Les bases de données relationnelles et leurs systèmes de gestion

Jacques Le Maitre

Département d'Informatique
UFR des Sciences et Techniques
Université de Toulon et du Var

Sommaire

Généralités **1**
Qu'est ce qu'une base de données ? 2
Le modèle ANSI-SPARC 3
Architecture d'un SGBD 9
Langage de bases de données 11
Transactions 13
Contrôles 14
Indépendance données-traitements 15
Qui intervient sur une BD ? 16
Concevoir une BD 17

Modèle relationnel **19**
Introduction 20
Constitution d'une BD relationnelle 21
Domaines 22
Relations 23
Intégrité 28
Exemple 29

Langages relationnels **35**
Introduction 36
Algèbre relationnelle 37
Calcul relationnel 50
Manipulation des valeurs nulles 55
Vues 56

Conception d'un schéma relationnel **57**
Introduction 58
Normalisation utilisant les dépendances fonctionnelles 59
Normalisation utilisant les dépendances multivaluées 66
Propriétés d'une décomposition 69
Décomposition d'un schéma relationnel en 4e forme normale 70

SQL **73**
Introduction 74
Définition des données 75
Manipulation des données 81
Index 97
Vues 98
SQL dans un programme 101

Maintien de l'intégrité **107**
Expression de l'intégrité 108
Assertions 109
Déclencheurs 110

Confidentialité **113**
Objectifs et moyens 114
Protection par les vues 115
Attribution des autorisations en SQL 116

Organisation physique **119**
Introduction 120
Fichiers 121
Gestion des n-uplets 122
Echange disque - mémoire centrale 126
Index 129

Arbres B^+ 132
Hachage statique 144

Résolution des requêtes 155
Introduction 156
Résolution des opérateurs relationnels 157
Arbre de requête 161
Propriétés des opérateurs relationnels 162
Choix d'un plan d'exécution 165
Descente des sélections et projections 166
Ordres de jointure 171
Recherche du meilleur plan d'exécution 172

Concurrence et reprise 173
Concept de transaction 174
Propriétés d'une transaction 176
Problémes dus à la concurrence 177
Exécution sérialisable 182
Le verrouillage 185
Verrouillage à deux phases 188
Reprise après panne 196
Outils pour la reprise 197
Etude de cas 200
Reprise à partir du journal 202

Bibliographie 206

Généralités

Qu’est ce qu’une base de données ?

Une **base de données** (BD en abrégé) est un ensemble d’informations archivées dans des mémoires accessibles à des ordinateurs en vue de permettre le traitement des diverses applications prévues pour elles.

L’intérêt d’une BD est de regrouper les données communes à une application dans le but :

- d’éviter les redondances et les incohérences qu’entraînerait fatalement une approche où les données seraient réparties dans différents fichiers sans connexions entre eux,
- d’offrir des langages de haut niveau pour la définition et la manipulation des données,
- de partager les données entre plusieurs utilisateurs,
- de contrôler l’intégrité, la sécurité et la confidentialité des données,
- d’assurer l’indépendance entre les données et les traitements.

Les bases de données sont gérées par des logiciels spécialisés appelés **systèmes de gestion de bases de données** (SGBD en abrégé).

Le modèle ANSI-SPARC

En 1975 le comité SPARC de l'ANSI a proposé une architecture à trois niveaux qui est devenue classique :

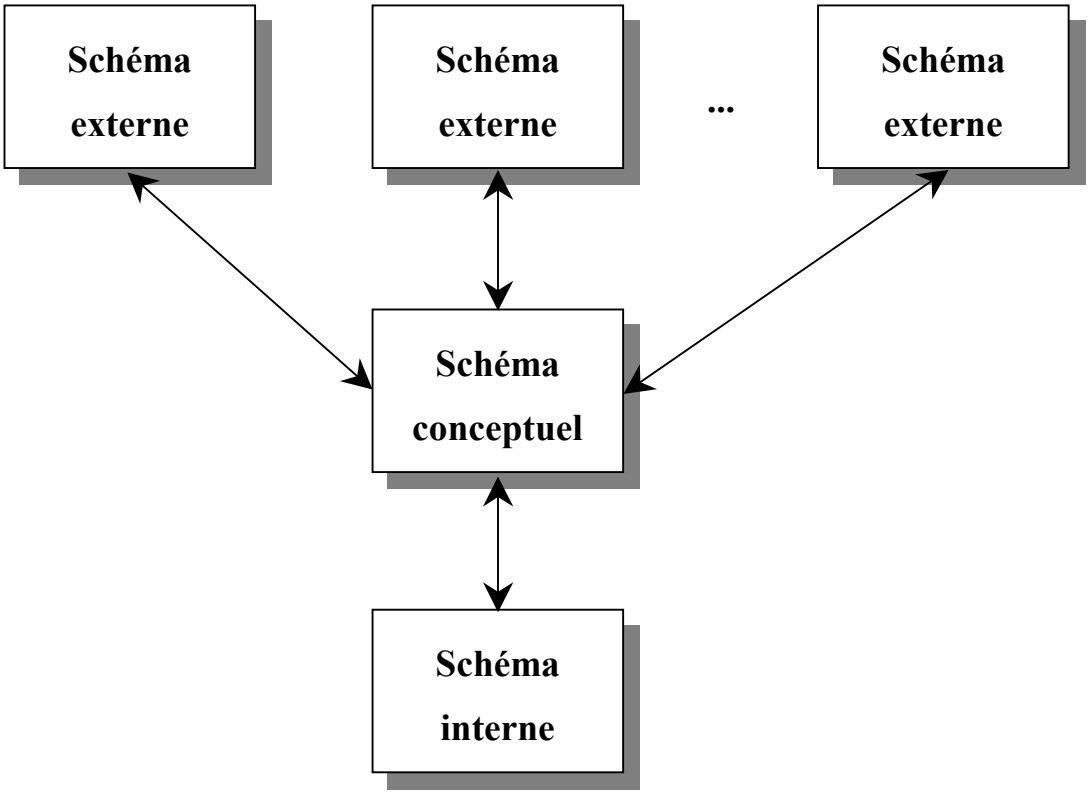


Schéma conceptuel

Rôle

Le schéma conceptuel est une représentation du monde réel auquel se rapporte la BD.

Principaux concepts

- entité (ou objet) : une personne, un livre.
- propriété (ou attribut) : titre d'un livre, adresse d'une personne.
- association : personne auteur d'un livre.
- agrégat : une adresse composée d'une rue et d'un code postal.
- collection : un ensemble de personnes, une liste de prénoms.

Principaux modèles conceptuels

- 1ère génération :
  - modèle hiérarchique (IMS d'IBM)
  - modèle réseau (DBTG CODASYL)
- 2e génération :
  - modèle relationnel
- 3e génération :
  - modèle orienté objet.
  - modèle objet / relationnel

Exemples

Soit une BD concernant les livres d'une bibliothèque et leurs auteurs. On suppose qu'un livre est identifié par sa cote et un auteur par son nom.

Modèle relationnel

Schéma

```
Livre(Cote: Chaîne, Titre: Chaîne)
Auteur(Nom: Chaîne, Cote: Chaîne)
Personne(Nom: Chaîne, Prénom: Chaîne, Année_naissance: Entier)
```

Instance

Livre	
Cote	**Titre**
BD/46	Les BD en BD

Personne		
Nom	**Prénom**	**Année_naissance**
Dupont	Jean	1960
Durand	Pierre	1953

Auteur	
Nom	**Cote**
Dupont	BD/46
Durand	BD/46

Modèle orienté-objet

Schéma

```
classe Livre
    attribut Cote: Chaîne
    attribut Titre: Chaîne
    attribut Auteurs: liste(Personne)

classe Personne
    attribut Nom: Chaîne
    attribut Prénom: Chaîne
    attribut Année_naissance: Entier
    méthode Age(): Entier
        {année_courante - self->Année_naissance}

livres: ensemble(Livre)
personnes: ensemble(Personne)
```

Instances

```
P1: {Nom = "Dupont", Prenom = "Jean", Année_naissance = 1960}
P2: {Nom = "Durand", Prenom = "Pierre", Année_naissance = 1953}
L1: {Cote = "BD/46", Titre = "Les BD en BD", Auteurs = [P1,P2]}

livres = {L1}
personnes = {P1, P2}
```

Schéma externe

Un **schéma externe** représente la façon dont un utilisateur final ou un programme d'application voit la partie de la BD qui le concerne.

Il existe en général plusieurs modèles externes pour une même BD.

Le schéma conceptuel d'une BD peut être complexe. Les schémas externes donnent aux utilisateurs une vision plus simple de ce schéma.

Les schémas externes permettent aussi de protéger la BD contre des manipulations incorrectes ou non autorisées, en cachant certaines données à certains utilisateurs.

Schéma interne (ou physique)

Rôle

Le schéma interne décrit l'organisation des données en mémoire secondaire (sur disque).

L'organisation choisie doit permettre :

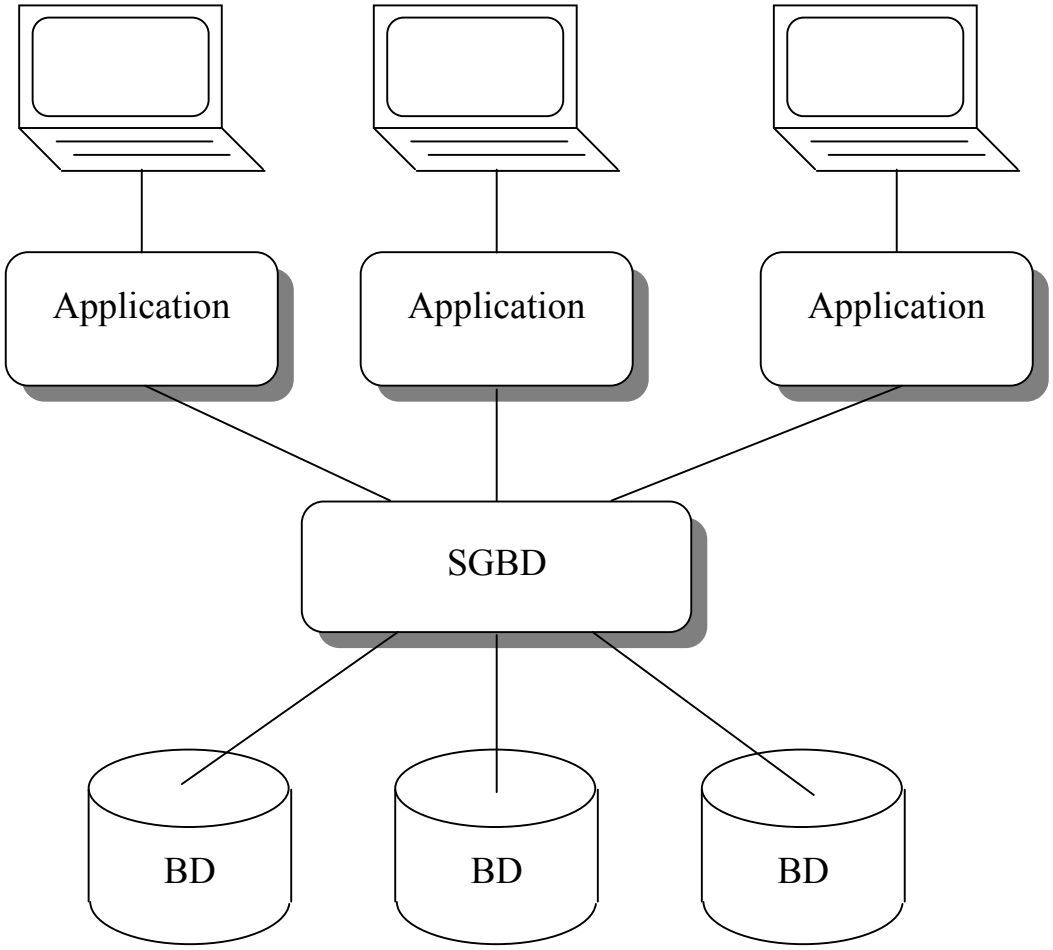
- d'accéder le plus rapidement possible à un ensemble de données vérifiant certaines conditions,
- de créer, modifier ou supprimer des données avec une réorganisation minimale et une utilisation optimale de la place disponible.

Quelques organisations de données

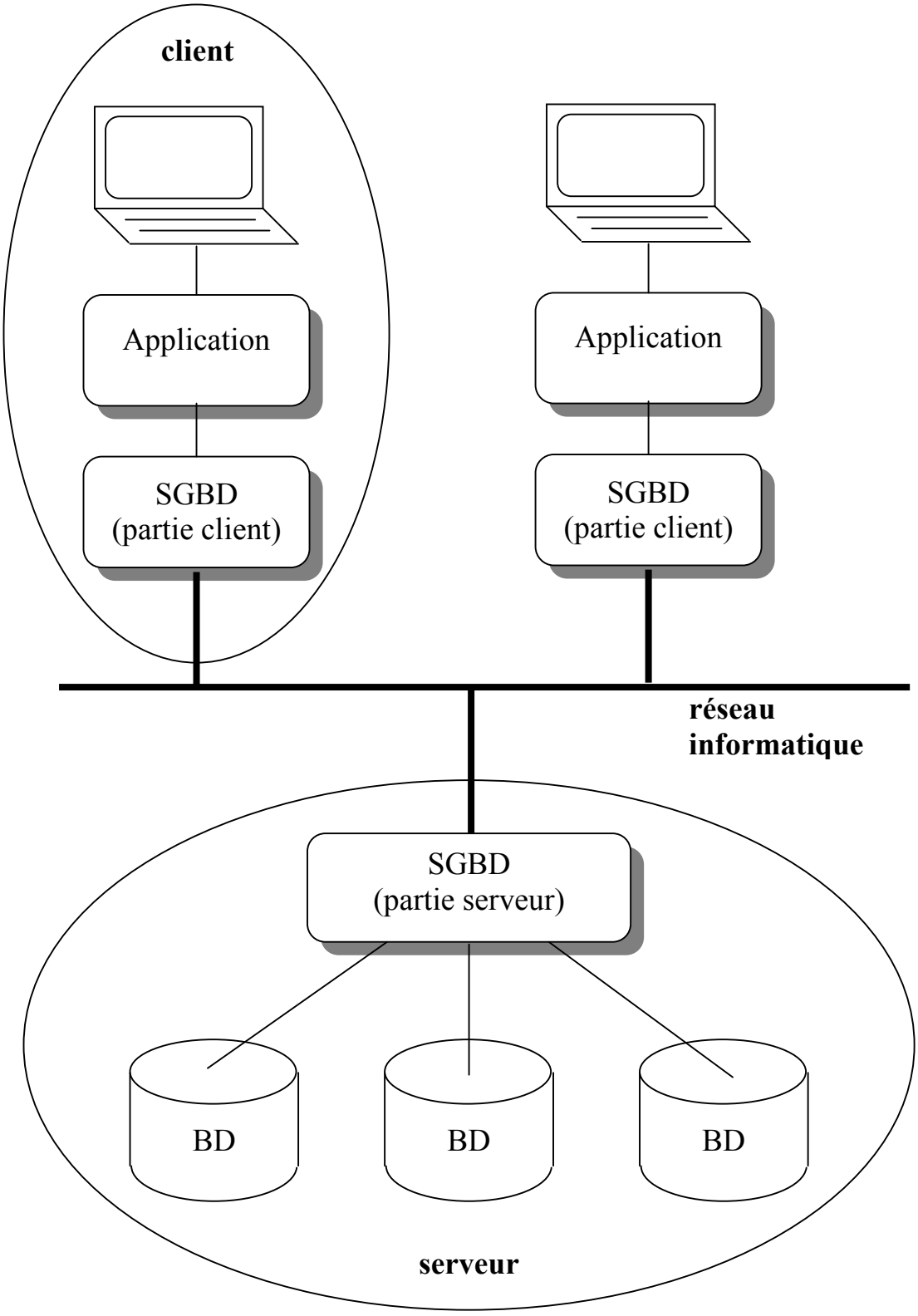
- séquentielle,
- chaînée (pointeurs),
- indexée.

Architecture d'un SGBD

Architecture centralisée



Architecture client-serveur



Langage de bases de données

C'est au travers d'un langage de haut niveau appelé **langage de bases de données** que sont réalisées la création, l'interrogation et la manipulation d'une BD. Le plus connu et le plus utilisé est de ces langages est SQL.

Le cœur d'un langage de base de données est constitué par le **langage de requêtes** qui est utilisé pour spécifier les données recherchées.

Fonctions

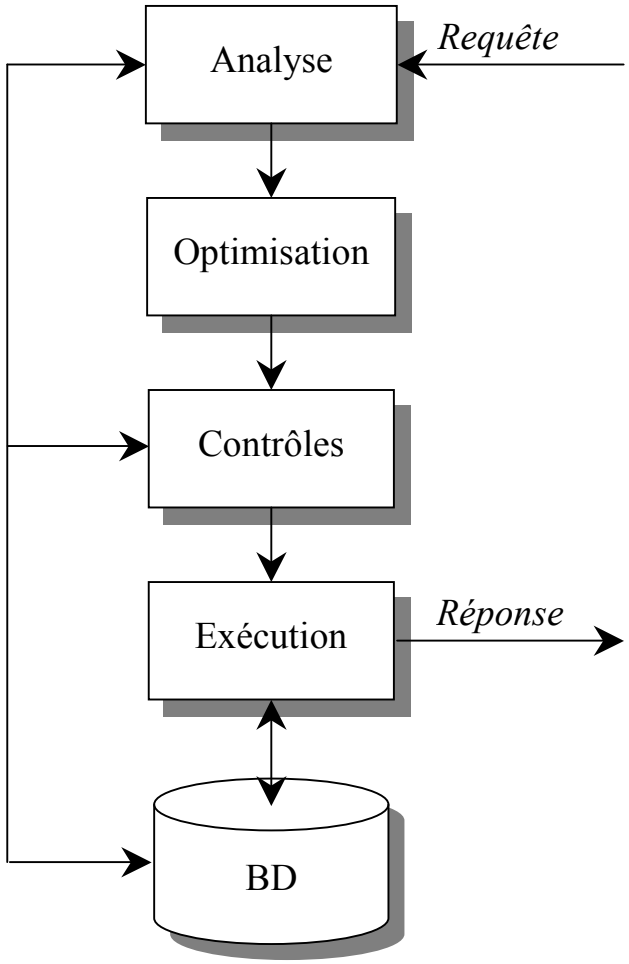
- Définition des données : schéma conceptuel, schémas externes, chemins d'accès aux données.
- Manipulation des données :
  - création, modification et suppression,
  - mises en place d'index pour aaméliorer les temps de réponse,
  - interrogation,
  - gestion des droits d'accès,

  …

Modes d'utilisation

- **intégré** dans un langage hôte (Cobol, C, Java, …),
- **autonome**.

Exécution d'une requête



Transactions

Pour un SGBD l'unité de traitement est la **transaction**.

Une transaction est un fragment de programme qui fait passer une BD d'un état cohérent à un autre état cohérent, par une suite d'actions élémentaires.

Un exemple classique de transaction est l'opération qui transfère une somme S d'un compte bancaire A à un compte bancaire B.

```
début transaction
solde(A) = solde(A) - S
solde(B) = solde(B) + S
fin transaction
```

Il est clair que cette opération ne doit pas être interrompue entre le débit de A et le crédit de B.

Contrôles

Intégrité

Les données stockées dans une BD doivent respecter un certain nombre de contraintes dites d'**intégrité**. Un SGBD doit assurer que les contraintes d'intégrité sont respectées.

Concurrence

En général plusieurs utilisateurs se partagent la même BD. Plusieurs transactions peuvent donc s'exécuter en même temps. Un SGBD doit assurer que les éventuels conflits entre ces transactions ne mettent pas la BD dans état incohérent.

Sécurité

Après une panne, qu'elle soit d'origine logicielle ou matérielle, un SGBD doit être capable de restaurer la BD dans un état cohérent, le même ou le plus proche de celui précédant la panne.

Confidentialité

Un SGBD doit permettre d'interdire à certaines personnes de réaliser certaines opérations sur une partie ou sur toute la BD.

Indépendance données-traitements

L'**indépendance données-traitements** est indispensable pour pouvoir faire évoluer facilement l'organisation logique ou physique d'une BD ou bien l'architecture matérielle du SGBD qui la gère.

L'indépendance données-traitements permet si elle est atteinte :

- de modifier l'organisation physique (par exemple ajouter un index pour un accès plus rapide) sans modifier le schéma conceptuel ou les programmes d'applications,
- de modifier le schéma conceptuel (par exemple ajouter un nouveau type d'entité ou d'association) sans modifier les programmes d'applications.

On parle aussi d'**indépendance logique** et d'**indépendance physique**.

Qui intervient sur une BD ?

- L'**administrateur** (une personne ou une équipe) :
  Il définit le schéma conceptuel de la BD et le fait évoluer.
  Il fixe les paramètres de l'organisation physique de façon à optimiser les performances.
  Il gère les droits d'accès et les mécanismes de sécurité.
- Les **programmeurs d'applications** :
  Ils définisent les schémas externes et construisent les programmes qui alimentent ou exploitent la BD en vue d'applications particulières.
  Ils utilisent pour cela le langage de bases de données du SGBD, éventuellement couplé avec un langage de programmation classique.
- Les **utilisateurs finals** :
  Ils accèdent à la BD au travers des outils construits par les programmeurs d'applications ou pour les plus avertis au travers du langage de requêtes.

Concevoir une BD

La **conception** d'une BD est une opération difficile. Elle nécessite une analyse approfondie du monde réel ainsi que des besoins des futurs utilisateurs.

Des méthodes de conception de BD ont donc été développées : UML, par exemple. Ces méthodes sont basées sur des modèles conceptuels de haut niveau, indépendants d'un SGBD particulier : orienté-objet, par exemple.

