Sort Method: top-N heapsort  Memory: 27kB
           -> Parallel Seq Scan on flights (actual rows=107434 lo...
(9 rows)
```

The Gather Merge node uses a binary heap[2] to adjust the order of rows fetched by several workers. It virtually merges several sorted sets of rows, just like external sorting would do, but is designed for a different use case: Gather Merge typically handles a small fixed number of data sources and fetches rows one by one rather than block by block.

Cost estimation. The startup cost of the Gather Merge node is based on the startup
p. 343 cost of its child node. Just like for the Gather node, this value is increased by the
1000 cost of launching parallel processes (estimated at *parallel_setup_cost*).

[1] backend/executor/nodeGatherMerge.c
[2] backend/lib/binaryheap.c

The received value is then further extended by the cost of building a binary heap, which requires sorting n values, where n is the number of parallel workers (that is, $n \log_2 n$). A single comparison operation is estimated at doubled *cpu_operator_cost*, 0.0025
and total share of such operations is typically negligible since n is quite small.

The total cost includes the expenses incurred by fetching all the data by several processes that perform the parallel part of the plan, and the cost of transferring this data to the leader. A single row transfer is estimated at *parallel_tuple_cost* increased 0.1
by 5 %, to compensate for possible waits on getting the next values.

The expenses incurred by binary heap updates must also be taken into account in total cost calculations: each input row requires $\log_2 n$ comparison operations and certain additional actions (they are estimated at *cpu_operator_cost*).[1]

Let's take a look at yet another plan that uses the Gather Merge node. Note that the workers here first perform partial aggregation by hashing, and then the Sort node *p. 437*
sorts the received results (it is cheap because few rows are left after aggregation) to be passed further to the leader process, which gathers the full result in the Gather Merge node. As for the final aggregation, it is performed on the sorted list of values:

```
=> EXPLAIN SELECT amount, count(*)
FROM ticket_flights
GROUP BY amount;
                              QUERY PLAN
----------------------------------------------------------------------
 Finalize GroupAggregate  (cost=123399.62..123485.00 rows=337 wid...
   Group Key: amount
   -> Gather Merge  (cost=123399.62..123478.26 rows=674 width=14)
       Workers Planned: 2
       -> Sort  (cost=122399.59..122400.44 rows=337 width=14)
           Sort Key: amount
           -> Partial HashAggregate  (cost=122382.07..122385.44 r...
               Group Key: amount
               -> Parallel Seq Scan on ticket_flights  (cost=0.00...
(9 rows)
```

Here we have three parallel processes (including the leader), and the cost of the Gather Merge node is calculated as follows:

[1] backend/optimizer/path/costsize.c, cost_gather_merge function

```
=> WITH costs(startup, run) AS (
  SELECT round((
    -- launching processes
    current_setting('parallel_setup_cost')::real +
    -- building the heap
    current_setting('cpu_operator_cost')::real * 2 * 3 * log(2, 3)
  )::numeric, 2),
  round((
    -- passing rows
    current_setting('parallel_tuple_cost')::real * 1.05 * 674 +
    -- updating the heap
    current_setting('cpu_operator_cost')::real * 2 * 674 * log(2, 3) +
    current_setting('cpu_operator_cost')::real * 674
  )::numeric, 2)
)
SELECT 122399.59 + startup AS startup,
  122400.44 + startup + run AS total
FROM costs;
  startup  |   total
-----------+-----------
 123399.61 | 123478.26
(1 row)
```

23.3 Distinct Values and Grouping

As we have just seen, grouping values to perform aggregation (and to eliminate duplicates) can be performed not only by hashing, but also by sorting. In a sorted list, groups of duplicate values can be singled out in one pass.

Retrieval of distinct values from a sorted list is represented in the plan by a very simple node called Unique[1]:

```
=> EXPLAIN (costs off) SELECT DISTINCT book_ref
FROM bookings
ORDER BY book_ref;
                        QUERY PLAN
-----------------------------------------------------------
 Result
   -> Unique
       -> Index Only Scan using bookings_pkey on bookings
(3 rows)
```

[1] backend/executor/nodeUnique.c

Aggregation is performed in the GroupAggregate node:[1]

```
=> EXPLAIN (costs off) SELECT book_ref, count(*)
FROM bookings
GROUP BY book_ref
ORDER BY book_ref;
                      QUERY PLAN
-------------------------------------------------------
 GroupAggregate
   Group Key: book_ref
   -> Index Only Scan using bookings_pkey on bookings
(3 rows)
```

In parallel plans, this node is called Partial GroupAggregate, while the node that completes aggregation is called Finalize GroupAggregate.