Une fois le schéma conceptuel de haut-niveau établi, il est traduit en termes du modèle conceptuel du SGBD choisi.

Modèle relationnel

Introduction

Créé par E.F. Codd d'IBM en 1969, le modèle relationnel a fait l'objet d'un grand nombre de travaux de recherche qui, depuis le début des années 80, ont débouché sur des produits commerciaux :

- DB2 d'IBM,
- Oracle,
- Informix,
- Sybase,
- Access ou SQL-Server de Microsoft,

…

Le succès du modèle relationnel est dû à :

- sa **simplicité** pour l'utilisateur : une BD est vue comme un ensemble de tables,
- ses **fondements théoriques** : l'algèbre relationnelle et la logique des prédicats

Constitution d'une BD relationnelle

Une **BD relationnelle** est constituée par :

- un ensemble de domaines,
- un ensemble de relations,
- un ensemble de contraintes d'intégrité.

Domaines

Un **domaine** est un ensemble de valeurs atomiques.

On distingue :

- les domaines **prédéfinis** :
  - chaînes de caractères,
  - entiers,
  - réels,
  - booléens,
  - date

  …
- les domaines **définis** :
  - en **extension**, c'est à dire en énumérant les valeurs. Par exemple :

    ```
    Couleur = {"rouge", "vert", "bleu", "jaune"}
    ```
  - en **intension**, c'est à dire en donnant la formule que doit vérifier chaque valeur, par exemple :

    ```
    Mois = {m| m ∈ Entier et 1 ≤ m ≤ 12}
    ```

Deux domaines sont dits **compatibles** si leurs valeurs sont sémantiquement comparables. Par exemple, les domaines :

```
Taille = {t | Réel t et t > 0}
Poids = {p | Réel p et p > 0}
```

ne sont pas compatibles car des mètres et des grammes ne sont pas comparables.

Relations

Une relation R est un sous-ensemble du produit cartésien de n domaines $D_1, \ldots, D_n$:

$$R \subseteq D_1 \times \ldots \times D_n$$

Une relation est définie par son nom, par son type et par son extension.

Type d'une relation

Le **type d'une relation** est une expression de la forme :

$$rel(A_1: D_1, \ldots, A_n: D_n)$$

où chaque D_i est un domaine et chaque A_i est un **nom d'attribut** qui indique le rôle du domaine D_i dans la relation. Par exemple :

```
rel(Nom: Chaîne, Age: Entier, Marié: Booléen)
```

est le type d'une relation construite sur les domaines Chaîne, Entier et Booléen et dont le premier représente un nom, le second un âge et le troisième le fait d'être marié ou non..

Extension d'une relation

L'**extension d'une relation** de type $rel(A_1: D_1, \ldots, A_n: D_n)$ est un ensemble de n-uplets :

$$\{A_1 = v_1, \ldots, A_n = v_n\}$$

tels que $v_1 \in D_1, \ldots, v_n \in D_n$.

L'extension d'une relation est variable au cours de la vie de la base de données.

Par exemple :

```
{{Nom = "Dupont", Age = 36, Marié = Vrai}
 {Nom = "Durand", Age = 22, Marié = Faux}}
```

est une extension de la relation de type :

```
rel(Nom: Chaîne, Age: Entier, Marié: Booléen)
```

Schéma d'une relation

Nous appellerons **schéma d'une relation** l'expression :

$R(A_1: D_1, \ldots, A_n: D_n)$

qui désigne une relation de nom R et de type rel(A_1: D_1, …, A_n: D_n). Par exemple :

```
Personne(Nom: Chaîne, Age: Entier, Marié: Booléen)
```

Lorsque l'indication des domaines n'est pas requise, un schéma de relation peut se réduire à l'expression :

$R(A_1, \ldots, A_n)$

Deux visions d'une relation

On suppose donnée la relation de schéma :

```
Personne(Nom: Chaîne, Age: Entier, Marié: Booléen)
```

et d'extension :

```
{{Nom = "Dupont", Age = 36, Marié = Vrai}
 {Nom = "Durand", Age = 22, Marié = Faux}}
```

Vision tabulaire

L'extension d'une relation de schéma $R(A_1: D_1, \ldots, A_n: D_n)$ peut être vue comme une table de nom R possédant n colonnes nommées $A_1, \ldots, A_n$ et dont chaque ligne représente un n-uplet de cette extension. Par exemple :

Personne		
Nom	**Age**	**Marié**
Dupont	36	Vrai
Durand	22	Faux

Vision assertionnelle

A toute relation de schéma $R(A_1: D_1, \ldots, A_n: D_n)$ il est associé un prédicat R tel que l'assertion R t est vraie si le n-uplet t appartient à l'extension de R et fausse sinon.

Par exemple, l'assertion :

```
Personne {Nom = "Dupont", Age = 36, Marié = Vrai}
```

est vraie.

Valeurs nulles

Il peut arriver que certaines informations soient inconnues ou non pertinentes. Par exemple, une conférence dont la date n'est pas encore fixée ou bien le nombre de couleurs pour un écran noir et blanc.

Pour représenter une telle absence d'information on utilise une valeur particulière : la **valeur nulle** que nous noterons `Nulle`.

Par exemple :

```
{Titre = "L'avenir des bases de données",
 Conférencier = "Paul Durand",
 Date = Nulle}
```

Clés

Constituants

On appelle **constituant** d'une relation un sous-ensemble, éventuellement vide, des attributs de cette relation. Par exemple `{Nom, Age}` est un constituant de la relation `Personne`.

Un constituant réduit à un seul attribut A sera noté indiféremment {A} ou A.

Clés candidates et clé primaire

Un constituant X est la **clé candidate** d'une relation R si :

- pour chaque n-uplet de R, la valeur de X identifie de façon unique ce n-uplet,
- aucun attribut de X ne peut être supprimé sans détruire la propriété précédente.

Une relation peut avoir une ou plusieurs clés candidates : l'une est choisie comme **clé primaire**.

Soit par exemple, la relation :

```
Personne(Nom, Prénom, NumSS, Pays)
```

Si l'on suppose qu'une personne est identifiée par son nom et son prénom ou bien par son numéro de sécurité sociale les clés candidates sont `{Nom, Prénom}` et `NumSS`. Cette dernière pourra être choisie comme clé primaire.

Par convention on souligne les attributs de la clé primaire. Par exemple :

Personne(Nom, Prénom, <u>NumSS</u>, Pays)

Clés étrangères

Un constituant Y d'une relation R_1 est une clé étrangère de R_1 s'il existe une relation R_2 possédant une clé primaire X et que Y a pour domaine l'ensemble des valeurs de X. On dit que Y **réfère** la relation R_2.

Soit par exemple la relation :

```
Personne(Nom, Prénom, Age)
Livre(Cote, Titre, Nom_Auteur, Prénom_Auteur)
```

Pour indiquer que l'auteur d'un livre est une personne, on déclare que le constituant `{Nom_Auteur, Prénom_Auteur}` est une clé étrangère de la relation `Livre`. Elle réfère la relation `Personne`.

Intégrité

Les contraintes d'intégrité d'une BD relationnelle peuvent s'exprimer par :

- l'appartenance des valeurs d'attributs à des domaines,
- la définition des clés,
- la normalisation des relations,
- un ensemble d'assertions,
- des conditions associées aux opérations de mise à jour.

Concernant les clés, deux formes d'intégrité jouent un rôle important :

- L'**intégrité d'entité** qui est vérifiée si les valeurs des attributs de la clé primaire ne sont pas nulles.
- L'**intégrité référentielle** qui est vérifiée si chaque valeur d'une clé étrangère Y :
  - soit existe comme valeur de la clé primaire d'un n-uplet de la relation que Y réfère,
  - soit est nulle.

Exemple

Cette BD concerne les premières ascensions sur un sommet de plus de 8000 m. Elle est nommée “Plus 8000”.

On veut représenter les faits suivants :

- Le nom et l’altitude de chaque sommet. Un sommet est identifié par son nom.
- La localisation d’un sommet, c’est à dire le nom du ou des pays dans lequel il se trouve. Un sommet peut se trouver dans plusieurs pays quand il appartient à la frontière de chacun de ces pays.
- La première ascension d’une face ou d’une arête d’un sommet et l’année de cette ascension. Une première est identifiée par le sommet et la face gravis.
- Les grimpeurs ayant atteint le sommet lors d’une première : ces grimpeurs sont identifiés par leur nom.

Domaines

```
Nom_de_sommet =
    {"Everest", "K2", "Kangchenjunga", "Lhotse", "Makalu",
     "Cho Oyu", "Dhaulagiri", "Manaslu", "Nanga Parbat",
     "Annapurna", "Hidden Peak", "Broad Peak",
     "Gasherbrum II", "Shisha Pangma"}
```

```
Altitude_de_sommet = {a | Entier a et a > 8000}
```

(on s'intéresse aux sommets de plus de 8000 m)

```
Orientation = {"N", "S", "O", "E", "NO", "SO", "NE", "SE"}
```

```
Année_d_ascension = {a | Entier a et a ≥ 1950}
```

(le premier 8000 a été conquis en 1950)

```
Nom_de_pays = {"Népal", "Chine", "Inde", "Pakistan"}
```

Relations

```
Sommet(Nom_Sommet: Nom_de_sommet, Altitude: Altitude_de_sommet)
```

```
Localisation(Nom_Sommet: Nom_de_sommet, Pays: Nom_de_pays)
```

```
Première(Nom_Sommet: Nom_de_sommet,
         Face: Orientation,
         Année: Année_d_ascension)
```

```
Conquête(Grimpeur: Chaîne,
         Nom_Sommet: Nom_de_sommet,
         Face: Orientation)
```

Clés

- Un sommet a une et une seule altitude. La clé de la relation `Sommet` est donc le constituant `Nom_Sommet`.
- Un sommet peut se trouver sur la frontière de plusieurs pays. La clé de la relation `Localisation` est donc formée de ses deux attributs `{Nom_Sommet, Pays}`. La localisation doit concerner un sommet de la BD. Le constituant `Nom_Sommet` de la relation `Localisation` est donc une clé étrangère qui réfère la relation `Sommet`.
- Une première est identifiée par le sommet et la face gravie. Le constituant (`Nom_Sommet`, `Face`) est donc la clé primaire de la relation `Première`.
  Le sommet gravi lors d'une première doit être un sommet décrit dans la BD. Le constituant `Nom_Sommet` est donc une clé étrangère de la relation `Première` qui réfère la relation `Sommet`.
- Un première peut avoir été réalisée par plusieurs grimpeurs. La clé de la relation `Conquête` est donc formée de ses trois attributs `{Grimpeur, Nom_Sommet, Face}`. Le sommet et la face conquis doivent être déclarés comme première. Le constituant {`Nom_Sommet`, `Face`} est donc une clé étrangère de la relation `Conquête` qui réfère la relation `Première`.

Vision tabulaire

Sommet	
Nom_Sommet	**Altitude**
Everest	8848
Manaslu	8163
Hidden Peak	8068

Localisation	
Nom_Sommet	**Pays**
Everest	Népal
Everest	Chine
Manaslu	Népal
Hidden-Peak	Chine
Hidden-Peak	Pakistan

Première		
Nom_Sommet	**Face**	**Année**
Everest	S	1953
Manaslu	S	1972
Hidden-Peak	NO	1975
Everest	SO	1975
Manaslu	O	1981

Conquête		
Grimpeur	**Nom_Sommet**	**Face**
Hillary	Everest	S
Tensing	Everest	S
Messner	Manaslu	S
Messner	Everest	SO
Habeler	Everest	SO
Habeler	Hidden Peak	NO
Messner	Hidden Peak	NO
Béghin	Manaslu	O

Vision assertionnelle

- L'extension de la relation `Sommet` est formée de l'ensemble des doublets {`Nom_Sommet` = s, `Altitude` = a} exprimant le fait que le sommet de nom s possède l'altitude a.
- L'extension de la relation `Localisation` est formée de l'ensemble des doublets {`Nom_Sommet` = s, `Pays` = p} exprimant le fait que le sommet de nom s se trouve dans le pays de nom p.
- L'extension de la relation `Première` est formée de l'ensemble des triplets {`Nom_Sommet` = s, `Face` = f, `Année` = a} exprimant le fait que la première ascension de la face f du sommet de nom s a été effectuée l'année a.
- L'extension de la relation `Conquête` est formée de l'ensemble des triplets {`Grimpeur` = g, `Nom_Sommet` = s, `Face` = f} exprimant le fait que le grimpeur de nom g a gravi le sommet de nom s lors de la première ascension de la face f.

Langages relationnels

Introduction

On distingue deux types de langages relationnels :

1) Les **langages algébriques** basés sur l'algèbre relationnelle : on exploite la vision ensembliste d'une BD relationnelle.

2) Les **langages prédicatifs** basés sur le calcul relationnel : on exploite la vision assertionnelle d'une BD relationnelle.

Il est démontré que ces deux types de langages ont la même puissance.

Algèbre relationnelle

L'**algèbre relationnelle** est un ensemble d'opérateurs qui manipulent des extensions de relations, c'est à dire des ensembles de n-uplets.

Tout opérateur de l'algèbre relationnelle s'applique à une ou plusieurs relations et produit une relation.

Parmi ces opérateurs on distingue :

- des opérateurs ensemblistes classiques : union, intersection, différence, produit cartésien, etc.
- des opérateurs spécifiques aux bases de données: sélection, projection, jointure, etc.

Notations

- Nous noterons :
  - P, Q, R des expressions ayant pour valeur une relation,
  - A, B, C des noms d'attributs,
  - X, Y, Z des constituants,
  - r.A la valeur de l'attribut A dans le n-uplet r,
  - attributs(X) l'ensemble des noms des attributs du constituant X,
  - $e_1, \ldots, e_n$: t le fait que les expressions $e_1, \ldots, e_n$ sont de type t.
  - e = v le fait que l'expression e a la valeur v.
- Si $X = \{A_1, \ldots, A_n\}$ est un constituant et que les domaines respectifs de $A_1, \ldots, A_n$ sont $D_1, \ldots, D_n$ alors les écritures abrégées suivantes seront utilisées :
  - X: D pour $A_1: D_1, \ldots, A_n: D_n$
  - r.X pour $\{A_1 = r.A_1, \ldots, A_n = r.A_n\}$
- Si n est le nom d'une relation R alors n est une expression qui a pour type le type de R et pour valeur l'extension de R.

Opérateurs ensemblistes classiques

Union, intersection et différence

Soit :

P, Q : rel(X: D)

on a :

union(P, Q) : rel(X: D)

union(P, Q) = {r | r ∈ P ou r ∈ Q}

inter(P, Q) : rel(X: D)

inter(P, Q) = {r | r ∈ P et r ∈ Q}

minus(P, Q) : rel(X: D)

minus(P, Q) = {r | r ∈ P et r ∉ Q}

Par exemple :

minus(

Sommet	Altitude
Hidden Peak	8086
Everest	8848
Manaslu	8163

,

Sommet	Altitude
Everest	8848
Manaslu	8163

) =

Sommet	Altitude
Hidden Peak	8086

Produit cartésien

Soit :

P : rel(X: D_X)

Q : rel(Y: D_Y)

attributs(X) $\cap$ attributs(Y) = $\varnothing$

on a :

pc(P, Q) : rel(X: D_X, Y: D_Y)

pc(P, Q) = $\{p \cup q \mid p \in P, q \in Q\}$

Par exemple :

pc(

Sommet1	Altitude
Everest	8848

,

Sommet2	Pays
Everest	Népal
Everest	Chine

) =

Sommet1	Altitude	Sommet2	Pays
Everest	8848	Everest	Népal
Everest	8848	Everest	Chine

Sélection

L'opérateur de **sélection** extrait l'ensemble des n-uplets d'une relation qui vérifient une condition donnée.

Soit :

R : rel(X: D)

f : {X: D} –> Booléen

on a :

sel(R, f) : rel(X: D)

sel(R, f) = {r | r ∈ R et f(r) = Vrai}

Dans le cas fréquent où la fonction f est un comparateur de valeur d'attribut on adoptera l'écriture simplifiée suivante :

sel(R, A θ v) ≡ sel(R, fun r r.A θ v)

Par exemple :

sel(

Sommet	Altitude
Everest	8848
Manaslu	8163
Hidden Peak	8086

, Altitude > 8500) =

Sommet	Altitude
Everest	8848

Projection

L'opérateur de **projection** extrait l'ensemble des valeurs d'un constituant d'une relation.

Soit :

R : rel(X: D_X, Y: D_Y)

on a :

proj(R, X) : rel(X: D_X)

proj(R, X) = {r.X | r ∈ R}

Par exemple :

proj(

Grimpeur	Sommet	Face
Hillary	Everest	S
Tensing	Everest	S
Messner	Manaslu	S
Habeler	Hidden Peak	NO
Messner	Hidden Peak	NO
Beghin	Manaslu	O

, {Sommet, Face}) =

Sommet	Face
Hidden Peak	NO
Manaslu	S
Everest	S
Manaslu	O

Jointure

L'opérateur de jointure fusionne deux relations en produisant toutes les unions de leurs n-uplets respectifs qui vérifient un certain critère.

Soit :

$P : rel(X: D_X)$

$Q : rel(Y: D_Y)$

$attributs(X) \cap attributs(Y) = \varnothing$

$f : \{X: D_X, Y: D_Y\} \rightarrow$ Booléen

on a :

join$(P, Q, f) : rel(X: D_X, Y: D_Y)$

join$(P, Q, f) = \{p \cup q \mid p \in P, q \in Q, f(p \cup q) = Vrai\}$

Il est intéressant de traiter comme une opération spécifique l'**équi-jointure** pour laquelle le critère est l'égalité de deux constituants.

On a :

equi_join$(P, Q, X = Y) \equiv$ **join**$(P, Q, fun(j)\ j.X = j.Y)$

où X et Y sont des constituants respectifs de P et Q de domaines compatibles.

Par exemple :

equi_join(

Sommet1	Face
Everest	N
Manaslu	S
Hidden Peak	NO
Manaslu	O

,

Sommet2	Altitude
Everest	8848
Manaslu	8163

,

Sommet1 = Sommet2) =

Sommet1	Face	Sommet2	Altitude
Everest	N	Everest	8848
Manaslu	S	Manaslu	8163
Manaslu	O	Manaslu	8163

Jointure externe

Il peut être utile de conserver dans le résultat d'une jointure, les n-uplets de chacune des deux relations qui ne peuvent être joints. Ceci peut être obtenu par l'opération de **jointure externe**.

Soit :

P : rel(X: D_X)

Q : rel(Y: D_Y)

attributs(X) $\cap$ attributs(Y) = $\varnothing$

$X = \{X_1, \ldots, X_m\}$

$Y = \{Y_1, \ldots, Y_n\}$

on a :

ext_join(P, Q, f) : rel(X: D_X, Y: D_Y)

ext_join(P, Q, f) = **union**(P_{ext}, J, Q_{ext})

où :

J = **join**(P, Q, f)

P_{ext} = **pc**(**minus**(P, proj(J, X)), {{Y_1 = `Nulle`, …, Y_n = `Nulle`}})

Q_{ext} = **pc**({{X_1 = `Nulle`, …, X_m = `Nulle`}}, **minus**(Q, **proj**(J, Y)))

Par exemple :

ext_equi_join(

Sommet1	Altitude
Manaslu	8163
Everest	8848

,

Sommet2	Face
Everest	S
Annapurna	N

,

Sommet1 = Sommet2) =

Sommet1	Altitude	Sommet2	Face
Manaslu	8163	Nulle	Nulle
Everest	8847	Everest	S
Nulle	Nulle	Annapurna	N

Semi-jointure

L'opérateur de semi-jointure extrait l'ensemble des n-uplets d'une relation dont l'union avec l'un au moins des n-uplets d'une seconde relation vérifie un certain critère. Il exprime la quantification existentielle.

Soit :

P : rel(X: D_X)

Q : rel(Y: D_Y)

attributs(X) $\cap$ attributs(Y) = $\varnothing$

f : {X: D_X, Y: D_Y} –> Booléen

on a :

semi_join(P, Q, f) : rel(X: D_X)

semi_join(P, Q, f) = {p | p $\in$ P et $\exists$q $\in$ Q, f(p $\cup$ q) = Vrai}

Par exemple :

equi_semi_join(

Sommet1	Face
Everest	N
Manaslu	S
Hidden Peak	NO
Manaslu	O
K2	NE
Annapurna	N

,

Sommet2
Annapurna
Manaslu

,

Sommet1 = Sommet2) =

Sommet1	Face
Annapurna	N
Manaslu	S
Manaslu	O

Division

L'opérateur de division exprime la quantification universelle.

Soit :

$P : rel(X: D_X, Y: D_Y)$

$Q : rel(Z: D_Y)$

$attributs(Y) \cap attributs(Z) = \emptyset$

$f : \{X: D_X, Y: D_Y, Z: D_Y\} \rightarrow$ Booléen

on a :

$div(P, Q, f) : rel(X: D_X)$

$div(P, Q, f) =$

$\{x \mid x \in proj(P, X) \text{ et } (\forall q \in Q, \exists p \in P, p.X = x \text{ et } f(p \cup q) = Vrai)\}$

Par exemple :

equi_div(

Sommet	Face1
Everest	S
Manaslu	S
Everest	N
Everest	O
Manaslu	O
Everest	E

,

Face2
N
E
S
O

, Face1 = Face2) =

Sommet
Everest

Renommage d'attribut

Comme nous l'avons vu, les opérateurs binaires de l'algèbre relationnelle imposent des contraintes sur les noms d'attributs des opérandes (ensemble des noms disjoints ou identiques). Il peut donc être nécessaire de renommer certains attributs avant d'appliquer ces opérateurs : c'est le rôle de l'opérateur **rename**.

Soit :

R : rel(…, A: D, …)

on a :

rename(R, A, B) : rel(…, B: D, …)

rename(R, A, B) =

relation de même type et de même extension que R,

excepté le renommage de l'attribut A par B.

Par exemple :

rename(

Nom	Altitude
Everest	8848
Manaslu	8163
Hidden Peak	8086

, Nom , Nom_Sommet) =

Nom_Sommet	Altitude
Everest	8848
Manaslu	8163
Hidden Peak	8086

Exemples

On renomme `Nom_Sommet_Conquis` l'attribut `Nom_Sommet` de la relation `Conquête` :

```
Conquête := rename(Conquête, Nom_Sommet, Nom_Sommet_Conquis)
```

Nom, face et altitude des sommets conquis par Messner ?

```
C := sel(Conquête, Grimpeur = "Messner")
SC := equi_join(Sommet, C, Nom_Sommet = Nom_Sommet_Conquis)
Réponse := proj(SC, {Nom_Sommet, Face, Altitude})
```

Nom des grimpeurs ayant conquis un sommet du Pakistan ?

```
L := sel(Localisation, Pays = "Pakistan")
V := equi_semi_join(Conquête,
                    L,
                    Nom_Sommet_Conquis = Nom_Sommet);
Réponse = proj(V, Grimpeur)
```

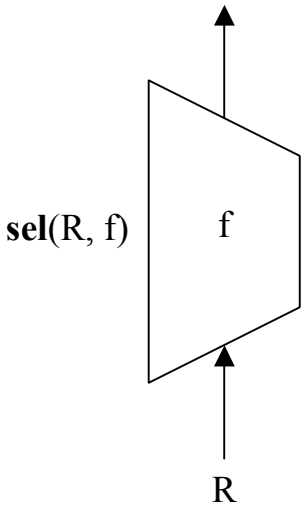
Nom des grimpeurs ayant conquis tous les sommets ?

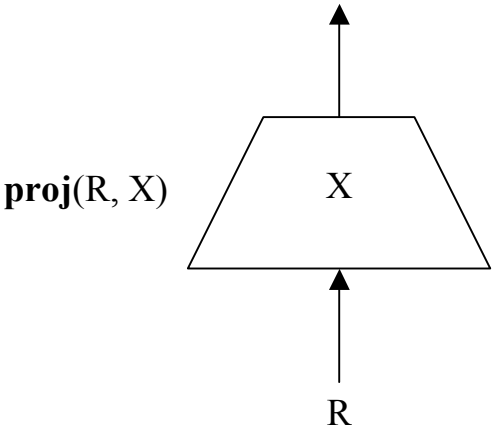
```
C = proj(Conquête, {Grimpeur, Nom_Sommet_Conquis})
S = proj(Sommet, Nom_Sommet)
Réponse = equi_div(C, S, Nom_Sommet_Conquis = Nom_Sommet)
```

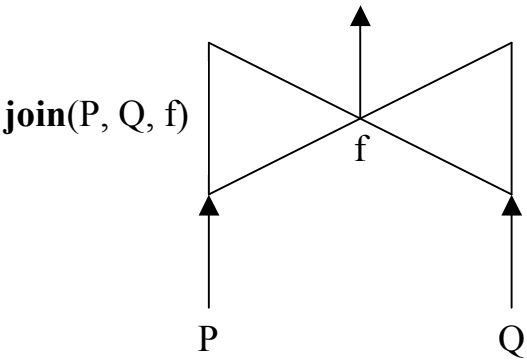
Nom des sommets de plus de 8500 mètres situés au Népal mais pas sur la frontière chinoise ?

```
S = sel(Sommet, fun(s) s.Altitude > 8500);
L1 = sel(Localisation, Pays = "Népal")
L2 = sel(Localisation, Pays = "Chine")
Réponse = inter(proj(S, Nom_Sommet),
                minus(proj(L1, Nom_Sommet),
                      proj(L2, Nom_Sommet)))
```

Représentation graphique







Calcul relationnel

Il existe deux types de **calcul relationnel** selon ce que représentent les variables :

- le **calcul relationnel de n-uplets** : une variable désigne un n-uplet (ou une ligne de table).
- le **calcul relationnel de domaines** : une variable désigne une valeur de domaine (le contenu d'une case d'une table).

Calcul relationnel de n-uplets

Une formule du calcul relationnel de n-uplets est définie par les règles suivantes :

- chaque variable est associée à une relation et est désigne un n-uplet de celle-ci ;
- si x est une variable, on note x.A la valeur de l'attribut A du n-uplet x ;
- R x est un terme qui est vrai si x est un n-uplet de la relation R ;
- t.A θ c (où c est une constante littérale) et $t_1.A_1$ θ $t_2.A_2$ sont des termes qui sont vrais si le prédicat de comparaison θ (< = > ≠) est vérifié, faux sinon ;
- un terme est une formule ;
- si f est une formule alors (f) et non f sont des formules ;
- si f et g sont des formules alors f et g, f ou g, f ⇒ g sont des formules ;
- si f est une formule dans laquelle x est une variable libre alors ∃x f et ∀x f sont des formules dans lesquelles x est liée et les variables libres de f autres que t restent libres ;
- ce sont les seules règles qui permettent de construire des formules.

Une expression du calcul relationnel de n-uplets est de la forme :

$$\{\{B_1 = x_1.A_1, \ldots, B_n = x_n.A_n\} \mid f\}$$

où f est une formule du calcul relationnel de n-uplets, $B_1, \ldots, B_n$ sont des noms d'attributs et $x_1, \ldots, x_n$ sont des variables libres dans f.

Elle se lit : « Calculer l'ensemble des n-uplets $\{B_1 = t_1.A_1, \ldots, B_n = t_n.A_n\}$ tels que f est vrai pour $x_1 = t_1, \ldots, x_n = t_n$.

Exemples

Nom et altitude des sommets de plus de 8100m ?

```
{{S = s.Nom_Sommet, A = s.Altitude} | Sommet s et
                                                s.Altitude > 8100}
```

Nom, face et altitude des sommets conquis par Messner ?

```
{{S = c.Nom_Sommet, F = c.Face, A = s.Altitude} | Conquête c et
        Sommet s et c.Nom_Sommet = s.Nom et c.Grimpeur = "Messner"}
```

Nom des grimpeurs ayant conquis un sommet du Pakistan ?

```
{{G = c.Grimpeur} | Conquête c et ∃l (Localisation l et
               l.Pays = "Pakistan" et l.Nom_Sommet = c.Nom_Sommet)}
```

Nom des grimpeurs ayant conquis tous les sommets ?

```
{{G = c.Grimpeur} | Conquête c et ∀s (Sommet s ⇒
                ∃c' (Conquête c' et c'.Nom_Sommet = s.Nom_Sommet et
                                        c'.Grimpeur = c.Grimpeur))}
```

Calcul relationnel de domaines

Une formule du calcul relationnel de domaines est définie par les règles suivantes :

- chaque variable est associée à un domaine et désigne une valeur de celui-ci ;
- $R(A_1 = v_1, \ldots, A_n = v_n)$ où $v_1, \ldots, v_n$ sont des constantes littérales ou des variables, est un terme qui est vrai s'il existe un n-uplet de la relation R ayant v_1 pour valeur de l'attribut $A_1, \ldots, v_n$ pour valeur de l'attribut A_n ;
- $v_1 \theta v_2$ où v_1 et v_2 sont des constantes littérales ou des variables, est un terme qui est vrai si le prédicat de comparaison θ ($<$ $=$ $>$ $\neq$) est vérifié et faux sinon ;
- les règles concernant les formules sont les mêmes que pour le calcul de n-uplets ;
- ce sont les seules règles qui permettent de construire des formules.

Une expression du calcul relationnel de domaines est de la forme :

$$\{\{A_1 = x_1, \ldots, A_n = x_n\} \mid f\}$$

où f est une formule du calcul relationnel de domaines, $A_1, \ldots, A_n$ sont des noms d'attributs et $v_1, \ldots, v_n$ sont des variables libres dans f.

Elle se lit : « Calculer l'ensemble des n-uplets $\{A_1 = v_1, \ldots, A_n = v_n\}$ tels que f est vraie pour $x_1 = v_1, \ldots, x_n = v_n$ ».

Exemples

Nom et altitude des sommets de plus de 8100m ?

```
{{S = s, A = a} | Sommet(Nom_Sommet = s, Altitude = a) et
                                                    a > 8100}
```

Nom, face et altitude des sommets conquis par Messner ?

```
{{S = s, F = f, A = a} |
          Conquête(Grimpeur = "Messner", Nom_Sommet = s, Face = f)
                        et Sommet(Nom_Sommet = s, Altitude = a)}
```

Nom des grimpeurs ayant conquis un sommet du Pakistan ?

```
{{G = g} | ∃s (Conquête(Grimpeur = g, Nom_Sommet = s) et
                  Localisation(Nom_Sommet = s, Pays = "Pakistan")}
```

Nom des grimpeurs ayant conquis tous les sommets ?

```
{{G = g} | ∀s (Sommet(Nom_Sommet = s) ⇒
                         Conquête(Grimpeur = g, Nom_Sommet = s)}
```

Manipulation des valeurs nulles

La présence éventuelle de valeurs nulles dans les n-uplets impliquent de définir des règles pour leur manipulation.

Il est nécessaire notamment d'introduire une troisième valeur de vérité que nous noterons `Indéfini`.

Ces règles sont les suivantes :

- Le résultat de la **comparaison** entre la valeur `Nulle` et toute autre valeur y compris `Nulle` a pour valeur `Indéfini`.
- Les **tables de vérité** des trois connecteurs logiques sont modifiées de la façon suivante :

et	Vrai	Faux	Indéfini
Vrai	Vrai	Faux	Indéfini
Faux	Faux	Faux	Faux
Indéfini	Indéfini	Faux	Indéfini

ou	Vrai	Faux	Indéfini
Vrai	Vrai	Vrai	Vrai
Faux	Vrai	Faux	Indéfini
Indéfini	Vrai	Indéfini	Indéfini

	non
Vrai	Faux
Faux	Vrai
Indéfini	Indéfini

Vues

Une **vue** est une relation virtuelle, exprimée en termes d'un langage relationnel.

Les vues permettent de définir les modèles externes.

Supposons, par exemple, que l'on travaille souvent sur les premières sur un sommet du Népal. On pourra définir la vue suivante:

```
Première_au_Népal(Nom_Sommet, Face, Altitude, Année) =
             {{Nom_Sommet = s, Face = f, Altitude = h, Année = a} |
                      Première(Nom_Sommet = s, Face = f, Année = a)
                            et Sommet(Nom_Sommet = s, Altitude = h)
                   et Localisation(Nom_Sommet = s, Pays = "Népal")}
```

qui pourra être ensuite manipulée comme si elle était une relation définie en extension.

Par exemple les noms des sommets népalais de plus de 8500m gravis en 1954 pourront être obtenus par la requête :

```
{{S = s} | Première_au_Népal(Nom_Sommet = s, Altitude = h,
                                          Année = 1954) et h > 8500}
```

Conception d’un schéma relationnel

Introduction

Une même BD peut être représentée par différents ensembles de relations. Selon la représentation choisie, certains problèmes peuvent se poser lors de la mise à jour de la BD. Considérons par exemple la relation :

Nom_Sommet	**Face**	**Altitude**	**Année**
Everest	S	8848	1953
Manaslu	S	8125	1972
Hidden-Peak	NO	8068	1975
Everest	SO	8848	1975

La mise à jour de cette relation pose les problèmes suivants :

- **insertion** : un sommet ne peut être inséré que si une première a eu lieu sur ce sommet.
- **modification** : si l'altitude de l'Everest est modifiée il faut la modifier dans toutes les premières sur ce sommet.
- **suppression** : si l'unique première sur un sommet est supprimée l'information concernant son altitude est perdue.

La solution à ces problèmes consiste à **normaliser** le schéma de la BD en décomposant certaines relations.

Cette décomposition s'appuie sur les **dépendances** qui existent entre les constituants d'une relation. Par exemple, la dépendance entre le nom d'un sommet et son altitude ou bien entre le nom d'un sommet et les pays dans lesquels il se trouve.

Normalisation utilisant les dépendances fonctionnelles

Dépendance fonctionnelle

Il y a une **dépendance fonctionnelle** entre un constituant X et un constituant Y d'une relation R si pour toute extension de R, à chaque valeur de X il correspond toujours la même valeur de Y. On dit que X **détermine fonctionnellement** Y et l'on note :

X –> Y

Une dépendance fonctionnelle X -> Y est dite **triviale** si Y est vide ou bien est un sous-ensemble de X. Par exemple, les dépendances :

```
{Nom_Sommet, Face} -> Face
{Nom_Sommet, Pays} -> {}
```

sont triviales.

Dans la BD "Plus 8000" on a les dépendances fonctionnelles non triviales suivantes :

```
Nom_Sommet –> Altitude
{Nom_Sommet, Face} –> Année
```

Propriétés des dépendances fonctionnelles

A partir d'un ensemble de dépendances fonctionnelles, on peut en déduire d'autres.

Par exemple, à partir des dépendances `Grimpeur -> Ville` (ville de naissance) et `Ville -> Pays` on peut déduire la dépendance `Grimpeur -> Pays` (pays de naissance).

Etant donné un ensemble de dépendances F construit sur l'ensemble R des attributs d'une relation, l'ensemble F^+ (**fermeture** de F) de toutes les dépendances fonctionnelles logiquement impliquées par F peut être calculé à partir des trois règles suivantes (axiomes d'Armstrong) :

(**réflexivité**) si $Y \subseteq X \subseteq R$ alors X -> Y

(**augmentation**) si X -> Y et $Z \subseteq R$ alors $X \cup Z$ -> $Y \cup Z$

(**transitivité**) si X -> Y et Y -> Z alors X -> Z

Des axiomes d'Armstrong on peut déduire deux règles plus pratiques pour calculer F^+ :

(**union**) si X -> Y et X -> Z alors X -> $Y \cup Z$.

(**décomposition**) si X -> $Y \cup Z$ alors X -> Y et X -> Z.

Dépendances fonctionnelles et clés

On peut définir une clé d'une relation comme un sous-ensemble de ses attributs qui détermine tous les autres.

Un constituant X d'une relation $R(A_1, \ldots, A_n)$ est une clé de cette relation si et seulement si :

- X -> $\{A_1, \ldots, A_n\}$,
- il n'existe pas de constituant Y inclus dans X tel que Y -> $\{A_1, \ldots, A_n\}$.

Etant données une relation et l'une de ses clés, on appelle **attribut clé** un attribut qui appartient à cette clé et **attribut non clé**, un attribut qui n'y appartient pas.

On appelle **super-clé** un constituant qui inclut une clé.

Par exemple dans la relation `Premiere(Nom_Sommet, Face, Année)` de clé `{Nom_Sommet, Face}` :

- `{Nom_Sommet, Face}` est une clé car `{Nom_Sommet, Face}` -> `Année` et par augmentation `{Nom_Sommet, Face}` -> `{Nom_Sommet, Face, Année}`,
- `Nom_Sommet` et `Face` sont des attributs clés pour cette clé,
- `Année` est un attribut non clé pour cette clé.

1ère forme normale

Une relation est en **1ère forme normale** si tous ses attributs ont une valeur atomique. Le fait que dans le modèle relationnel les domaines soient atomiques implique que toutes les relations sont en 1ère forme normale.

Par exemple, la relation suivante n'est pas en 1ère forme normale car l'attribut `Pays` possède des valeurs non atomiques.

Nom_Sommet	**Pays**
Everest	Chine, Népal
K2	Chine, Pakistan

La mise sous forme normale associe chaque sommet à chacun des pays auquel il appartient.

Nom_Sommet	**Pays**
Everest	Chine
Everest	Népal
K2	Chine
K2	Pakistan

2e forme normale

Une relation est en **2e forme normale** si elle est en 1ère forme normale et si chaque attribut non clé dépend totalement et non partiellement de la clé primaire.

Par exemple, la relation suivante :

Nom_Sommet	**Face**	**Altitude**	**Année**
Everest	S	8848	1953
Manaslu	S	8125	1972
Hidden-Peak	NO	8068	1975
Everest	SO	8848	1975

n'est pas en 2e forme normale car étant donné la dépendance `Nom_Sommet -> Altitude` l'attribut `Altitude` ne dépend que de l'attribut `Nom_Sommet` de la clé primaire.

Elle peut être décomposée en deux relations :

Nom_Sommet	**Face**	**Année**
Everest	S	1953
Manaslu	S	1972
Hidden-Peak	NO	1975
Everest	SO	1975

Nom_Sommet	**Altitude**
Everest	8848
Manaslu	8125
Hidden-Peak	8068

qui sont en 2e forme normale et préservent les dépendances fonctionnelles.

On remarquera qu'une relation qui a une clé primaire mono-attribut est en 2e forme normale.

3e forme normale

Une relation est en **3e forme normale** si elle est en 2e forme normale et si pour chaque clé candidate il n'existe pas de dépendances entre les attributs n'appartenant pas à cette clé.

Par exemple, si l'on suppose les dépendances fonctionnelles `Grimpeur -> Ville` et `Ville -> Pays` la relation suivante :

Grimpeur	**Ville**	**Pays**
Compagnoni	Cortina	Italie
Lacedelli	Cortina	Italie
Buhl	Innsbruck	Autriche

n'est pas en 3e forme normale car l'on a une dépendance entre `Ville` et `Pays` qui n'appartiennent pas à la clé.

Elle peut être décomposée en deux relations :

Grimpeur	**Ville**
Compagnoni	Cortina
Lacedelli	Cortina
Buhl	Innsbruck

Ville	**Pays**
Cortina	Italie
Innsbruck	Autriche

qui sont en 3e forme normale et préservent les dépendances fonctionnelles.

Forme normale de Boyce-Codd

Une relation R est en **forme normale de Boyce-Codd** (**BCNF**) si pour chaque dépendance fonctionnelle non triviale X -> Y, X est une super-clé de R.

La forme normale de Boyce-Codd implique la 3e forme normale.

Cette forme normale a été introduite pour tenir compte des dépendances entre parties de clé et des dépendances entre un attribut non clé et une partie de la clé.

Par exemple, la relation

```
Adresse(Rue, Ville, Code_postal)
```

avec les dépendances fonctionnelles `{Rue, Ville} -> Code_postal` et `Code_postal -> Ville`, a deux clés candidates `{Rue, Ville}` et `{Rue, Code_postal}`.

Elle n'est pas en BCNF car dans la dépendance `{Rue, Code_postal} -> Ville`, `Code_postal` n'est pas une super-clé.

Elle peut être décomposée en deux relations :

```
RC(Rue, Code_postal)
CV(Code_postal, Ville)
```

qui sont en forme normale de Boyce-Codd. Remarquons cependant que cette décomposition ne préserve pas la dépendance fonctionnelle `{Rue, Ville} -> {Code_postal}`.

Normalisation utilisant les dépendances multivaluées

Dépendance multivaluée

On dit qu'il y a une **dépendance multivaluée** entre un constituant X et un constituant Y d'une relation R(X, Y, Z) si pour toute extension de R, à chaque valeur de X il correspond toujours le même ensemble de valeurs de Y et que cet ensemble de valeurs ne dépend pas des valeurs de Z. On dit que X **multidétermine** Y et l'on note :

X ->> Y

Dans la relation `Localisation` on a la dépendance multivaluée :

```
Nom_Sommet ->> Pays
```

et dans la relation `Conquête` :

```
{Nom_Sommet, Face} ->> Grimpeur
```

Propriétés des dépendances fonctionnelles et multivaluées

Etant donné un ensemble de dépendances D, fonctionnelles ou multivaluées, construit sur l'ensemble R des attributs d'une relation, la fermeture D^+ de toutes les dépendances logiquement impliquées par D peut être calculée à partir des axiomes d'Armstrong et des 5 règles suivantes :

(**complémentation**) si X ->> Y alors X ->> R - X - Y

(**augmentation multivaluée**) si X ->> Y et $V \subseteq R$ et $W \subseteq V$ alors
$V \cup X$ ->> $W \cup Y$.

(**transitivité multivaluée**) si X ->> Y et Y ->> Z alors X ->> Z - Y.

(**réplication**) si X -> Y alors X ->> Y.

(**fusion**) si X ->> Y et $Z \subseteq Y$ et si il existe $W \subseteq R$ et disjoint de Y tel que
W -> Z, alors X -> Z.

De ces 8 règles on peut déduire 3 règles plus simples pour calculer D^+ :

(**union multivaluée**) si X ->> Y et X ->> Z alors X ->> $Y \cup Z$

(**intersection**) si X ->> Y et X ->> Z alors X ->> $Y \cap Z$

(**différence**) si X ->> Y et X ->> Z alors X ->> $Y - Z$ et X ->> Z - Y.

4e forme normale

Une relation R est en 4e forme normale, si pour chaque dépendance multivaluée non triviale X ->> Y telle que X ∪ Y n'inclut pas tous les attributs de R, X est une super-clé de R.

La 4e forme normale implique la forme normale de Boyce-Codd puisqu'une dépendance fonctionnelle est un cas particulier de dépendance multivaluée.

Par exemple, la relation suivante

Grimpeur	**Nom_Sommet**	**Pays**	**Face**
Hillary	Everest	Népal	S
Hillary	Everest	Chine	S
Tensing	Everest	Népal	S
Tensing	Everest	Chine	S

n'est pas en 4e forme normale car la dépendance multivaluée `Nom_Sommet ->> Pays` qui n'est pas une super-clé, n'inclut pas tous les attributs.

Elle peut être décomposée en deux relations :

Nom_Sommet	**Face**	**Grimpeur**
Everest	S	Hillary
Everest	S	Tensing

Nom_Sommet	**Pays**
Everest	Népal
Everest	Chine

qui sont en 4e forme normale car la dépendance `{Nom_Sommet, Face} ->> Grimpeur` inclut tous les attributs de la première relation et la dépendance `Nom_Sommet ->> Pays` inclut tous les attributs de la deuxième relation.