Both hashing and sorting strategies can be combined in a single node if grouping is performed by several column sets (specified in the GROUPING SETS, CUBE, or ROLLUP clauses). Without getting into rather complex details of this algorithm, I will simply provide an example that performs grouping by three different columns in conditions of scarce memory: v. 10

```
=> SET work_mem = '64kB';
=> EXPLAIN (costs off) SELECT count(*)
FROM flights
GROUP BY GROUPING SETS (aircraft_code, flight_no, departure_airport);
          QUERY PLAN
-------------------------------
 MixedAggregate
   Hash Key: departure_airport
   Group Key: aircraft_code
   Sort Key: flight_no
     Group Key: flight_no
   -> Sort
       Sort Key: aircraft_code
       -> Seq Scan on flights
(8 rows)
=> RESET work_mem;
```

Here is what happens while this query is being executed. The aggregation node, which is shown in the plan as MixedAggregate, receives the data set sorted by the aircraft_code column.

[1] backend/executor/nodeAgg.c, agg_retrieve_direct function

First, this set is scanned, and the values are grouped by the aircraft_code column (Group Key). As the scan progresses, the rows are reordered by the flight_no column (like it is done by a regular Sort node: either via the quicksort method if the memory is sufficient, or using external sorting on disk); at the same time, the executor places these rows into a hash table that uses departure_airport as its key (like it is done by hash aggregation: either in memory, or using temporary files).

At the second stage, the executor scans the data set that has just been sorted by the flight_no column and groups the values by the same column (Sort Key and the nested Group Key node). If the rows had to be grouped by yet another column, they would be resorted again as required.

Finally, the hash table prepared at the first stage is scanned, and the values are grouped by the departure_airport column (Hash Key).

23.4 Comparison of Join Methods

As we have seen, two data sets can be joined using three different methods, and each of them has its own pros and cons.

The *nested loop join* does not have any prerequisites and can start returning the first rows of the result set right away. It is the only join method that does not have to fully scan the inner set (as long as index access is available for it). These properties make the nested loop algorithm (combined with indexes) an ideal choice for short OLTP queries, which deal with rather small sets of rows.

The weak point of the nested loop becomes apparent as the data volume grows. For a Cartesian product, this algorithm has quadratic complexity—the cost is proportionate to the product of sizes of the data sets being joined. However, the Cartesian product is not so common in practice; for each row of the outer set, the executor typically accesses a certain number of rows of the inner set using an index, and this average number does not depend on the total size of the data set (for example, an average number of tickets in a booking does not change as the number of bookings and bought tickets grows). Thus, the complexity of the nested loop algorithm often shows linear growth rather than quadratic one, even if with a high linear coefficient.

An important distinction of the nested loop algorithm is its universal applicability: it supports all join conditions, whereas other methods can only deal with equi-joins. It allows running queries with any types of conditions (except for the full join, which cannot be used with the nested loop), but you must keep in mind that a non-equi-join of a large data set is highly likely to be performed slower than desired.

A *hash join* works best on large data sets. If RAM is sufficient, it requires only one pass over two data sets, so its complexity is linear. Combined with sequential table scans, this algorithm is typically used for OLAP queries, which compute the result based on a large volume of data.

However, if the response time is more important than throughput, a hash join is not the best choice: it will not start returning the resulting rows until the whole hash table is built.

The hash join algorithm is only applicable to equi-joins. Another restriction is that the data type of the join key must support hashing (but almost all of them do).