Propriétés d'une décomposition

Soit R un schéma de relation et R_1, …, R_n les relations issues d'une décomposition D de R :

- D est **sans perte** si la relation R peut être recomposée par jointures des relations R_1, …, R_n.
- D **préserve les dépendances fonctionnelles** si les dépendances fonctionnelles de R sont impliquées par les dépendances fonctionnelles de R_1, …, R_n.

Il est démontré que :

- Toute relation a au moins une décomposition en 3e forme normale qui préserve les dépendances fonctionnelles et qui est sans perte.
- Toute relation a au moins une décomposition en 4e forme normale qui est sans perte mais qui peut ne pas préserver les dépendances fonctionnelles.

  C'est le cas notamment de la relation

  ```
  Adresse(Rue, Ville, Code_postal)
  ```

  étudiée ci-dessus.

Décomposition d'un schéma relationnel en 4e forme normale

La décomposition d'un schéma relationnel en 4e forme normale, ou en BCNF s'il n'y a pas de dépendances multivaluées, peut être obtenue par l'algorithme suivant :

1) On part d'un schéma S constitué d'une seule relation R, dite **universelle**, composée de tous les attributs et de l'ensemble D des dépendances entre les attributs de U.

2) Tant qu'il existe dans S une relation R qui n'est pas en 4e forme normale, on cherche dans D^+ une dépendance X ->> Y telle que R(X, Y, Z) et X n'est pas une clé de R et l'on remplace R dans S par les deux relations R_1(X, Y) et R_2(X, Z).

3) On élimine les redondances en remplacant tout couple de relations P(X) et Q(Y) par P(X) si $X \subset Y$.

Pour obtenir une décomposition qui préserve les dépendances fonctionnelles de D, il ne faut effectuer, à l'étape 2, que des décompositions qui préservent ces dépendances.

Exemple 1

```
S = {R(Nom_Sommet, Altitude, Pays, Face, Année, Grimpeur)}
D = {Nom_Sommet -> Altitude,
     Nom_Sommet ->> Pays,
     {Nom_Sommet, Face} -> Année,
     {Nom_Sommet, Face} ->> Grimpeur}
_____
S = {R1(Nom_Sommet, Altitude),
     R2(Nom_Sommet, Pays, Face, Année, Grimpeur)}
_____
S = {R1(Nom_Sommet, Altitude),
     R3(Nom_Sommet, Pays),
     R4(Nom_Sommet, Face, Année, Grimpeur)}
_____
S = {R1(Nom_Sommet, Altitude),
     R3(Nom_Sommet, Pays),
     R5(Nom_Sommet, Face, Année),
     R6(Nom_Sommet, Face, Grimpeur)}
```

Le schéma S est en 4e forme normale (et donc en 3e forme normale) et les dépendances sont préservées.

Exemple 2

```
S = {R(NCom, NLig, COpt, CVoit, PCat, PFac)}
D = {{NCom, NLig} -> CVoit,
     {NCom, NLig, COpt} -> PFac,
     {CVoit, COpt} -> PCat}
```

Décomposition avec préservation des dépendances

```
S = {R1(CVoit, COpt, PCat),
     R2(NCom, NLig, COpt, CVoit, PFac)}
```

```
S = {R1(CVoit, COpt, PCat),
     R21(NCom, NLig, CVoit),
     R22(NCom, NLig, COpt, PFac)}
```

Le schéma S est en BCNF (et donc en 3e forme normale) et les dépendances fonctionnelles sont préservées.

Décomposition sans préservation des dépendances

```
S = {R1(NCom, NLig, CVoit),
     R2(NCom, NLig, COpt, PCat, PFac)}
```

Le schéma S est en BCNF (et donc en 3e forme normale) mais la dépendance fonctionnelle `{CVoit, COpt} -> PCat` est perdue.

SQL

Introduction

Le langage SQL est la version commercialisée du langage SEQUEL du prototype System R développé à IBM San José à partir de 1975.

Il a depuis fait l'objet de plusieurs normes dont la plus récente est SQL2 et il a été adopté par la plupart des SGBD relationnels du marché.

Le langage SQL est basé à la fois sur l'algèbre relationnelle et le calcul relationnel de n-uplets.

Le langage décrit dans ce chapitre est conforme à la norme SQL2.

Dans les règles de grammaire les métasymboles suivants seront utilisés :

|	disjonction
[]	optionalité
liste de *T*	≡ *T* [, …, *T*]

Définition des données

Création d'un domaine

`CREATE DOMAIN` *nom définition_de_domaine* ;

définition_de_domaine ::=

type

`DEFAULT` *valeur_par_défaut*

[`CONSTRAINT` *nom*] `CHECK (`*condition*`)`

type ::=

`CHAR` [`(`*longueur*`)`]

| `VARCHAR` [`(`*longueur*`)`]

| `INTEGER`

| `FLOAT`

| `DATE`

…

valeur_par_défaut ::=

constante

| `USER`

| `NULL`

| `CURRENT_DATE`

…

Exemples

```
CREATE DOMAIN NOM_DE_SOMMET
   CHAR(20);

CREATE DOMAIN ORIENTATION
   CHAR(2)
   CHECK (VALUE IN ('N', 'S', 'O', 'E', 'NO', 'SO', 'NE', 'SE'))
   NOT DEFERRABLE;

CREATE DOMAIN NOM_DE_PAYS
   CHAR(10)
   CHECK (VALUE IN ('Pakistan', 'Chine', 'Nepal', 'Inde'))
   NOT DEFERRABLE;
```

Création d'une table

`CREATE TABLE` *nom*

(liste de *définition_de_colonne*

[liste de *contrainte_de_table*]);

définition_de_colonne ::=

nom

(*nom_de_domaine* | *type*)

[liste de *contrainte_de_colonne*]

[`DEFAULT` *valeur_par_défaut*]

contrainte_de_colonne ::=

[`CONSTRAINT` *nom*]

type_de_contrainte_de_colonne

mode_de_contrainte

type_de_contrainte_de_colonne ::=

`PRIMARY KEY`

| `NOT NULL`

| `UNIQUE`

| `CHECK (`*condition_sur_valeur*`)`

| `REFERENCES` *nom_de_table*`(`*nom_de_colonne*`)`

mode_de_contrainte ::=

[`NOT`] `DEFERRABLE`

contrainte_de_table ::=

[`CONSTRAINT` *nom*]

type_de_contrainte_de_table

mode_de_contrainte

type de contrainte de table ::=

`PRIMARY KEY` (liste de *nom_de_colonne*)

| `NOT NULL` (liste de *nom_de_colonne*)

| `UNIQUE` (liste de *nom_de_colonne*)

| `CHECK` (*condition_sur_ligne*)

| `FOREIGN KEY` liste de *nom_de_colonne* `REFERENCES`

nom de table (liste de *nom_de_colonne*)

Exemples

```
CREATE TABLE SOMMET
   (NOM_SOMMET NOM_DE_SOMMET PRIMARY KEY,
    ALTITUDE INTEGER CHECK (ALTITUDE > 8000));

CREATE TABLE LOCALISATION
   (NOM_SOMMET NOM_DE_SOMMET REFERENCES SOMMET(NOM),
    PAYS NOM_DE_PAYS);

CREATE TABLE PREMIERE
   (NOM_SOMMET NOM_DE_SOMMET,
    FACE ORIENTATION,
    ANNEE INTEGER NOT NULL CHECK (ANNEE > 1950),
    PRIMARY KEY (NOM_SOMMET, FACE)
    FOREIGN KEY (NOM_SOMMET) REFERENCES SOMMET(NOM_SOMMET));

CREATE TABLE CONQUETE
   (GRIMPEUR CHAR(20) NOT NULL,
    NOM_SOMMET NOM_DE_SOMMET,
    FACE ORIENTATION,
    FOREIGN KEY (NOM_SOMMET, FACE)
       REFERENCES PREMIERE(NOM_SOMMET, FACE));
```

Modification ou suppression de tables ou de domaines

En SQL la modification est réalisée par la commande `ALTER` et sa suppression par la commande `DROP`.

`ALTER DOMAIN` *nom_de_domaine définition_de_domaine* ;

`ALTER TABLE` *nom_de_table modification_de_table* ;

modification_de_table ::=

- `ADD COLUMN` *définition_de_colonne*
- `ADD CONSTRAINT` *contrainte_de_table*
- `DROP COLUMN` *nom_de_colonne*
- `DROP CONSTRAINT` *nom_de_contrainte*

`DROP DOMAIN` *nom_de_domaine* [`RESTRICT` | `CASCADE`] ;

`DROP TABLE` *nom_de_table* [`RESTRICT` | `CASCADE`] ;

Exemples

Ajouter l'attribut chef d'expédition à la table PREMIERE

```
ALTER TABLE PREMIERE
   ADD COLUMN CHEF_D_EXPEDITION CHAR(30);
```

Supprimer la table CONQUETE

```
DROP TABLE CONQUETE;
```

Manipulation des données

Interrogation

La syntaxe (simplifiée) d'une requête SQL est définie par les règles suivantes la suivante :

requête ::=

SELECT [DISTINCT] *projection*

FROM liste de (*nom_de_table* [[AS] *nom*]) | (*requête* AS *nom*)

WHERE *condition*

[GROUP BY liste de *nom_de_colonne*

[HAVING *condition*]

[ORDER BY liste de ((*nom_de_colonne* | *rang_de_colonne*) (ASC | DESC)];

requête ::= *requête* (UNION | INTERSECT | EXCEPT) *requête*

requête ::= (*requête*)

L'opérateur SELECT…FROM…WHERE construit une table à partir des tables spécifiées par la clause FROM. Ces tables peuvent être des tables de base ou bien le résultat de requêtes. Une table de base peut être renommée. Une table résultant d'une requête doit être nommée. Les colonnes de la table construite par l'opérateur SELECT…FROM…WHERE peuvent être nommées. Un nom de table ou de colonne est attribué par une clause AS.

projection ::=

`*`

| *nom_de_table*`.*`

| liste de (*terme_de_projection* [`AS` *nom*])

terme_de_projection ::=

expression

| *agregation*

expression ::=

valeur

| *nom_de_colonne*

| *expression_arithmétique*

…

valeur ::= *nombre* | *chaîne_de_caractères*

nom_de_colonne ::= [*nom_de_table*`.`]*nom*

agrégation ::=

`COUNT(*)`

opérateur_d'agrégation`(`[`DISTINCT`] *expression*`)`

opérateur_d'agrégation ::=

(`COUNT` | `SUM` | `AVG` | `MAX` | `MIN`)

condition ::=

condition_élémentaire

| `NOT` *condition*

| *condition* (`AND` | `OR`) *condition*

| (*condition*)

condition_élémentaire ::=

reconnaissance_de_modèle

| *test_de_valeur_nulle*

| *comparaison*

| *appartenance_à_un_intervalle*

| *appartenance_à_un_ensemble*

| *test_d'ensemble_vide*

reconnaissance_de_modèle ::= *expression* [`NOT`] `LIKE` *modèle_de_chaîne*

test_de_valeur_nulle ::= *nom_de_valeur* `IS` [`NOT`] `NULL`

comparaison ::=

terme_de_comparaison (= | <> | > | < | <= | >=) *terme_de_comparaison*

| *expression* (= | <> | > | < | <= | >=) (`ALL` | `SOME`) *requête_mono_colonne*

terme_de_comparaison ::= *expression* | *requête*

appartenance_à_un_intervalle ::=

expression `BETWEEN` *expression* `AND` *expression*

appartenance_à_un_ensemble ::=

expression (`IN` | `NOTIN`) (*ensemble_de_valeurs*)

| (liste de *expression*) (`IN` | `NOTIN`) (*requête*)

| *requête* (`IN` | `NOTIN`) (*requête*)

ensemble_de_valeurs :=

(liste de *valeur*) | *requête_mono_colonne*

test_d'ensemble_vide ::= `EXISTS` (*requête*)

Sémantique de l'instruction SFW

La réponse à une requête

```
SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ... ORDER BY ...
```

est la table T_6 construite de la façon suivante :

1. On réalise le produit cartésien des tables citées dans la clause `FROM`. On obtient une table T_1
2. On restreint T_1 aux lignes qui vérifient la condition de la clause `WHERE`. On obtient une table T_2.
3. On regroupe les lignes de T_2 selon les critères $c_1, \ldots, c_k$ de la clause `GROUP BY`. On obtient une table T_3 dont chaque ligne est de la forme $<v_1, \ldots, v_k, g>$ où g est le sous-ensemble (ou groupe) des lignes de T_2 qui ont la valeur $v_1, \ldots, v_k$ pour les critères $c_1, \ldots, c_k$.
4. On restreint chaque groupe de T_3 aux lignes qui vérifient la condition de la clause `HAVING`. On obtient une table T_4.
5. On projette T_4 sur les colonnes définies dans la clause `SELECT`. Pour chaque ligne, les opérations d'agrégation s'appliquent au groupe g et les autres opérations au n-uplet $<v_1, \ldots, v_k>$. On obtient la table T_5.
6. On trie T_5 selon les critères de la clause `SORT BY`. On obtient la table T_6 réponse à la requête.

Exemples

Requêtes monotables

Noms des sommets ?

```
SELECT NOM_SOMMET
  FROM SOMMET;
```

Premières réalisées en 1953 sur une face sud ou une face nord ?

```
SELECT *
  FROM PREMIERE
 WHERE ANNEE = 1953
   AND (FACE = 'S' OR FACE = 'N');
```

ou bien

```
SELECT *
  FROM PREMIERE
 WHERE ANNEE = 1953
   AND FACE IN ('S', 'N');
```

Nom et altitude en pieds des sommets dont le nom contient le mot "Peak" ?

```
SELECT NOM_SOMMET, 3.29 * ALTITUDE
  FROM SOMMET
 WHERE NOM_SOMMET LIKE '%Peak%';
```

Nom et altitude des sommets dont l'altitude est comprise entre 8100 m et 8500 m ?

```
SELECT NOM_SOMMET, ALTITUDE
  FROM SOMMET
WHERE  ALTITUDE BETWEEN 8100 AND 8500;
```

Jointure et utilisation de synonymes

Donner pour chaque première sur un sommet de plus de 8100m , le nom et l'altitude du sommet et la face gravie ?

```
SELECT SOMMET.NOM_SOMMET, SOMMET.ALTITUDE, PREMIERE.FACE
  FROM PREMIERE, SOMMET
 WHERE PREMIERE.NOM_SOMMET = SOMMET.NOM_SOMMET
   AND SOMMET.ALTITUDE > 8100;
```

ou bien en désignant les tables `PREMIERE` et `SOMMET` par les synonymes `P` et `S`.

```
SELECT S.NOM_SOMMET, S.ALTITUDE, P.FACE
  FROM PREMIERE P, SOMMET S
 WHERE P.NOM_SOMMET = S.NOM_SOMMET
   AND S.ALTITUDE > 8100;
```

Eliminer les doubles : la clause DISTINCT

Nom des grimpeurs ayant conquis le sommet lors d'une première ?

```
SELECT DISTINCT GRIMPEUR
  FROM CONQUETE;
```

Premières sur un sommet népalo-chinois ?

```
SELECT DISTINCT P.NOM_SOMMET, P.FACE
  FROM PREMIERE P, LOCALISATION L
 WHERE P.NOM_SOMMET = L.NOM_SOMMET
   AND L.PAYS IN ('Nepal', 'Chine');
```

Opérateurs d'agrégation (COUNT, AVG, MIN, MAX, …)

Nombre de sommets ?

```
SELECT COUNT(*)
  FROM SOMMET;
```

ou bien à partir de la table `LOCALISATION`

```
SELECT COUNT(DISTINCT NOM_SOMMET)
  FROM LOCALISATION;
```

Altitudes minimale, moyenne et maximale des sommets ?

```
SELECT MIN(ALTITUDE), AVG(ALTITUDE), MAX(ALTITUDE)
  FROM SOMMET;
```

Hauteur totale des sommets conquis par Messner ?

```
SELECT SUM(S.ALTITUDE)
  FROM SOMMET S, CONQUETE C
 WHERE C.GRIMPEUR = 'Messner'
   AND C.NOM_SOMMET = S.NOM_SOMMET;
```

(Un sommet est compté autant de fois qu'il a été conquis).

Sous-requêtes (1) : comparaison entre une valeur et une table

Premières réalisées la même année que la première de la face SO de l'Everest ?

```
SELECT *
  FROM PREMIERE
 WHERE ANNEE = (SELECT ANNEE
                  FROM PREMIERE
                 WHERE NOM_SOMMET = 'Everest'
                   AND FACE = 'SO');
```

Nom des sommets dont l'altitude est supérieure à la moyenne des altitudes de tous les sommets ?

```
SELECT NOM_SOMMET
  FROM SOMMET
 WHERE ALTITUDE > (SELECT AVG(ALTITUDE)
                      FROM SOMMET);
```

Sous-requêtes (2) : IN et NOT IN

Nom des grimpeurs ayant conquis un sommet chinois par sa face N, NE ou NO ?

```
SELECT DISTINCT GRIMPEUR
  FROM CONQUETE
 WHERE NOM_SOMMET IN (SELECT NOM_SOMMET
                        FROM LOCALISATION
                       WHERE PAYS = 'Chine')
   AND FACE IN ('N', 'NE', 'NO');
```

Nom des sommets situés au Népal mais pas sur la frontière avec la Chine ?

```
SELECT NOM_SOMMET
  FROM SOMMET
 WHERE NOM_SOMMET IN (SELECT NOM_SOMMET
                        FROM LOCALISATION
                       WHERE PAYS = 'Népal')
   AND NOM_SOMMET NOT IN (SELECT NOM_SOMMET
                            FROM LOCALISATION
                           WHERE PAYS = 'Chine');
```

Sous-requêtes (3) : quantificateurs ALL et SOME

Nom des sommets dont l'altitude est supérieure à toutes celles des sommets du Pakistan ?

```
SELECT NOM_SOMMET
  FROM SOMMET
 WHERE ALTITUDE >ALL
   (SELECT ALTITUDE
      FROM SOMMET
     WHERE NOM_SOMMET =SOME (SELECT NOM_SOMMET
                               FROM LOCALISATION
                              WHERE PAYS = 'Pakistan'));
```

Sous-requêtes (4) : quantification et universelle

Grimpeurs ayant conquis au moins un sommet conquis par Messner ?

```
SELECT DISTINCT C1.GRIMPEUR
  FROM CONQUETE C1
 WHERE EXISTS (SELECT *
                 FROM CONQUETE C2
                WHERE C2.GRIMPEUR = 'Messner'
                  AND C2.NOM_SOMMET = C1.NOM_SOMMET);
```

Grimpeurs ayant conquis tous les sommets ?

```
SELECT C1.GRIMPEUR
  FROM CONQUETE C1
 WHERE (SELECT NOM_SOMMET
          FROM CONQUETE C2
         WHERE C2.GRIMPEUR = C1.GRIMPEUR)
       =
       (SELECT NOM_SOMMET
          FROM SOMMET);
```

Groupement et agrégation

Donnez pour chaque pays : son nom et la moyenne des altitudes de ses sommets ?

```
  SELECT L.PAYS, AVG(S.ALTITUDE)
    FROM SOMMET S, LOCALISATION L
   WHERE S.NOM_SOMMET = L.NOM_SOMMET
GROUP BY L.PAYS;
```

Donnez pour chaque sommet : son nom, son altitude et le nombre des premières sur ce sommet ?

```
  SELECT S.NOM_SOMMET, S.ALTITUDE, COUNT(*)
    FROM SOMMET S, PREMIERE P
   WHERE S.NOM_SOMMET = P.NOM_SOMMET
GROUP BY S.NOM_SOMMET, S.ALTITUDE;
```

Noms des sommets conquis par plus d'une face en 1975 et nombre de faces gravies ?

```
  SELECT NOM_SOMMET, COUNT(*)
    FROM PREMIERE
   WHERE ANNEE = 1975
GROUP BY NOM_SOMMET
  HAVING COUNT(*) > 1;
```

Tri

Triplets (Pays, Altitude, Nom de sommet) triés par pays puis par altitude décroissante ?

```
  SELECT L.PAYS, S.ALTITUDE, S.NOM_SOMMET
    FROM LOCALISATION L, SOMMET S
   WHERE L.NOM_SOMMET = S.NOM_SOMMET
ORDER BY L.PAYS ASC, S.ALTITUDE DESC;
```

Altitude moyenne des sommets de chaque pays, triés par altitude moyenne ?

```
  SELECT L.PAYS, AVG(S.ALTITUDE)
    FROM LOCALISATION L, SOMMET S
   WHERE L.NOM_SOMMET = S.NOM_SOMMET
GROUP BY L.PAYS
ORDER BY 1;
```

ou bien

```
  SELECT L.PAYS, AVG(S.ALTITUDE) AS ALTITUDE_MOYENNE, L.PAYS
    FROM LOCALISATION L, SOMMET S
   WHERE L.NOM_SOMMET = S.NOM_SOMMET
GROUP BY L.PAYS
ORDER BY ALTITUDE_MOYENNE;
```

Combinaison de requêtes

Nom des sommets de plus de 8500 m situés au Népal mais pas sur la frontière avec la Chine ?

```
(SELECT NOM_SOMMET
   FROM SOMMET
  WHERE ALTITUDE > 8500)
INTERSECT
(SELECT NOM_SOMMET
   FROM LOCALISATION
  WHERE PAYS = 'Népal')
EXCEPT
(SELECT NOM_SOMMET
   FROM LOCALISATION
  WHERE PAYS = 'Chine');
```

Mise à jour d'une table

Insertion de lignes dans une table

`INSERT INTO` *nom_de_table* [(liste de *nom_de_colonne*)]

`VALUES` (liste de *valeur*)

| *requête*;

valeur ::=

expression

| `NULL`

| `USER`

| `CURRENT_DATE`

…

Modification de lignes d'une table

`UPDATE` *nom_de_table*

`SET` liste de *nom_de_colonne* = *expression*

`WHERE` *condition*;

Suppression de lignes d'une table

`DELETE FROM` *nom_de_table*

`WHERE` *condition*;

Exemples

Ajouter la première en solitaire de Messner à la face O du Nanga Parbat en 1978.

```
INSERT INTO PREMIERE
VALUES ('Nanga Parbat', 'O', 1978);

INSERT INTO CONQUETE
VALUES ('Messner', 'Nanga Parbat', 'O');
```

Ajouter une nouvelle table contenant les premières de Messner.

```
CREATE TABLE PREMIERE_DE_MESSNER
   (NOM_SOMMET NOM_DE_SOMMET,
    FACE ORIENTATION);

INSERT INTO PREMIERE_DE_MESSNER
SELECT NOM_SOMMET, FACE
  FROM CONQUETE
 WHERE GRIMPEUR = 'Messner';
```

Convertir en pieds l'altitude de chaque sommet du Pakistan.

```
UPDATE SOMMET
SET ALTITUDE = 3.29 * ALTITUDE
WHERE NOM_SOMMET IN
   (SELECT NOM_SOMMET
      FROM LOCALISATION
     WHERE PAYS = 'Pakistan');
```

Supprimer les premières sur le K2.

```
DELETE
  FROM PREMIERE
 WHERE NOM_SOMMET = 'K2';
```

Index

Un index permet un accès rapide aux lignes d'une relation par rapport aux valeurs de certaines colonnes.

En général on crée des index sur les clés primaires et étrangères des tables intervenant dans des jointures ou sur les colonnes intervenant dans des critères de sélection.

Création d'un index

`CREATE` [`UNIQUE`] `INDEX` *nom_d'_index* `ON`
nom_de_table (liste de *nom_de_colonne*);

Suppression d'un index

`DROP INDEX` *nom_d'_index*;

Exemples

Créer un index unique sur le constituant {SOMMET, FACE} de la table PREMIERE.

```
CREATE UNIQUE INDEX PSF ON PREMIERE(NOM_SOMMET, FACE);
```

Créer un index sur le constituant (SOMMET, FACE) de la table CONQUETE.

```
CREATE INDEX CSF ON CONQUETE(NOM_SOMMET, FACE);
```

Supprimer l'index CSF.

```
DROP INDEX CSF;
```

Vues

Une vue est une table virtuelle résultat d'une requête.

Une vue peut être utilisée dans des requêtes au même titre qu'une table permanente. Par contre, la mise à jour au travers d'une vue n'est réalisable que sous certaines conditions que nous énoncerons.

Création d'une vue

`CREATE VIEW` *nom* [(liste de *nom_de_colonne*)] `AS` *requête*;

Suppression d'une vue

`DROP VIEW` *nom_de_vue*;

Exemples

Créer la table virtuelle PREMIERE_AU_NEPAL des premières sur les sommets du Népal.

```
CREATE VIEW PREMIERE_AU_NEPAL(NOM_SOMMET, ALTITUDE, FACE, ANNEE)
AS SELECT S.NOM_SOMMET, S.ALTITUDE, P.FACE, P.ANNEE
     FROM SOMMET S, PREMIERE P, LOCALISATION L
    WHERE L.PAYS = 'Népal'
      AND S.NOM_SOMMET = L.NOM_SOMMET
      AND S.NOM_SOMMET = P.NOM_SOMMET;
```

Nom des sommets de plus 8500 m dont une face a été gravie pour la première fois en 1953 ?

```
SELECT NOM_SOMMET
  FROM PREMIERE_AU_NEPAL
 WHERE ANNEE = 1953
   AND ALTITUDE > 8500;
```

Supprimer la vue PREMIERE_AU_NEPAL.

```
DROP VIEW PREMIERE_AU_NEPAL;
```

Mise à jour au travers d'une vue

Pour être utilisée pour une mise à jour une vue doit vérifier les conditions suivantes :

- La vue ne doit pas inclure la clause `DISTINCT`.
- Chaque élément de la clause `SELECT` doit être un nom de colonne.
- La clause `FROM` ne doit contenir qu'une seule table, elle-même modifiable.
- La clause `WHERE` ne doit pas contenir de sous-requête.
- La vue ne doit contenir ni clause `GROUP BY`, ni clause `HAVING`.

SQL dans un programme

Trois modes d'utilisation de SQL existent :

- **Mode interactif**.
- **Mode module**

  Un module est une unité compilable séparément composée d'un ensemble de procédures dont chacune est une instruction SQL paramétrée appelable par un programme.
- **Mode intégré**

  Les instructions SQL sont placées directement dans le texte d'un programme hôte, C par exemple. Le code ainsi obtenu est traité par un précompilateur.

Nous étudions le mode intégré.

Intégration de SQL dans un programme

Les principes sont les suivants :

- Chaque instruction SQL est préfixée par `EXEC SQL`.
- Ces instructions peuvent contenir des variables du programme qui :
  - seront préfixées par :
  - devront avoir un type compatible avec les valeurs d'attributs qui leur seront affectées.
- Un mécanisme de curseur permet de parcourir une table ligne par ligne et d'affecter les valeurs de chaque ligne à des variables du programme.
- Le programme doit posséder une variable numérique `SQLCODE` dont la valeur résume l'exécution, correcte ou incorrecte, d'une instruction SQL.

Déclarations

Déclaration des variables utilisées dans les instructions SQL

EXEC SQL BEGIN DECLARE SECTION

liste de déclaration_de_variables

EXEC SQL END DECLARE SECTION

Définition d'un curseur

EXEC SQL DECLARE *nom_de_curseur* CURSOR

FOR *requête*;

Instructions exécutables

Manipulation d'un curseur

```
EXEC SQL OPEN nom de curseur;

EXEC SQL FETCH nom de curseur INTO liste de variables;

EXEC SQL
   UPDATE …
      SET …
    WHERE CURRENT OF nom_de_curseur;

EXEC SQL
   DELETE
     FROM …
    WHERE CURRENT OF nom_de_curseur;

EXEC CLOSE nom de curseur
```

Construction d'une table d'une seule ligne

```
EXEC SQL
     SELECT …
        INTO liste de variables
        FROM …
      WHERE …
   GROUP_BY …
     HAVING …;
```

Exemples

Affecter à la variable `une_annee` *l'année de la première ascension de la face* `une_face` *de l'Everest ?*

```
EXEC SQL BEGIN DECLARE SECTION
    char une_face[2];
    int une_annee;
EXEC SQL END DECLARE SECTION
…
EXEC SQL
    SELECT ANNEE
      INTO :une_annee
      FROM PREMIERE
     WHERE NOM_SOMMET = 'Everest'
       AND FACE = :une_face;
…
```

Augmenter l'altitude de chaque sommet du Népal d'une valeur `incr` *calculée par programme ?*

```
EXEC SQL BEGIN DECLARE SECTION
    char sommet[20];
    int altitude, incr;
EXEC SQL END DECLARE SECTION
    int SQLCODE;
EXEC SQL DECLARE s CURSOR FOR
    SELECT NOM_SOMMET, ALTITUDE
      FROM SOMMET
     WHERE NOM_SOMMET IN
        (SELECT NOM_SOMMET
          FROM LOCALISATION
         WHERE PAYS = 'Népal');
EXEC SQL OPEN s;
EXEC SQL FETCH s INTO :nom, :altitude;
while(SQLCODE = 0) {
    Calcul de incr …
    EXEC SQL
        UPDATE SOMMET
        SET ALTITUDE = ALTITUDE + :incr
        WHERE CURRENT OF SOMMET;
    EXEC SQL FETCH s INTO :nom, :altitude;
}
EXEC SQL CLOSE s;
```

Maintien de l'intégrité

Expression de l'intégrité

Il existe plusieurs façons d'exprimer les contraintes d'intégrité d'une BD :

- définition du domaine ou du type d'un attribut,
- condition sur les valeurs des attributs d'un n-uplet (clause `CHECK` de SQL),
- définition de la clé primaire et des clés étrangères d'une relation,
- assertions générales,
- déclencheurs.

Nous avons déjà étudié les trois premières.

Une **assertion** est une formule logique qui doit être vraie quelque soit l'extension de la BD. La vérification des assertions est une opération coûteuse, car cette vérification doit être faite après chaque mise à jour de la BD. Il existe des techniques permettant de minimiser ce coût, mais elles sont complexes à mettre en œuvre.

Un **déclencheur** est une règle, dite **active**, de la forme :

"événement - condition - action"

L'action est déclenchée à la suite de l'événement et si la condition est vérifiée. Une action peut être une vérification ou une mise à jour.

Ce chapitre est consacré à l'expression des assertions et des déclencheurs en SQL.

Assertions

Une assertion est créée par la commande SQL :

`CREATE ASSERTION` nom `CHECK` (condition)

Soit par exemple la BD relationnelle décrivant des employés et les départements dans lesquels ils travaillent :

```
EMPLOYE(NUM_EMP, NUM_DEPT, SALAIRE)
DEPARTEMENT(NUM_DEPT, NOM_DEPT, NB_EMP)
```

La contrainte d'intégrité : "Tout employé du département 'Recherche' a un salaire supérieur à 10 000, s'exprime par l'assertion suivante :

```
CREATE ASSERTION CHECK
   (NOT EXISTS
      (SELECT *
         FROM EMPLOYE
        WHERE SALAIRE <= 10000
          AND NUM_DEPT =
                (SELECT NUM_DEPT
                   FROM DEPARTEMENT
                  WHERE NOM = 'Recherche')))
```

Déclencheurs

En SQL un déclencheur est activé par une requête de mise à jour.

Pour définir un déclencheur, il faut :

- spécifier l'événement qui déclenche l'action en indiquant le type de la mise à jour (`INSERT`, `UPDATE`, `DELETE`), le nom de la relation et éventuellement le nom des attributs mis à jour ;
- indiquer si l'action est réalisée avant, après ou à la place de la mise à jour ;
- donner un nom à l'ancien et au nouveau n-uplet (uniquement le nouveau en cas d'insertion et uniquement l'ancien en cas de suppression) ;
- décrire la condition sous laquelle se déclenche l'événement sous la forme d'une expression SQL booléenne, c.-à-d. une expression pouvant être placée dans une clause `WHERE` ;
- décrire l'action à réaliser sous la forme d'une procédure SQL ;
- indiquer si l'action est réalisée pour chaque n-uplet mis à jour ou une seule fois pour la requête.

Création d'un déclencheur

commande ::=

`CREATE TRIGGER` *nom*

instant_de_déclenchement

événement

[*nommage*]

[*condition*]

action`;;`

instant_de_déclenchement ::=

`BEFORE` | `AFTER` | `INSTEAD OF`

événement ::=

(`INSERT` | `DELETE` | `UPDATE` [`OF` liste de *nom_de_colonne*])

`ON` *nom_de_table*

nommage ::=

`REFERENCING` (`OLD AS` *nom* | `NEW AS` *nom*)

condition ::=

`WHEN` *expression_booléenne*

action ::=

(liste de *commande*)

[`FOR EACH` (`ROW` | `STATEMENT`)]

Exemples

Création d'un déclencheur qui signale les mises à jour de salaire qui sont des diminutions et les annihilent.

```
CREATE TRIGGER DIMINUTION_SALAIRE
BEFORE UPDATE OF SALAIRE ON EMPLOYE
REFERENCING OLD AS A NEW AS N
WHEN N.SALAIRE < A.SALAIRE
(ERROR)
FOR EACH ROW;
```

Création d'un déclencheur mettant automatiquement à jour le nombre d'employés d'un département à la suite de l'insertion d'un nouvel employé.

```
CREATE TRIGGER INCREMENTER_NB_EMP
AFTER INSERT ON EMPLOYE
REFERENCING NEW AS E
(UPDATE DEPARTEMENT
    SET NB_EMP = NB_EMP + 1
  WHERE NUM_DEPT = E.NUM_DEPT)
FOR EACH ROW;
```

Confidentialité

Objectifs et moyens

Une BD doit être protégée des accès malveillants.

La solution adoptée classiquement consiste à n'**autoriser** un utilisateur à effectuer une **opération** sur un **objet** que s'il en a obtenu le **droit** (ou le **privilège**).

L'identification des utilisateurs se fait en général par leur nom et par un mot de passe. Des procédés plus sophistiqués peuvent être utilisés comme les cartes à puce ou la reconnaisssance des empreintes digitales ou rétiniennes.

Dans un SGBD relationnel, les objets à protéger sont les tables et les vues et les opérations sont la consultation par une requête, l'insertion, la modification ou la suppression de n-uplets.

L'attribution des droits peut être :

- centralisé : l'administrateur a tous les droits sur tous les objets de la BD et peut transmettre certains de ces droits à d'autres utilisateurs.
- décentralisée : l'utilisateur qui crée un objet a tous les droits sur cet objet et peut les transmettre en totalité ou en partie à d'autres utilisateurs.

Protection par les vues

Les vues jouent un rôle important pour la confidentialité en permettant de spécifier de façon très fine les données auxquels un utilisateur a le droit d'accéder.

Soit, par exemple, la BD composée des deux relations suivantes :

```
EMPLOYE(NOM_EMP, NUM_DEPT, SALAIRE)
DEPARTEMENT(NUM_DEPT, NOM_DEPT, DIRECTEUR)
```

On pourra spécifier un accès à l'affectation des employés dans les départements en définissant la vue suivante :

```
CREATE VIEW AFFECTATION(NOM_EMP, NOM_DEPT) AS
   SELECT E.NOM, D.NOM
     FROM EMPLOYE AS E, DEPARTEMENT AS D
    WHERE E.NUM_DEPT = D.NUM_DEPT
```

Un utilisateur dont les droits d'accès se limite à la vue `AFFECTATION` ne pourra connaître ni le numéro et le salaire d'un employé, ni le numéro et le directeur d'un département.

Attribution des autorisations en SQL

Un utilisateur peut transmettre des privilèges à un autre utilisateur par la commande `GRANT` et il peut lui en retirer par la commande `REVOKE`.

La syntaxe de ces deux commandes est la suivante :

`GRANT` (`ALL PRIVILEGES` | liste de *opération*)
`ON` *nom_de_table* | *nom_de_vue*
`TO` (`PUBLIC` | *identificateur_d'utilisateur*)
[`WITH GRANT OPTION`]

`REVOKE` (`ALL PRIVILEGES` | liste de *opération*)
`ON` *nom_de_table* | *nom_de_vue*
`FROM PUBLIC` | *identificateur_d'utilisateur*

opération ::=
(`INSERT` | `DELETE` | `UPDATE` [liste de *nom_de_colonne*])

La clause `WITH GRANT OPTION` transmet le droit de transmettre le privilège.

Exemple

Supposons qu'une utilisatrice ait créé les relations `EMPLOYE` et `DEPARTEMENT` ainsi que la vue `AFFECTATION`. Elle pourra :

- Transmettre à l'utilisateur Jean Dupont le droit de consulter les employés et de modifier le salaire d'un employé, par la commande suivante :

  ```
  GRANT SELECT, UPDATE(SALAIRE) ON EMPLOYE TO 'JEAN DUPONT'
  ```

- Transmettre à tous les utilisateurs le droit de consulter les affectations des employés, par la commande suivante :

  ```
  GRANT SELECT ON AFFECTATION TO PUBLIC
  ```

- Retirer à Jean Dupont le droit de modifier le salaire d'un employé, par la commande suivante :

  ```
  REVOKE UPDATE(SALAIRE) ON EMPLOYE TO 'JEAN DUPONT'
  ```

  (Notons que Jean Dupont possède encore le droit de consulter les employés.)

Organisation physique

Introduction

Rappelons qu'une BD relationnelle est constituée d'un ensemble de relations qui ont chacune une extension qui est un ensemble de n-uplets.

Physiquement ces n-uplets sont stockés dans un ou plusieurs fichiers qui peuvent être répartis sur un ou plusieurs sites (dans le cas de BD distribuées). Le format de stockage choisi doit permettre : une utilisation optimale de la mémoire, un accès rapide et des mises à jour faciles.

Les n-uplets manipulés par une application doivent être préalablement transférés en mémoire centrale. Ce transfert est réalisé au travers d'une zone particulière de la mémoire centrale appelée "tampon" ou "cache". Pour minimiser les temps de calcul, il y a intérêt, tant que cela est possible, à maintenir en mémoire centrale les n-uplets qui sont fréquemment manipulés.

Afin d'accéder le plus rapidement possible à un ensemble de n-uplets vérifiant certaines conditions, toute BD relationnelle est munie d'index qui permettent d'associer chaque valeur d'un constituant à l'ensemble des n-uplets qui possèdent cette valeur pour ce constituant.

Fichiers

Un **fichier** est stocké sur un disque. Il peut s'étendre sur un ou plusieurs **secteurs** de celui-ci.

Un fichier possède un nom et est constitué par une suite de **blocs**.

L'espace occupé par un fichier varie au cours du temps. On distingue quatre phases :

1. L'**allocation initiale** : des blocs sont alloués au fichier lors de sa création.
2. L'**expansion** : des blocs supplémentaires sont alloués au fur à mesure des besoins. Cette allocation peut être faite :
   - bloc par bloc, à partir d'une pile de blocs libres,
   - par groupe de blocs contigus : cela permet de regrouper physiquement des n-uplets souvent traités ensemble (ceux d'une relation ou bien ceux possédant des valeurs de clés identiques).
3. La **contraction** : les blocs qui ne sont plus utilisés sont restitués. Cette restitution peut être faite :
   - bloc par bloc sur la pile de blocs libres,
   - par groupe de blocs libres contigus : cela permet d'éviter l'éparpillement de la mémoire.
4. La **réorganisation** : on récupère les espaces mémoire libres qui ont pu apparaître au cours des phases 2 et 3.

Gestion des n-uplets

Les n-uplets d'une BD relationnelle sont rangés dans des pages.

Une **page** est stockée dans un bloc de fichier. Nous supposerons qu'un bloc contient une seule page.

L'adresse physique d'un n-uplet est un quadruplet (s, f, b, r) où :

- s est un identificateur de site,
- f est un identificateur de fichier,
- b et r sont des entiers positifs ou nuls.

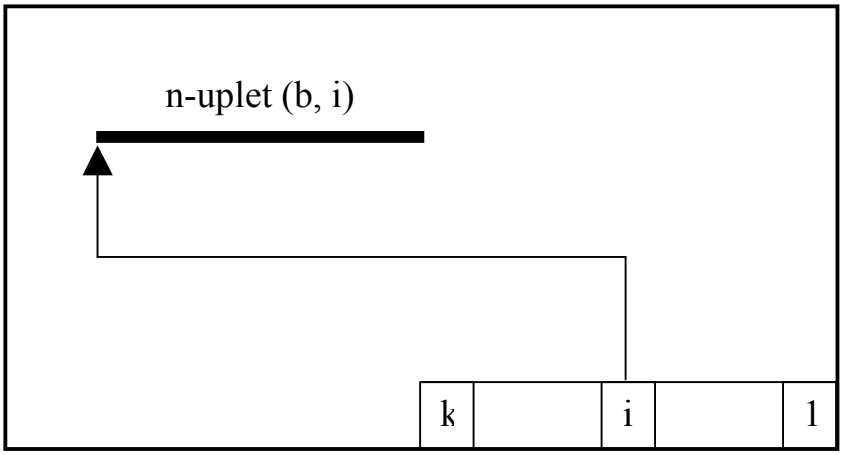
Cette adresse désigne le r^e n-uplet de la page enregistrée dans le b^e bloc du fichier f du site s.

Nous nous placerons dans le cas simple où la BD est mono-site et mono-fichier. Dans ce cas, l'adresse d'un n-uplet se réduit à un doublet (b, r).

Organisation d'une page

Une page est découpée en deux espaces :

1. La zone des n-uplets : les n-uplets y sont implantés dans un ordre quelconque.
2. Un répertoire qui est implanté à partir de la fin de la page. La i^e case de ce répertoire indique le déplacement dans la page du n-uplet de rang i.



page stockée dans un bloc b et contenant k n-uplets

Cette organisation permet une certaine évolution de la taille des n-uplets d'une page sans qu'il soit nécessaire de modifier leurs adresses.

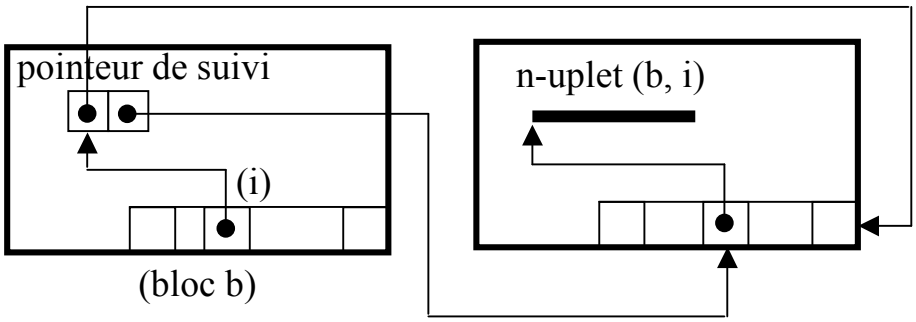
Identification d'un n-uplet

Il est nécessaire d'identifier chaque n-uplet afin :

- de les distinguer des autres n-uplets,
- de pouvoir y accéder à partir des index.

Trois méthodes d'identification peuvent être utilisées :

1. Par **adressage direct** : l'identificateur d'un n-uplet est son adresse physique. Cette méthode impose la mise en place d'un pointeur de suivi dans le cas où un n-uplet doit être changé de page à la suite d'une mise à jour augmentant sa taille.