The nested loop join can sometimes beat the hash join, taking advantage of caching the rows of the inner set in the Memoize node (which is also based on a hash table). While the hash join always scans the inner set in full, the nested loop algorithm does not have to, which may result in some cost reduction. v. 14

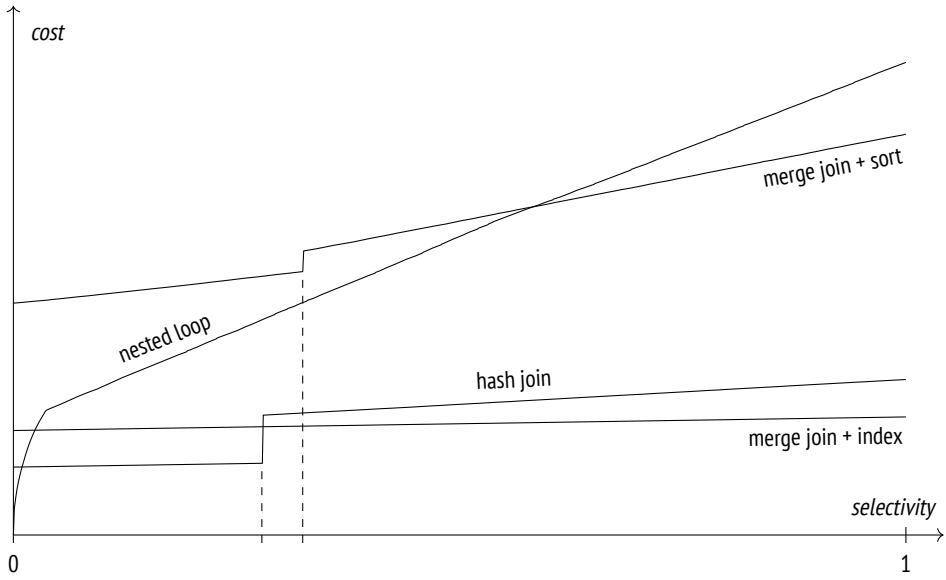
A *merge join* can perfectly handle both short OLTP queries and long OLAP ones. It has linear complexity (the sets to be joined have to be scanned only once), does not require much memory, and returns the results without any preprocessing; however, the data sets must already have the required sort order. The most cost-effective way to do it is to fetch the data via an index scan. It is a natural choice if the row count is low; for larger data sets, index scans can still be efficient, but only if the heap access is minimal or does not happen at all.

If no suitable indexes are available, the sets have to be sorted, but this operation is memory-intensive, and its complexity is higher than linear: $O(n \log_2 n)$. In this case, a hash join is almost always cheaper than a merge join—unless the result has to be sorted.

An added bonus of a merge join is the equivalence of the inner and outer sets. The efficiency of both nested loop and hash joins is highly dependent on whether the planner can assign inner and outer sets correctly.

Merge joins are limited to equi-joins. Besides, the data type must have a B-tree operator class.

The following graph illustrates approximate dependencies between the costs of various join methods and the fraction of rows to be joined.



If the selectivity is high, the nested loop join uses index access for both tables; then the planner switches to the full scan of the outer table, which is reflected by the linear part of the graph.

Here the hash join is using a full scan for both tables. The "step" on the graph corresponds to the moment when the hash table fills the whole memory and the batches start getting spilled to disk.

If an index scan is used, the cost of a merge join shows small linear growth. If the *work_mem* size is big enough, a hash join is usually more efficient, but a merge join beats it when it comes to temporary files.

The upper graph of the sort-merge join shows that the costs rise when indexes are unavailable and the data has to be sorted. Just like in the case of a hash join, the

"step" on the graph is caused by insufficient memory, as it leads to using temporary files for sorting.

It is merely an example; in each particular case the ratio between the costs will be different.