2. Par **adressage** indirect : l'identificateur d'un n-uplet est un nombre entier attribué à sa création. Cette méthode nécessite la mise en place d'une table de correspondance pour passer de l'identificateur d'un n-uplet à son adresse physique.
3. Par la **clé primaire**. Un index primaire permet d'assurer la correspondance entre la clé et l'adresse physique du n-uplet.

Performances

La 1ère méthode est la plus efficace :

- elle nécessite 1 accès disque pour accéder à un n-uplet, s'il n'y a pas de pointeur de suivi et 2 accès disque s'il y en a un.
- elle ne nécessite pas de table de correspondance.

Représentation d'un n-uplet

Valeurs d'attributs

Elles peuvent être représentées :

- sous leur forme externe,
- sous la forme sous laquelle elles sont utilisées par les programmes d'application.

n-uplet courts (taille inférieure à une page)

Les deux formats possibles sont les suivants :

- format fixe :
  - valeurs d'attributs enregistrées dans des champs de longueur fixe.
- format variable :
  - valeurs d'attributs précédées de leur longueur,
  - valeurs d'attributs précédées du nom de l'attribut.

Le choix d'un format doit prendre en compte les critères suivants :

- possibilité de représenter des attributs de longueur variable,
- évolutivité des schémas de relation par ajout ou suppression d'attributs,
- rapidité d'accès aux valeurs d'attributs.

n-uplet longs (taille supérieure à une page)

Ils sont fragmentés en plusieurs pages.

Cette fragmentation peut être réalisée en séparant les attributs longs stockés chacun dans une page différente, des attributs courts stockés dans la même page.

Echange disque - mémoire centrale

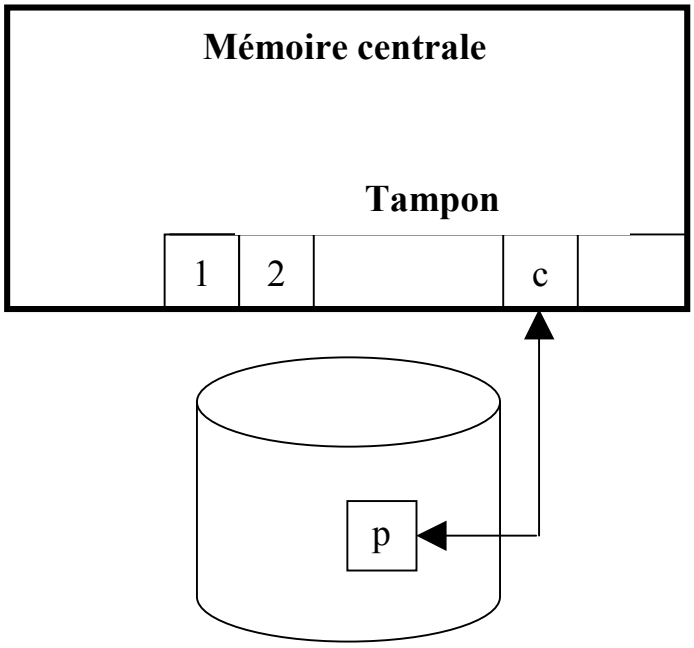
Les échanges de données entre disque et mémoire centrale se font au travers d'une zone particulière de la mémoire appelée **tampon** ("buffer" en anglais) et sont pris en charge par le gestionnaire de tampon.

Les principaux objectifs du gestionnaire de tampon sont :

- de rendre les pages et les n-uplets qu'elles contiennent accessibles en mémoire centrale,
- de coordonner l'écriture des pages sur le disque en coopération avec le gestionnaire de transactions,
- de minimiser le nombre d'accès disques.

Structure d'un tampon

Un tampon est formé d'une suite de **cases** contigues dont chacune peut contenir une page. En mémoire centrale une page est donc repérée par le numéro de la case du tampon dans laquelle elle est rangée.



Recherche d’une page

L’opération de recherche d’une page :

- a pour argument l’adresse p de la page cherchée,
- retourne le numéro c de la case du tampon dans laquelle cette page est rangée.

Cette opération se déroule selon les étapes suivantes :

1. Si la page p est dans la case c du tampon, retourner c. **On économise un accès disque**.
2. Si la page p n’est pas dans le tampon, il faut la lire sur le disque. On teste s’il existe une case libre pour la recevoir. Si oui, soit c cette case.
3. Sinon, il faut libérer une case et donc renvoyer une page du tampon sur le disque. Deux stratégies sont posibles : renvoyer la page la moins récemment utilisée (“LRU”) ou la page la plus récemment utilisée (“MRU”). Soit c la case contenant cette page.
4. Si la page à rejeter a été modifiée pendant son séjour en mémoire centrale, il faut la réécrire sur le disque. Ce travail est pris en compte par le gestionnaire de transactions.
5. Transférer la page p du disque dans la case c et retourner c.

Index

Un **index** est un fichier qui contient un ensemble d'enregistrements formés d'une clé et d'une information associée et qui permet un accès direct aux enregistrements dont la clé :

- est donnée,
- appartient à un intervalle de valeurs donné.

Dans une BD relationnelle, l'information associée est en général un identificateur de n-uplet ou bien un ensemble d'identificateurs de n-uplets, selon que l'index est primaire ou secondaire :

- Un **index primaire** est construit sur la clé primaire d'une relation. Il donne accés à l'identificateur du n-uplet dont la clé est donnée.
- Un **index secondaire** est construit sur un constituant quelconque d'une relation, par exemple l'attribut `Altitude` de la relation `Sommet`. Il donne accès aux identificateurs des n-uplets dont la valeur pour ce constituant est donnée.

Un exemple

Dans la suite nous considèrerons à titre d'exemple, un index primaire construit sur la clé de la relation suivante qui décrit les entrées d'un dictionnaire (le Larousse de Poche).

Dictionnaire	
Mot	**Définition**
bateau	Nom des embarcations, des navires autres que les navires de guerre…
corde	Assemblage de fils tressés ou tordus ensemble…
dessin	Représentation sur une surface de la forme d'un objet ou d'une figure…
école	Etablissement où se donne un enseignement collectif…
kayak	Embarcation étanche et légère, manœuvrée à la pagaie double.
mélodie	Suite de sons formant un air…
nez	Partie saillante du visage, entre la bouche et le front…
terre	Planète habitée par l'homme…
zébu	Bœuf à longues cornes et à bosse sur le garrot…

Classification des index

- Accès mono-critère :
  - Arbres :
    - séquentiel-indexé,
    - arbres B.
  - Accès par hachage
    - statique,
    - dynamique.
- Accès multi-critères :
  - Arbres :
    - arbres k-d,
    - arbres R.
  - Accès par hachage :
    - hachage partitionné.

Arbres B^+

Historique

Les arbres B ont été introduits par Bayer et McCreight en 1972 et ont fait l'objet de nombreux développements par la suite.

Nous décrivons une variante des arbres B : les arbres B^+, qui sont en général utilisés dans les BD.

Organisation

Un arbre B^+ d'ordre e_{max} (entier impair ≥ 3) est un arbre équilibré dont chaque nœud est une page stockée sur disque.

- Une feuille contient une séquence d'enregistrements :

  $(c_1\ a_1) \ldots (c_n\ a_n)$

  triés par ordre croissant de clé. Une feuille est au moins à moitié remplie :

  $(e_{max} + 1) / 2 \leq n \leq e_{max}$,

  sauf si elle est l'unique nœud de l'index.

- Les feuilles sont chaînées entre elles dans l'ordre de leur première clé.

- Un nœud non terminal contient une séquence de couples clé-pointeur :

  $([c_1]\ p_1) \ldots (c_m\ p_m)$

  triée par ordre croissant de clé et dans laquelle :

  - pour i de 1 à m - 1, p_i est un pointeur vers la racine du sous-arbre dont les feuilles contiennent les enregistrements dont la clé est égale ou supérieure à c_i et inférieure à c_{i+1}.
  - p_m est un pointeur vers la racine du sous-arbre dont les feuilles contiennent les enregistrements dont la clé est égale ou supérieure à c_m.
  - la clé c_1 est virtuelle : elle n'est pas enregistrée car elle n'est pas nécessaire pour la recherche d'une clé.

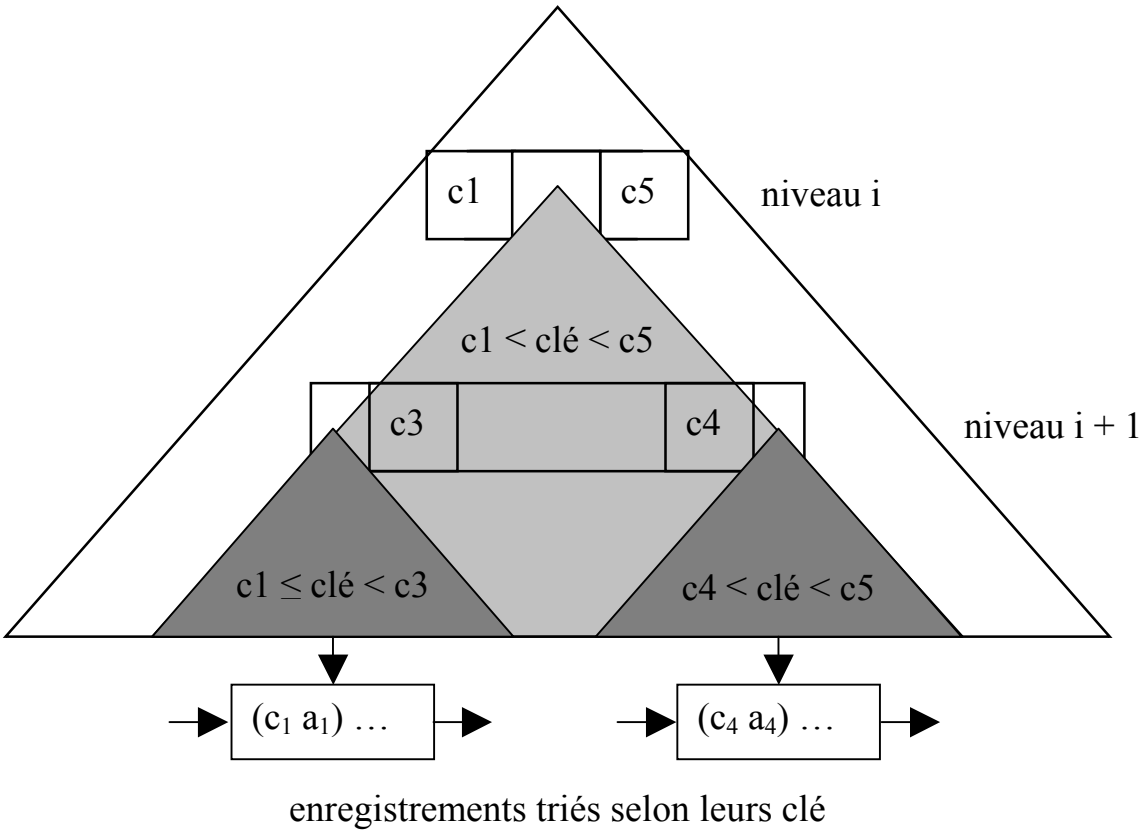
  Une nœud non terminal est au moins à moitié rempli

  $(e_{max} + 1) / 2 \leq m \leq e_{max}$

  sauf s'il est la racine de l'index, auquel cas son contenu peut se réduire à :

  $([c_1]\ p_1) \ldots (c_2\ p_2)$.

Un arbre B^+ peut être schématisé de la façon suivante :



Opérations

A sa création un arbre B^+ est réduit à un nœud unique dont le contenu est vide.

Recherche d'un enregistrement

(Il s'agit de rechercher l'enregistrement dont la clé c est donnée ou, s'il n'existe pas, le nœud qui devrait le contenir.)

Le nœud courant est la racine de l'arbre B^+.

Tant que le nœud courant est un nœud non terminal de contenu

(p_1) $(c_2, p_2)\ldots$ (c_n, p_n) **répéter**

  Si $c < c_2$ **alors**

    Accéder au nœud p_1 qui devient le nœud courant.

  sinon

    Rechercher séquentiellement le dernier c_i inférieur ou égal à c

    et accéder au nœud p_i qui devient le nœud courant.

  fin.

fin.

Rechercher l'enregistrement de clé c dans le nœud courant.

<u>La recherche est terminée</u>.

Insertion d'un enregistrement

(Il s'agit d'insérer l'enregistrement de clé c et d'information associée a.)

Rechercher l'enregistrement de clé c.

S'il existe **alors**

Le signaler.

sinon

Le nœud courant est celui sur lequel s'est arrêté la recherche et l'item à insérer est (c, a).

Répéter

Soit p le nœud courant, i l'item à insérer et s le contenu de p.

Insérer i dans s en respectant l'ordre des clés.

Si longueur(s) $\leq e_{max}$ **alors**

Enregistrer s dans p.

L'insertion est terminée.

sinon

Fission du nœud courant.

fin.

jusqu'à ce que l'insertion soit terminée**.**

fin.

(La longueur d'une séquence est le nombre d'enregistrements qui la composent.)

Fission du nœud courant

(Il s'agit de répartir sur deux nœuds le nouveau contenu s du nœud courant p.)

Créer un nouveau nœud p'.

Découper s en deux séquences s_1 et s_2 de longueur égale : c'est possible, car la longueur de s est $e_{max} + 1$ qui est un nombre pair.

Soit c_{21} la première clé de s_2.

Enregistrer s_1 dans p.

Enregistrer s_2 dans p' après avoir supprimé sa première clé, si p est un nœud non terminal.

Si p est une feuille **alors**

Insérer p' à la suite de p dans la chaîne des feuilles.

fin.

Si p est la racine **alors**

Créer un nouveau nœud de contenu (p) (c_{21} p').

<u>L'insertion est terminée</u> (la hauteur de l'index a augmenté de 1).

sinon

Le père de p devient le nœud courant et (c_{21} p') devient l'item à insérer.

fin.

Suppression d'un enregistrement

(Il s'agit de supprimer l'enregistrement dont la clé c est donnée.)

Rechercher l'enregistrement de clé c.

S'il n'existe pas **alors**

Le signaler.

sinon

Le nœud courant est celui qui contient l'enregistrement de clé c.

L'item à supprimer est l'enregistrement de clé c.

Répéter

Soit p le nœud courant, s le contenu de p et i l'item à supprimer.

Supprimer i de s.

Si longueur(s) $\geq (e_{max} + 1) / 2$ ou p est la racine **alors**

Enregistrer s dans p.

La suppression est terminée.

sinon

Fusion du nœud courant.

fin.

jusqu'à ce que la suppression soit terminée**.**

fin.

Fusion du nœud courant

(Il s'agit de fusionner le nouveau contenu s du nœud courant avec le contenu de son frère gauche ou celui de son frère droit.)

Si la fusion se fait avec le frère gauche du nœud courant **alors**

Soit p_2 le nœud courant et p_1 le frère gauche de p.

Ré-associer sa clé au premier pointeur de s, si p_2 est un nœud non terminal.

Soit s' la concaténation du contenu de p_1 avec s.

sinon (la fusion se fait avec le frère droit du nœud courant)

Soit p_1 le nœud courant et p_2 le frère droit de p.

Ré-associer sa clé au premier pointeur du contenu de p_2, si p_2 est un nœud non terminal.

Soit s' la concaténation de s avec le contenu de p_2.

Si longueur(s') $\leq e_{max}$ **alors**

Enregistrer s' dans p_1.

Supprimer p_2.

Si p_1 est fils unique **alors**

Supprimer le père de p_1.

<u>La suppression est terminée</u> (la hauteur de l'index a diminué de 1).

fin.

Le père de p_2 devient le nœud courant et l'item de ce nœud qui pointe vers p_2 devient l'item à supprimer.

sinon

Découper s' en deux séquences s'_1 et s'_2

de longueurs égales à un item prêt.

Enlever sa clé au premier item de s'_2, si p_2 est un nœud non terminal.

Enregistrer s'_1 dans p_1 et s'_2 dans p_2.

Remplacer dans la page père de p_2, la clé de l'item qui pointe vers p_2,

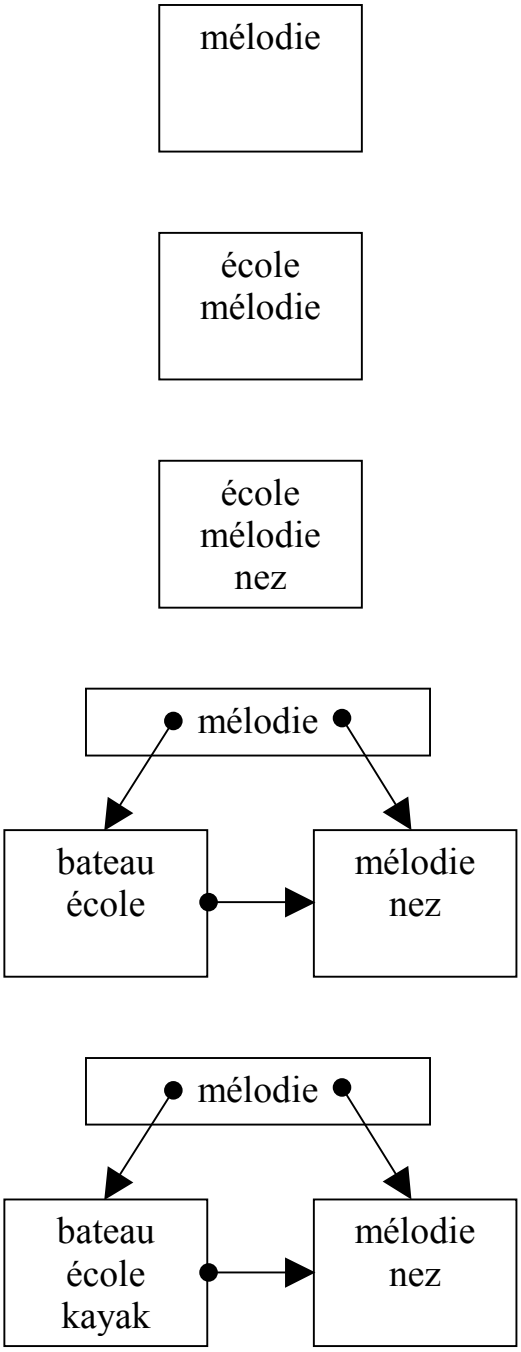
par la première clé de s'_2.

<u>La suppression est terminée</u>.

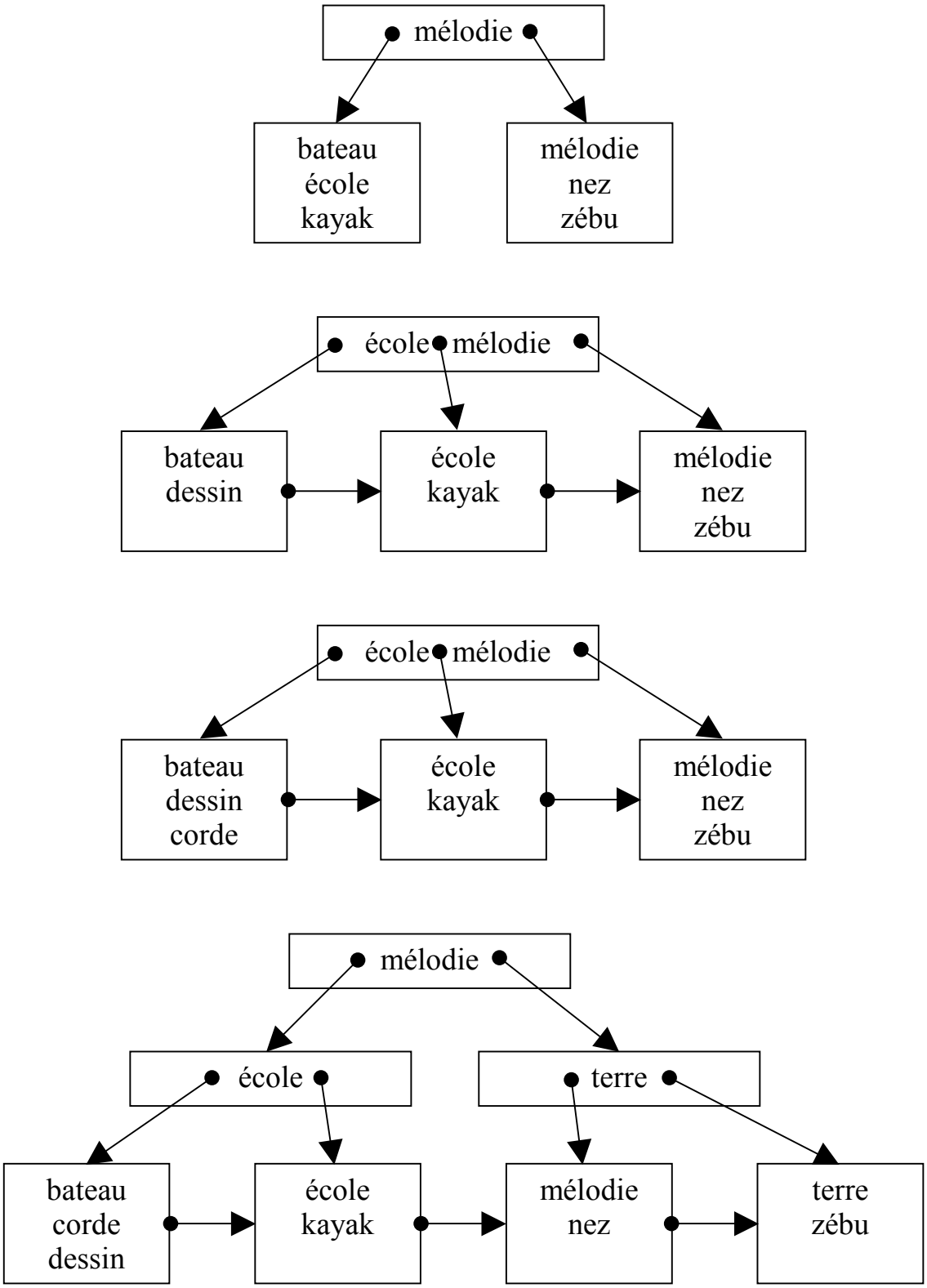
fin.

Exemple : Arbre B^+ d'ordre $e_{max} = 3$

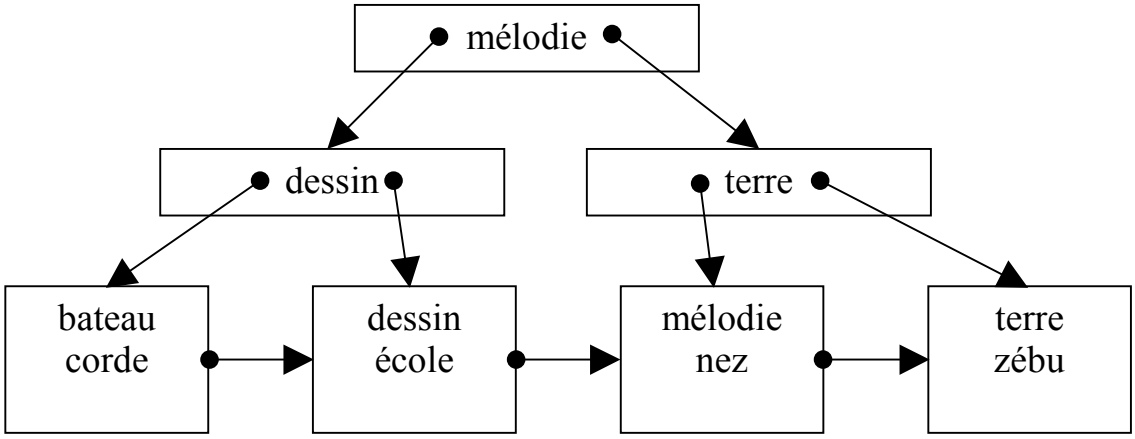
Insertions successives des enregistrements : mélodie, école, nez, bateau, kayak.

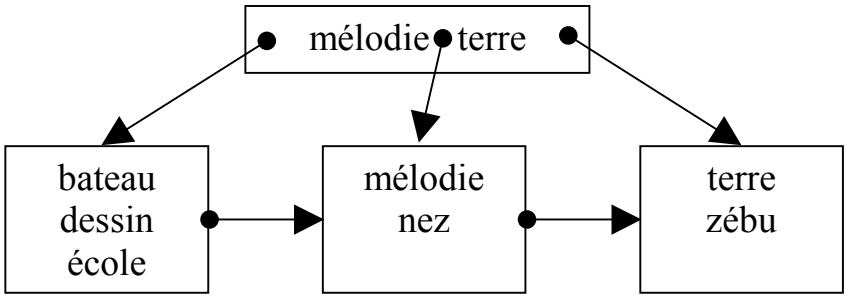


Insertions successives des enregistrements : zébu, dessin, corde et terre.



Suppressions successives des enregistrements : kayak et corde.





Performances

On suppose que l'index est primaire et contient N enregistrements et l'on pose $n = e_{max} - 1$.

- Le nombre maximum de nœuds de chaque branche (c'est à dire la hauteur de l'arbre) est $\log_{n/2}(N)$. La recherche d'un enregistrement dont la clé est donnée nécessite donc de parcourir $\log_{n/2}(N)$ nœuds et donc autant d'accès disque, puisque chaque nœud est une page stockée sur disque.
- Si l'on suppose :
  - que les pages ont une taille de 4000 octets,
  - qu'une clé a une longueur de 32 octets et qu'un pointeur a une taille de 8 octets..
  - qu'il y a 10^6 d'enregistrements dans l'index

  alors $n = 100$ et $\log_{50}(10^6) = 4$. Il faut donc 4 accès disque pour retrouver un enregistrement. Dans le cas habituel où la racine réside en mémoire centrale, 3 accès disque suffisent.
- Il est montré de plus, que le coût d'une insertion ou d'une suppression est proportionnel à $\log_{n/2}(N)$, c'est à dire à la hauteur de l'arbre.

Hachage statique

Un index à accès par hachage peut être organisé de deux façons différentes :

- avec répertoire,
- sans répertoire.

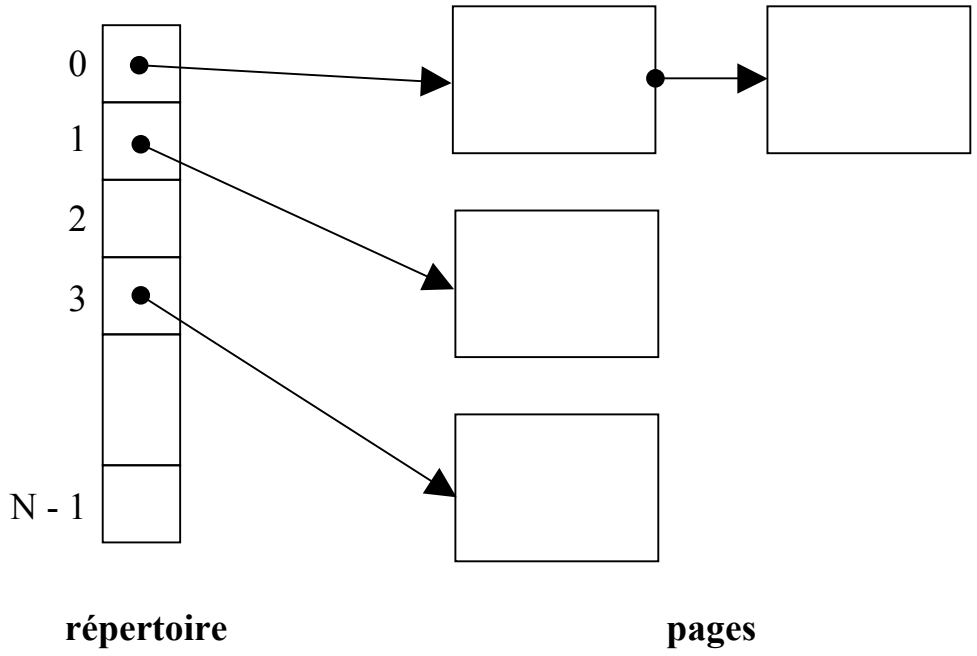
Nous présentons en détail l'organisation avec répertoire puis, sur un exemple, l'organisation sans répertoire.

Organisation

Un index à accès par hachage statique avec répertoire est un quadruplet (N, H, P, R) où :

- N est un nombre entier positif.
- H est une **fonction de hachage** qui appliquée à une clé produit un nombre entier compris entre 0 et N - 1, appelé **code haché** ("hash-code").
- P est un ensemble de pages, stockées sur disque, contenant des enregistrements dont chacun à la forme (c a) où :
  - c est la clé,
  - a est l'information associée.
- R est un répertoire de N cases. Chaque case i ($0 \leq i \leq N - 1$) est le début d'une chaîne (éventuellement vide) de pages dont tous les enregistrements sont tels que H(c) = i.

Cette organisation est résumée par la figure suivante :



Fonction de hachage

La fonction de hachage s'applique à une clé, que nous supposerons être une chaîne de caractères, et à un nombre entier N. Elle retourne un nombre entier compris entre 0 et N - 1.

Dans le cas général, où N est petit devant le nombre qui serait obtenu en convertissant la clé dans un système de numération ayant pour base le nombre de lettres de l'alphabet, il n'est pas possible de construire une fonction de hachage injective, c'est à dire, telle que :

$$c_1 \neq c_2 \Rightarrow H(c_1) \neq H(c_2)$$

On construit donc une fonction qui minimise le nombre de collisions et les répartit uniformément.

Les deux techniques les plus utilisées sont la division et le pliage. Dans les deux cas, on construit à partir des codes des caractères de la clé c, un nombre k grand devant N. La valeur de H(c) est alors calculée comme suit :

- **Division**. H(c) est le reste de la division de k par N. Il est montré que l'on a intérêt à choisir N premier.
- **Pliage.** On découpe la représentation binaire de k en tranches de b bits. H(c) est égal au "ou exclusif" des nombres binaires ainsi obtenus.

Opérations

Recherche d'un enregistrement

(Il s'agit de rechercher l'enregistrement dont la clé c est donnée.)

Parcourir les pages liées à la case H(c) jusqu'à trouver une page qui contienne un enregistrement de clé c : c'est l'enregistrement recherché.

Si le bout de la chaîne est atteint **alors**

L'enregistrement recherché n'existe pas.

fin.

Insertion d'un enregistrement

(Il s'agit d'insérer un nouvel enregistrement de clé c et d'information associée a).

Rechercher l'enregistrement de clé c.

S'il existe **alors**

Le signaler.

sinon

Parcourir les pages liées à la case H(c) jusqu'à en trouver une qui posséde une place suffisante pour le nouvel enregistrement.

S'il en existe une **alors**

Y insérer l'enregistrement : l'insertion est terminée.

sinon (le bout de la chaîne est atteint)

Créer une nouvelle page.

L'ajouter au bout de la chaîne des pages liées à la case H(c).

Y insérer l'enregistrement.

fin.

fin.

Suppression d'un enregistrement

(Il s'agit de supprimer l'enregistrement de clé c.)

Rechercher l'enregistrement de clé c.

S'il n'existe pas **alors**

    Le signaler.

sinon

    Supprimer cet enregistrement de la page qui le contient.

    Si cette page devient vide **alors**

        La supprimer et remettre à jour la chaîne des pages.

    fin.

fin.

Exemple

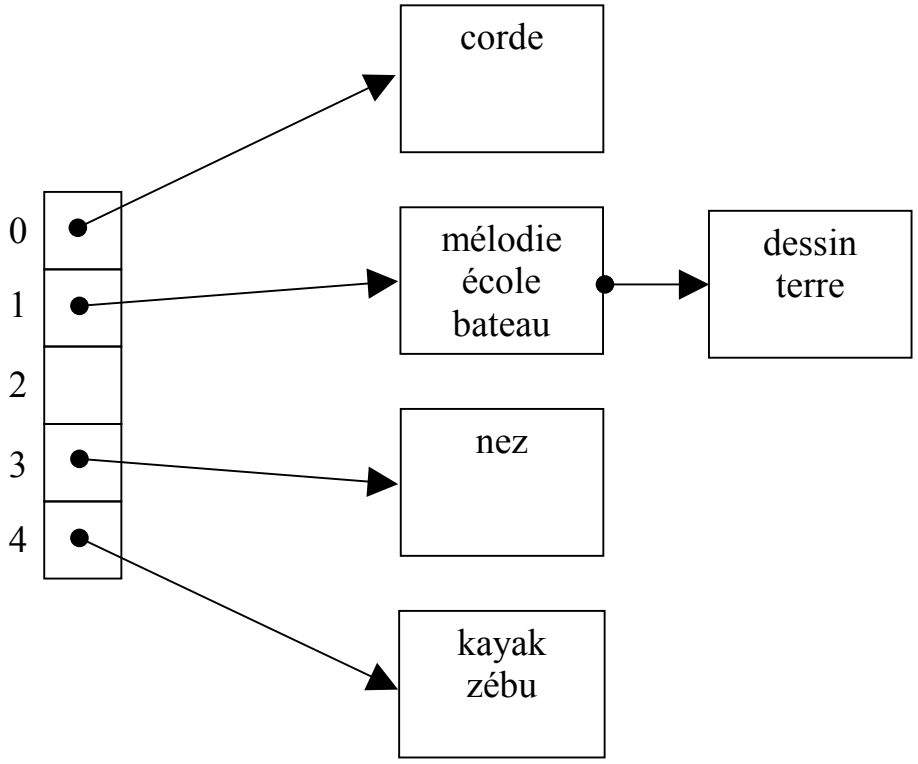
On construit l'index primaire de la relation `Dictionnaire` avec les paramètres suivants :

- N = 5 : répertoire de 5 cases,
- pages de 3 enregistrements au maximum,
- H(c) = (somme des codes ASCII des caractères de c) modulo 5.

Le tableau suivant donne les codes hachés des mots du dictionnaire.

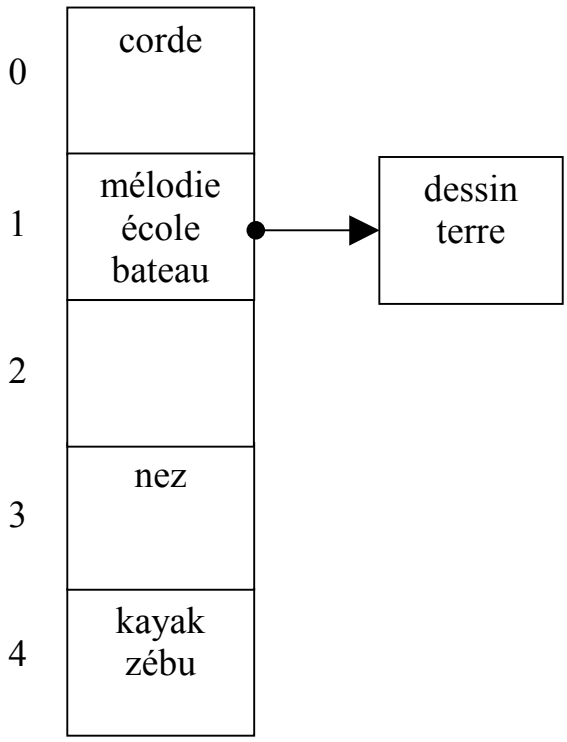
c	**h(c)**
mélodie	1
école	1
nez	3
bateau	1
kayak	4
zébu	4
dessin	1
corde	0
terre	1

Après insertion des mots dans l'ordre du tableau précédent, l'index primaire de la relation `Dictionnaire` a la forme suivante :



Organisation sans répertoire

On peut éviter l'utilisation d'un répertoire en créant un index de N pages contigues. Le code haché donnera alors directement accès à la page contenant les enregistrements recherchés. L'index primaire de la relation `Dictionnaire` est alors le suivant :



Performances

- Sans répertoire et avec un bon ajustement des valeurs de N et du nombre d'enregistrements par page, on peut accéder à un enregistrement en 1,1 ou 1,2 accès disque en moyenne.
- On peut obtenir les mêms temps avec un répertoire pourvu que celui-ci tienne en mémoire centrale.
- Le répertoire permet de ne pas associer inutilement une page à un code-haché non instancié (2 dans notre exemple). Par contre, un répertoire occupe de la place et s'il ne tient pas en mémoire centrale un accés disque supplémentaire sera nécessaire.
- Le hachage statique est mal adapté à un index très évolutif car la taille du répertoire peut s'avérer sous-évaluée, entraînant un accroissement du nombre de collisions et donc du nombre d'accès disque. Pour y palllier, des méthodes de hachage dites dynamiques ou extensibles ont été proposées, qui font évoluer la fonction de hachage en fonction du nombre d'enregistrements.

Résolution des requêtes

Introduction

L'accès à une BD relationnelle est réalisé au travers d'un langage de requêtes (celui de SQL, le plus souvent). Les performances de l'évaluateur de requêtes sont donc cruciales.

Rappelons qu'un langage de requêtes est déclaratif : l'utilisateur exprime sa requête mais pas la façon d'y répondre. Or pour une requête donnée, il existe en général plusieurs stratégies pour construire la réponse. C'est au SGBD de choisir la stratégie optimale.

La résolution d'une requête se déroule en trois phases :

1. **Traduction** de la requête en un arbre d'opérateurs relationnels (sélection, jointure, projection, etc.) que nous appelerons **arbre de requête**.
2. **Optimisation**. Il s'agit de déterminer un plan d'exécution de coût minimal. Deux procédures sont principalement mises en œuvres :
   - Transformation de l'arbre de requête en un arbre équivalent, basée sur les propriétés d'associativité et de commutativité de l'algèbre relationnelle.
   - Evaluation du coût de résolution de chaque opérateur.
3. **Evaluation** du plan d'exécution produit par la phase 2.

Ce chapitre est consacré à l'optimisation d'une requête SQL.

L'optimisation d'une requête SQL est un problème complexe, notamment si l'on prend en compte les opérateurs de tri et de regroupement et les requêtes imbriquées. Ce chapitre est restreint à l'étude de l'optimisation de requêtes composées d'un simple bloc `SELECT…FROM…WHERE`.

Résolution des opérateurs relationnels

Méthodes

Sélection

- boucle (parcours séquentiel),
- utilisation d'un index primaire pour retrouver un n-uplet dont la clé est donnée ou appartient à un intervalle donné,
- utilisation d'un index secondaire pour retrouver un ensemble de n-uplets dont la clé est donnée ou appartient à un intervalle donné.

Jointure

- Sur un critère quelconque :
  - double boucle.
- Equi-jointure :
  - Pas d'index sur les constituants de jointure :
    - double boucle,
    - tri-fusion,
    - hachage.
  - Index sur l'un des constituants de jointure :
    - parcours séquentiel de la relation non indexée et recherche dans l'index des n-uplets à joindre.

Projection

- boucle,
- tri puis boucle, si les doubles doivent être éliminés.

Coût de quelque opérateurs

Paramètres

t_L	temps de lecture d'une page
t_E	temps d'écriture d'une page
lp	longueur (en octets) d'une page
card(R)	cardinalité de la relation R
lv(R, X)	longueur moyenne du constituant X de la relation R
ll(R) = lv(R, {A_1, …, A_n})	longueur moyenne des n-uplets de la relation R(A_1, …, A_n)
nlp(R) = ceil(lp / ll(R))	nombre moyen de n-uplets par page pour la relation R (on suppose qu'une page ne contient que des n-uplets d'une même relation)
nbp(R) = ceil(card(R) / nlp(R))	nombre de pages occupées par la relation R
nvd(R, X)	nombre de valeurs différentes du constituant X de la relation R
hi(R, X)	hauteur de l'index en arbre B^+ sur le constituant X de la relation R

Avec ceil(x) = plus petit entier supérieur ou égal à x.

Sélection

S = **sel**(R, f)

Méthode	**Coût**
boucle	$nbp(R)\times t_L+nbp(S)\times t_E$
f = λp (p.X = v) index en arbre B^+ sur X	$(hi(X,R)+\frac{card(R)}{nvd(X,R)})\times t_L+nbp(S)\times t_E$

Jointure

J = **join**(R_1, R_2, f)

Méthode	**Coût**
double boucle	$nbp(R_1)\times(1+nbp(R_2))\times t_2+nbp(J)\times t_L$
f = λj (j.X = j.Y) (equi-jointure avec index en arbre B^+ sur Y)	$(nbp(R_1)+card(R_1)\times(hi(R_2,Y)+\frac{card(R_2)}{nvd(R_2,Y)}))\times t_L+nbp(J)\times t_E$

Projection

P = **proj**(R, X)

Méthode	**Coût**
boucle	$nbp(R)\times t_L+\frac{card(R)\times lv(R,X)}{lp}\times t_E$

Evaluation de la taille des résultats

La principale difficulté dans le calcul du coût d'une sélection ou d'une jointure est d'estimer la taille des résultats. Elle nécessite de connaître, entre autres :

- La distribution des valeurs des constituants de chaque relation. En l'absence de connaissances plus précise, on suppose une distribution uniforme.
- Le nombre de valeurs distinctes d'un constituant.
- Le facteur de sélectivité des opérations booléennes (comparaisons et connexions logiques)

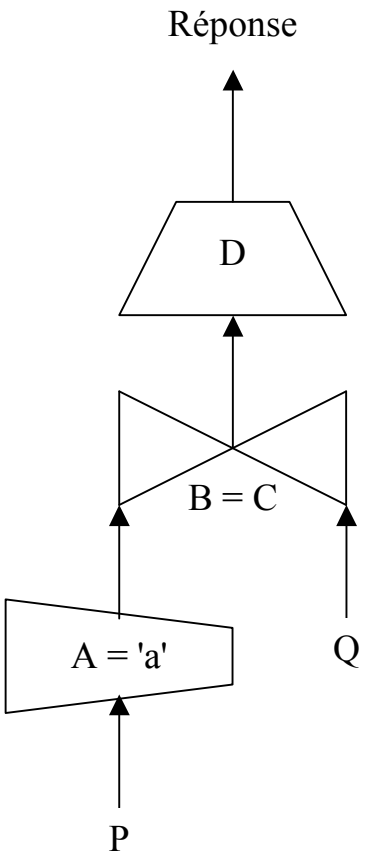
Arbre de requête

Un arbre de requête est un arbre dont :

- les feuilles sont les relations sur lesquelles portent la requête,
- les nœuds non terminaux sont des opérateurs de l'algèbre relationnelle,
- la racine est la relation qui constitue la réponse à la requête.

Plusieurs arbres de requête peuvent être associés à une même requête. Ils sont dits équivalents.