Index

A

Aborting transactions 82, 86, 89, 249, 269
Access method
  index 354, 415
  properties 364
  table 333
Aggregate 338–339
Aggregation 338, 343
  hashing 437, 457
  sorting 457
Alignment 73
Analysis 126, 309, 387
Anomaly
  dirty read 44, 46, 50
  lost update 46, 56, 58
  non-repeatable read 47, 52, 59
  phantom read 47, 59, 268
  read skew 54, 56, 60
  read-only transaction 63, 66, 268
  write skew 62, 65, 268
Append 438
"Asterisk," the reasons not to use it 35, 419, 453
Atomicity 45, 89
autoprewarm leader 185–187
autoprewarm worker 187
autovacuum 127
autovacuum launcher 127–129
autovacuum worker 128
autovacuum_analyze_scale_factor 131
autovacuum_analyze_threshold 131
autovacuum_enabled 119, 129
autovacuum_freeze_max_age 147, 152–153
autovacuum_freeze_min_age 153
autovacuum_freeze_table_age 153
autovacuum_max_workers 128, 137, 142
autovacuum_multixact_freeze_max_age 245
autovacuum_naptime 128–129
autovacuum_vacuum_cost_delay 137, 142, 153
autovacuum_vacuum_cost_limit 137, 142
autovacuum_vacuum_insert_scale_factor 130–131
autovacuum_vacuum_insert_threshold 130–131
autovacuum_vacuum_scale_factor 129–130
autovacuum_vacuum_threshold 129–130
autovacuum_work_mem 128
autovacuum_freeze_max_age 152

B

Backend 37

Background worker 125, 128, 345
Background writing 203
    setup 206
Batch processing 164, 255
bgwriter 203, 206–208, 222
bgwriter_delay 206
bgwriter_lru_maxpages 206, 208
bgwriter_lru_multiplier 206
Binding 303
Bison 288
Bitmap 386
    NULL values 73
Bitmap Heap Scan 328, 386, 390, 392
Bitmap Index Scan 328, 386, 390, 392, 395
BitmapAnd 389
Bloating 103, 117, 163, 337
Block *see* page
Buffer cache 36, 169, 190, 196, 275, 336, 355, 379
    configuration 182
    eviction 177
    local 187, 353
Buffer pin 171, 173, 276
Buffer ring 179, 336

C

Cardinality 298, 308, 379
    join 404
Cartesian product 397, 399
Checkpoint 196, 214
    monitoring 206
    setup 203
checkpoint_completion_target 203–204
checkpointer 196–197, 202, 204, 206–208, 214
checkpoint_timeout 204, 207
checkpoint_warning 206
CLOG 79, 153, 190, 193, 196
Cluster 21
Cmin and cmax 99
Collation 359
Combo-identifier 99
Commit 79, 193, 249
    asynchronous 210
    synchronous 210
commit_delay 210
commit_siblings 210
Consistency 43, 45
Correlated predicates 299, 328
Correlation 323, 374, 385
Cost 293, 297, 300
cpu_index_tuple_cost 376
cpu_operator_cost 338, 376, 421, 443, 448, 457
cpu_tuple_cost 337, 339, 377, 409, 421, 443, 448
CTE Scan 350–351
CTID 73, 110
Cursor 98, 174, 298, 306, 349
cursor_tuple_fraction 298, 306

D

Database 21
data_checksums 215
Deadlocks 230, 256, 265–266
deadlock_timeout 257, 265, 278
debug_print_parse 289
debug_print_plan 292
debug_print_rewritten 290
default_statistics_target 309, 317–318, 321, 332
default_table_access_method 333

default_transaction_isolation 68
Demo Database 285
Dirty read 46, 50
Durability 45

E
effective_cache_size 379–380
effective_io_concurrency 386
enable_bitmapscan 378
enable_hashjoin 443, 445
enable_memoize 410
enable_mergejoin 411
enable_parallel_hash 429, 432
enable_seqscan 260, 378
Equi-join 397, 435, 442
Eviction 177, 192, 203
Execution 300, 304
Execution plan 292
    generic and custom 304

F
fastupdate 266
fdatasync 214
fillfactor 106–107, 113–114, 145, 148, 156, 269
Finalize Aggregate 344
Finalize GroupAggregate 459
Flex 288
force_parallel_mode 349
Foreign keys 240, 242, 404
Fork 26
    free space map 28, 106, 118
    initialization 28
    main 27, 72
    visibility map 29, 106, 147–148, 161, 382
Freezing 144, 160, 175, 244
    manual 153
from_collapse_limit 294, 296
fsync 214
Full page image 200
full_page_writes 217, 219

G
Gather 340, 342–344, 350, 456
Gather Merge 456–457
geqo 296
geqo_threshold 296
Getting the result 306
GIN
    deferred update 265
GroupAggregate 459
Grouping 437, 458

H
Hash 417, 420, 425
Hash Join 417, 420, 425
Hash table 172, 275, 277, 408, 417
HashAggregate 437–438
hash_mem_multiplier 408, 418, 431, 438
Header
    page 70, 120
    row version 73
    tuple 239
Hint bits *see* information bits
Histogram 318
Horizon 100–101, 106, 121, 163, 383
HOT updates 110

I
idle_in_transaction_session_timeout 164
ignore_checksum_failure 216

Incremental Sort 454
Index 354, 360
    covering 368, 381, 384
    integrity constraint 366, 368
    multicolumn 367
    on expression 326, 361
    ordering 365, 370
    partial 372
    pruning 116
    statistics 326
    unique 240, 366, 368
    versioning 84
Index Only Scan 381
Index Scan 373–375, 378, 404, 406
Indexing engine 355, 364
Information bits 73, 77, 80, 93, 217, 239
InitPlan 314, 352
Instance 21
Integrity constraints 43
Isolation 45
    snapshot 49, 65, 92, 239

J

Join
    anti- and semi- 398, 412
    cost estimation 400, 406, 409, 420, 427, 442, 447, 449, 452, 455–456
    different methods 460
    hashing 417, 422
    inner 397
    merging 440
    nested loop 398
    order 292, 294, 419, 442
    outer 397, 411, 442
    parallel hash 430, 432
    parameterized 403
join_collapse_limit 294–296

L

Locks 48, 227, 355
    advisory 266
    escalation 239, 271
    heavyweight 229, 240
    lightweight 275
    memory 171
    no waits 164, 254
    non-relation 263
    page 265
    predicate 268
    queue 235, 245, 251
    relation 126, 157, 162, 222, 232
    relation extension 265
    row 165, 239
    spinlocks 274
    tranche 276
    transaction ID 231
    tuple 245
lock_timeout 255–256
log_autovacuum_min_duration 141
log_checkpoints 206
logical 220, 224
log_lock_waits 278
log_temp_files 425, 452
Lost update 46, 56, 58

M

maintenance_io_concurrency 387
maintenance_work_mem 124, 138, 141
Map
    free space 28, 106, 118
    freeze 29, 147, 150, 161

visibility 29, 106, 147–148, 161, 382
Materialization 350, 400, 407
Materialize 400, 402–403, 407–409, 411
max_connections 230, 271
max_locks_per_transaction 230
max_parallel_processes 185
max_parallel_workers 345
max_parallel_workers_per_gather 345–347
max_pred_locks_per_page 271
max_pred_locks_per_relation 272
max_pred_locks_per_transaction 271–272
max_wal_senders 220
max_wal_size 204, 207
max_worker_processes 128, 345
Memoize 407–410, 461
Merge 440, 450, 456
Merge Join 440
minimal 214, 220, 222–223
min_parallel_index_scan_size 125
min_parallel_table_scan_size 346
min_wal_size 205
MixedAggregate 459
Multitransactions 243
wraparound 244
Multiversion concurrency control 50, 72, 117

N

Nearest neighbor search 370
Nested Loop 293, 398–399, 404, 409
Nested Loop Anti Join 413
Nested Loop Left Join 398, 411
Nested Loop Semi Join 414
Non-repeatable read 47, 52, 59
Non-uniform distribution 315, 425
NULL 73, 312, 371

O

OID 22
old_snapshot_threshold 164
Operator class 357, 415
support functions 362
Operator family 362
Optimization *see* planning

P

Page 30
dirty 170
fragmentation 73, 108
full image 200
header 155, 161
prefetching 386
split 116
pageinspect 70, 74, 78, 84, 146, 192, 241
Parallel Bitmap Heap Scan 395
Parallel execution 340, 345, 393, 415, 429, 443, 456, 459
limitations 348
Parallel Hash 431
Parallel Hash Join 431
Parallel Index Only Scan 430
Parallel Seq Scan 341–342
parallel_leader_participation 340, 342
parallel_setup_cost 343, 456
parallel_tuple_cost 343, 457
parallel_workers 346
Parsing 288
Partial Aggregate 343

Partial GroupAggregate 459
pgbench 212, 217, 280
pg_buffercache 171, 183
pg_checksums 215
pg_controldata 199
PGDATA 21
pg_dump 104
pg_prewarm 185
pg_prewarm.autoprewarm 185
pg_prewarm.autoprewarm_interval 185
pg_rewind 191
pgrowlocks 244, 261
pgstattuple 157–158
pg_test_fsync 214
pg_visibility 120, 147
pg_wait_sampling 280
pg_wait_sampling.profile_period 281
pg_waldump 195, 202, 221
Phantom read 47, 59, 268
plan_cache_mode 306
Planning 292, 304
Pointers to tuples 72
Portal 300
postgres 35
postmaster 35–37, 128, 199, 201, 340
Preparing a statement 302
ProcArray 80, 94
Process 35
Protocol 38
    extended query 302
    simple query 288
Pruning 106, 113, 116
psql 15, 18, 22, 90–91, 279, 285

R

random_page_cost 337, 378, 390
Read Committed 47, 49–52, 54, 56, 59–60, 68–69, 92, 100, 102, 104, 121, 249
Read skew 54, 56, 60
Read Uncommitted 46–47, 49–50
Read-only transaction anomaly 63, 66, 268
Recheck 355, 373, 388
Recovery 199
Relation 25
Repeatable Read 47, 49–50, 59–60, 62–63, 65, 67–69, 92, 101, 104, 154, 249, 269
replica 220, 222–224
Rewriting *see* transformation
Row version *see* tuple
RTE 289
Rule system 290

S

Savepoint 86
Scan
    bitmap 369, 385
    cost estimation 336, 341, 374, 382, 389
    index 270, 369, 373
    index-only 310, 371, 381
    method comparison 395
    parallel index 393
    parallel sequential 341
    sequential 269, 335
Schema 23
search_path 23
Segment 26, 194
Selectivity 298, 336
    join 404
Seq Scan 293, 336, 338–339, 352

seq_page_cost 337, 378, 390, 428
Serializable 48–49, 65, 67–69, 92, 101, 104, 249, 268–269, 272, 349
Server 21
shared_buffers 182
shared_preload_libraries 185, 280
slowfs 281
Snapshot 92, 95, 222
  export 104
  system catalog 103
Sort 445–446, 448, 457, 460
Sorting 370, 440, 445
  external 450
  heapsort 448
  incremental 454
  parallel 456
  quicksort 447
Special space 71
startup 199–201
Starvation 245, 251
statement_timeout 256
Statistics 126, 298
  basic 308, 382
  correlation 323, 375
  distinct values 313, 329
  expression 324, 332
  extended 325
  field width 323
  histogram 318, 442
  most common values 315, 331, 405, 425
  multivariate 327
  non-scalar data types 322
  NULL fraction 312
SubPlan 351–352
Subtransaction 86, 196
Support functions 362
Synchronization 210, 214
synchronous_commit 209–211
System catalog 22, 222, 289

T

Tablespace 24
temp_buffers 188
temp_file_limit 188, 423
Tid Scan 374
Timeline 194
TOAST 23, 30, 85, 180
track_commit_timestamp 94
track_counts 127
track_io_timing 177
Transaction 44, 76, 92
  abort 82, 86, 89, 249, 269
  age 143
  commit 79, 193, 210, 249
  ID lock 231
  status 94, 193
  subtransaction 86, 196
  virtual 85, 231
Transaction ID
  wraparound 143, 151
Transformation 289
Tree
  parse 288
  plan 292
Truncation 126
Tuple 72
  insert only 125, 130
Tuple ID 72, 354
Tuple pointer 108

U
Unique 458

V
Vacuum 101, 174, 309, 355, 383
    aggressive 149
    autovacuum 127, 257
    full 156
    monitoring 138, 159
    phases 124
    routine 118
vacuum_cost_delay 136, 153
vacuum_cost_limit 136–137
vacuum_cost_page_dirty 136
vacuum_cost_page_hit 136
vacuum_cost_page_miss 136
vacuum_failsafe_age 147, 153
vacuum_freeze_min_age 147–148, 150, 154
vacuum_freeze_table_age 147, 149–150
vacuum_index_cleanup 154
vacuum_multixact_failsafe_age 245
vacuum_multixact_freeze_min_age 245
vacuum_multixact_freeze_table_age 245
vacuum_truncate 126
vacuum_freeze_min_age 148
Virtual transaction 85
Visibility 93, 98, 335, 355, 373, 382
Volatility 55, 362, 372

W
Wait-for graph 256
Waits 278
    sampling 280
    unaccounted-for time 279, 281
WAL *see* write-ahead log277
wal_buffers 191
wal_compression 217
wal_keep_size 206
wal_level 220
wal_log_hints 217
wal_recycle 205
wal_segment_size 194
walsender 209, 220
wal_skip_threshold 220–221
wal_sync_method 214
walwriter 210–211
wal_writer_delay 210–211
wal_writer_flush_after 211
WindowAgg 446
work_mem 17, 301, 387–389, 391, 400, 408, 418, 425, 429, 431, 438, 447, 459
Write skew 62, 65, 268
Write-ahead log 37, 189, 334, 355
    levels 219

X
Xmin and xmax 73, 75, 79, 81, 93, 143, 239, 244