Par exemple, l'arbre de requête :

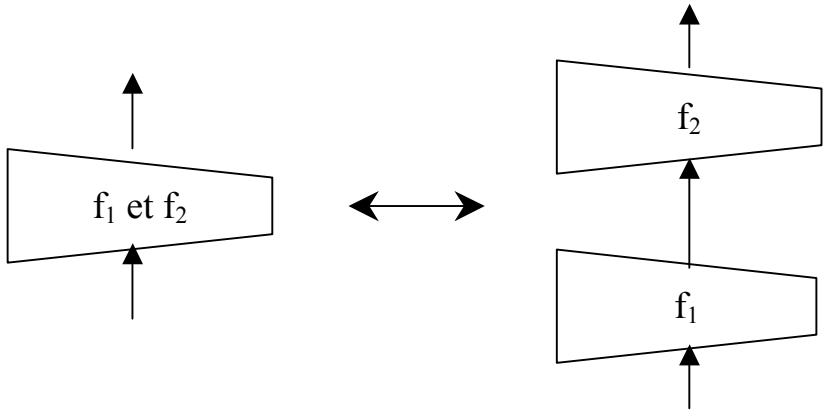


est l'un des arbres de requête possible pour la requête :

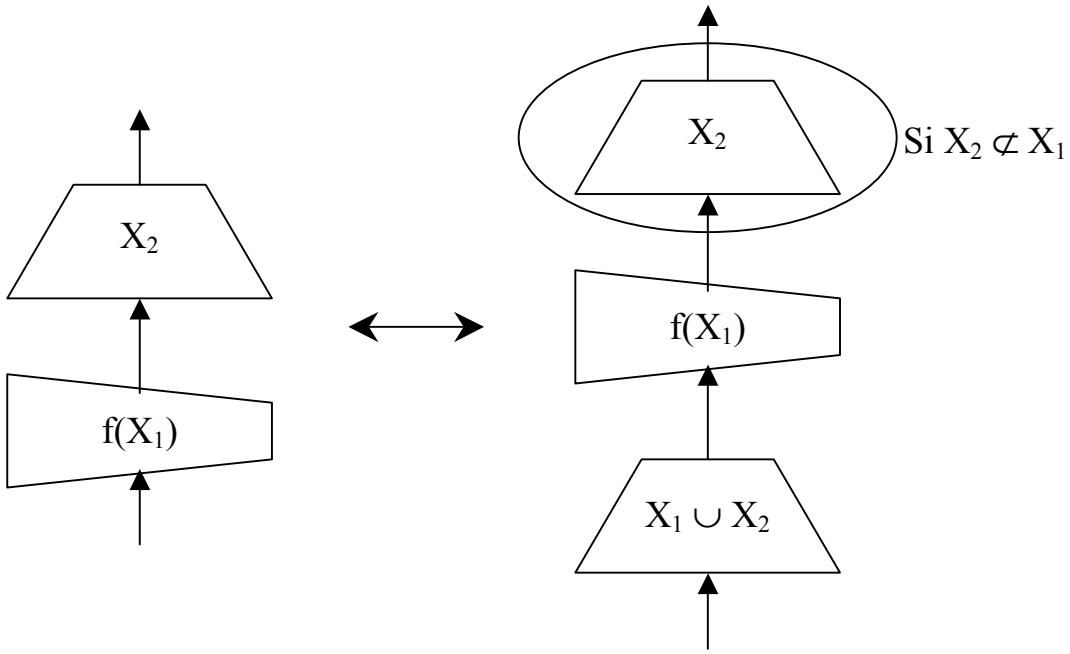
```
SELECT Q.D
  FROM P, Q
 WHERE P.A = 'a'
   AND P.B = C.D
```

Propriétés des opérateurs relationnels

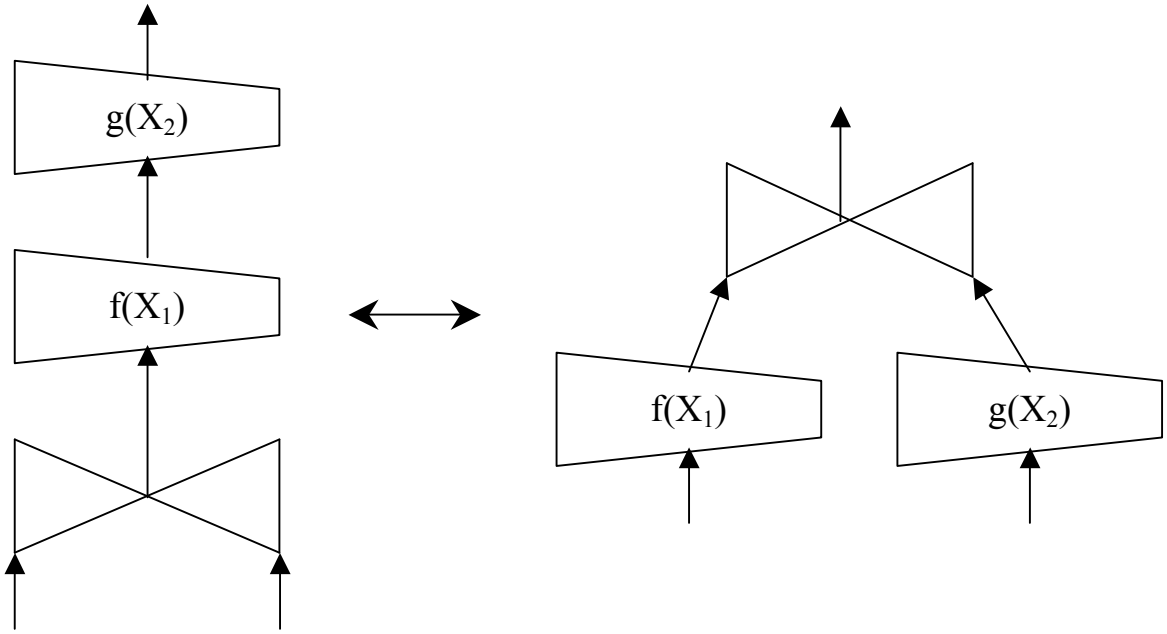
T_1 : Eclatement et regroupement des sélections



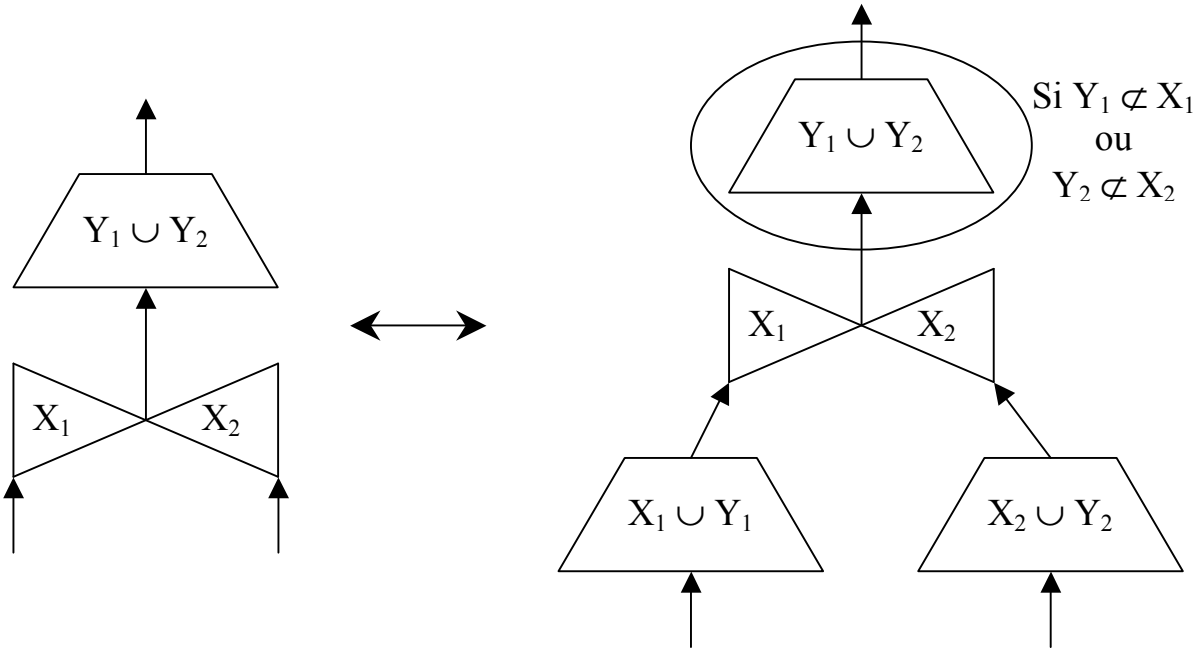
T_2 : Permutation projection-sélection



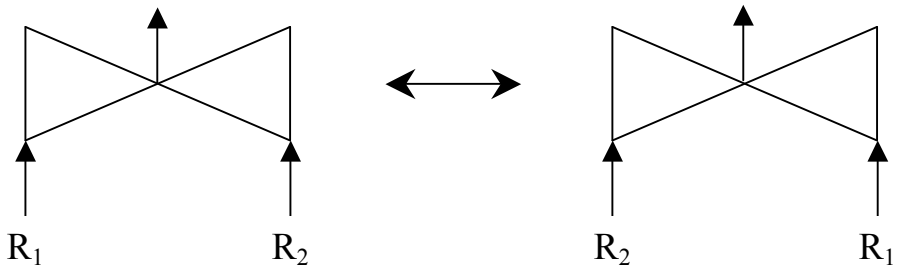
T_3 : Permutation sélection - jointure



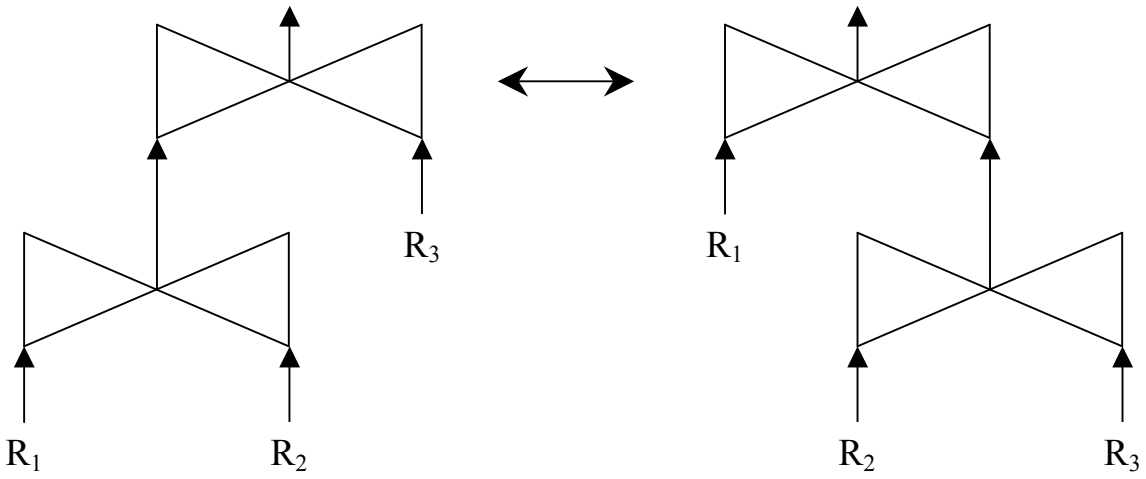
T_4 : Permutation projection - jointure



T_5 : Commutativité de la jointure



T_6 : Associativité de la jointure



Choix d'un plan d'exécution

En théorie, le choix d'un plan d'exécution peut être effectué de la façon suivante :

1. Traduire la requête sous forme d'un arbre d'opérateurs relationnels.
2. Construire l'ensemble des arbres équivalents en appliquant les propriétés de commutativité et d'associativité des opérateurs.
3. A partir de chacun de ces arbres générer tous les plans d'exécution possibles en associant à chaque opérateur chacune des méthodes de résolution applicable.
3. Evaluer le coût de chaque plan d'exécution.
4. Choisir le plan de coût minimal.

L'inconvénient majeur de cette méthode est que le nombre de plans d'exécution à examiner croit très vite avec le nombre de relations sur lesquelles portent la requête.

Par exemple, on peut montrer que le nombre d'ordres de jointures de n relations est :

$$\frac{(2 \times (n-1))!}{(n-1)!}$$

Ce nombre croit donc très vite quand n augmente. Il est égal à 12 pour n = 3, 1680 pour n = 5, 665280 pour n = 7, etc.

En pratique, on utilise des règles heuristiques pour diminuer le nombre de plans d'exécution à évaluer.

Descente des sélections et projections

L'objectif de cette méthode d'optimisation est de minimiser la cardinalité des relations temporaires générées pendant la résolution d'une requête.

On considère des requêtes construites uniquement à partir :

- de sélections dont la condition est une disjonction de comparaisons attribut-valeur,
- de projections,
- de jointures par comparaison de valeurs d'attributs.

Le plan d'exécution est généré de la façon suivante :

1. Mettre la requête sous la forme d'une suite de jointures, suivie d'une sélection, suivie d'une projection.
2. Décomposer les sélections en sélections unaires en appliquant la transformation T_1.
3. Descendre les sélections aussi bas que possible en appliquant les transformations T_2 et T_3, puis regrouper celles portant sur les mêmes relations en appliquant la transformation T_1.
4. Descendre les projections sous les jointures à l'aide de la transformation T_4.
5. Eliminer les projections inutiles.

Exemple

Soit le schéma relationnel suivant :

```
BIBLIOTHEQUE(NOBIB, NOM, VILLE)
LIVRE(ISBN, TITRE, ANNEE, MATIERE)
POSSEDE(COTE, NOBIB, ISBN)
EMPRUNTE(NOM, COTE, DATE)
```

et la requête suivante :

```
SELECT E.NOM
  FROM EMPRUNTE E, POSSEDE P, LIVRE L, BIBLIOTHEQUE B
 WHERE L.ANNEE > 1985
   AND B.VILLE = 'Marseile'
   AND E.COTE = P.COTE
   AND P.ISBN = L.ISBN
   AND P.NOBIB = B.NOBIB
```

qui extrait les noms des emprunteurs des livres publiés aprés 1985 et possédés par une bibliothèque de Marseille ?

Paramètres de calcul de coût

- Les temps d'accès à une page, en lecture et en écriture sont supposés égaux. Le coût d'une opération est mesuré en nombre d'accès à une page. La longueur des pages est de 2000 octets.
- Taille des relations de base :

Relation	Attribut	lv	ll	nlp	card
EMPRUNTE	NOM	20	45	44	10000
	COTE	15			
	DATE	10			
POSSEDE	COTE	15	32	63	300000
	NOBIB	2			
	ISBN	15			
LIVRE	ISBN	15	79	25	50000
	TITRE	50			
	ANNEE	4			
	MATIERE	10			
BIBLIOTHEQUE	NOBIB	2	37	54	30
	NOM	20			
	VILLE	15			

- Sélectivité :

sel(`EMPRUNTE.DATE = '15/10/1996'`) = 4

sel(`LIVRE.ANNEE > 1985`) = 20

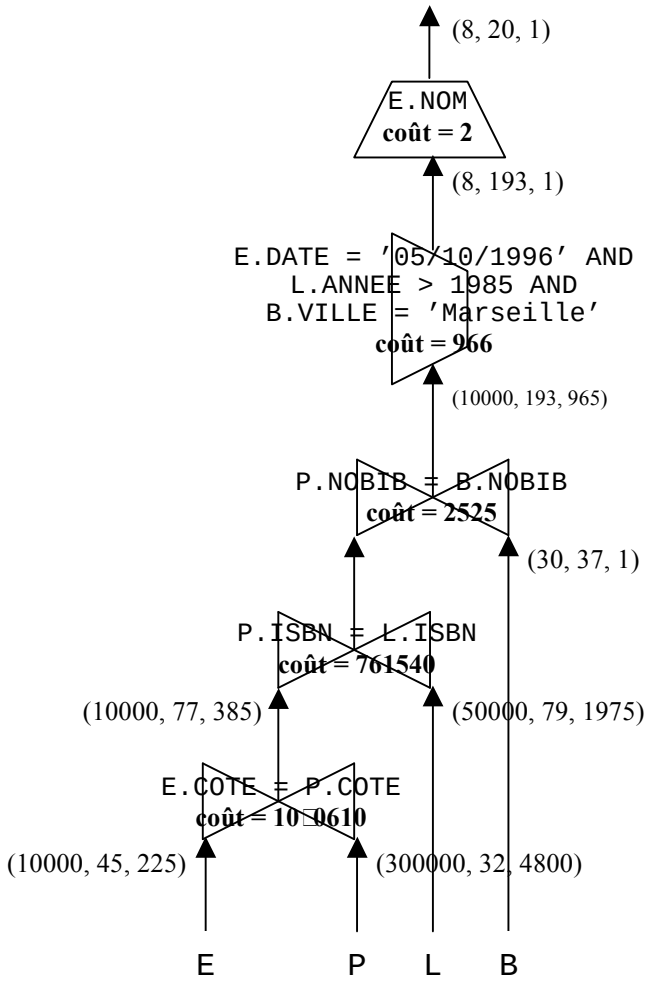
sel(`BIBLIOTHEQUE.NOM = 'Marseille'`) = 10

Avant optimisation

Le volume d'une relation est mesuré par un triplet :

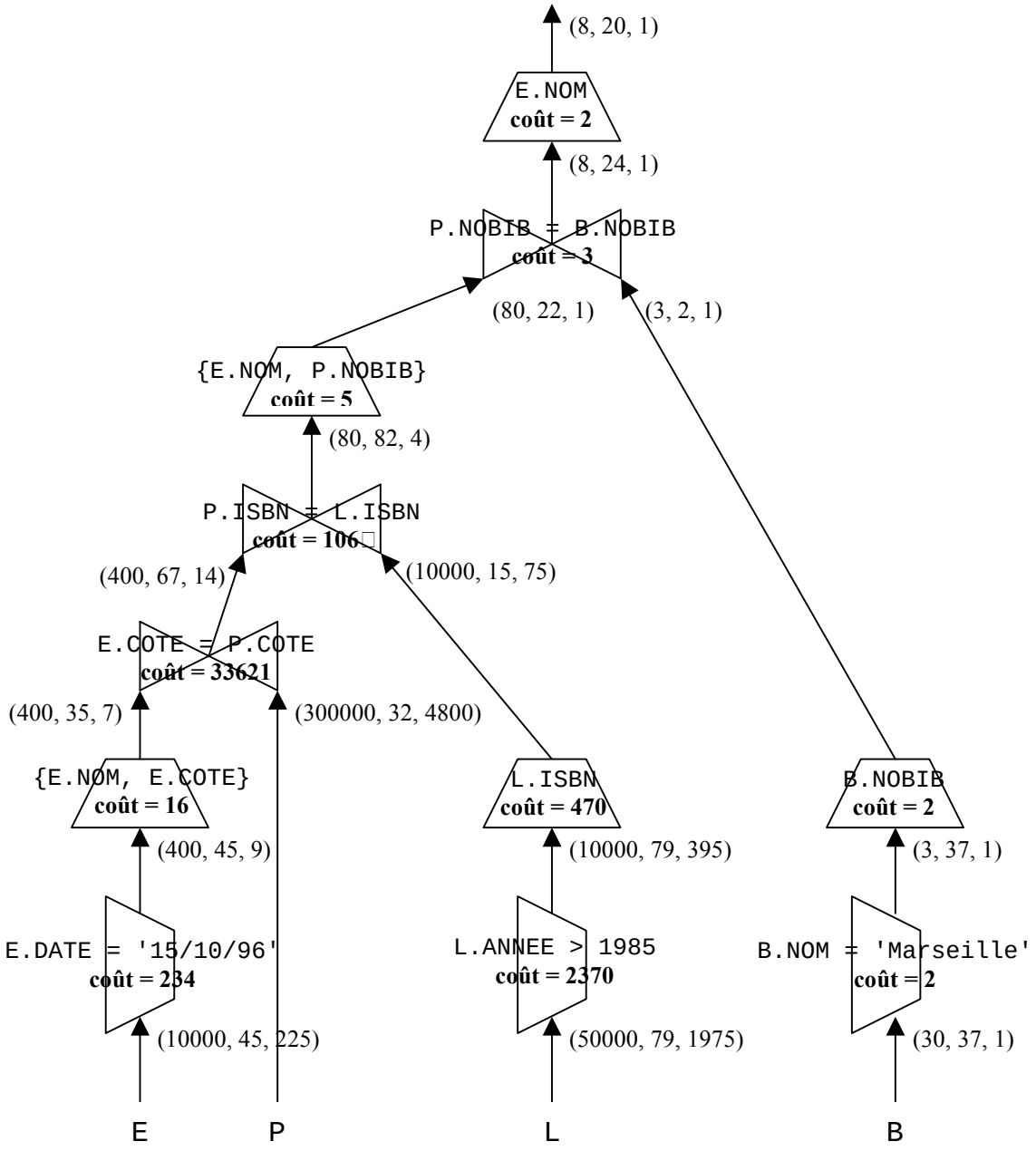
(cardinalité, longueur d'un n-uplet en octets, nombre de pages)

Coût total = 1 845 643 accès à une page.



Après optimisation

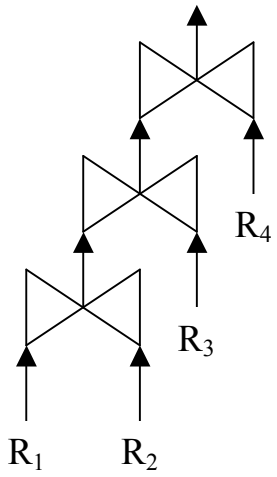
Coût total = 37 793 accès à une page.



Ordres de jointure

La méthode d'optimisation précédente ne tient pas compte de l'ordre de jointure. Or trouver un ordre de jointure optimal est le point clé de l'optimisation d'une requête relationnelle.

Nous avons vu que le nombre d'ordres de jointure croissait très fortement avec le nombre de relations à joindre. Pour minimiser le nombre d'ordres de jointure à considérer on peut se restreinde aux **ordres linéaires gauche de jointure** ("**left-deep join orders**") qui sont tels que chaque jointure a pour opérande gauche une relation de base :



Notons que pour n relations il y a n! ordres linéaires gauche de jointure. Ce qui est beaucoup moins que le nombre total d'ordres de jointure. Par exemple, il y 1680 ordres de jointure de 5 relations dont 120 ordres linéaires gauches.

Un ordre linéaire gauche de jointure est intéressant lorsqu'il existe un index sur l'attribut de jointure ou en cas de résolution "pipeline".

On peut de plus éliminer les ordres de jointure qui contiennent un produit cartésien, car cette opération a en général un coût élévé.

Recherche du meilleur plan d'exécution

Pour rechercher le meilleur plan d'exécution ou tout du moins un plan qui s'en rapproche, chaque SGBD a sa propre méthodes. Voici trois méthodes classiques :

- On construit un seul plan d'exécution basé sur un ordre linéaire gauche de jointure, dont les jointures sont classées par ordre de sélectivité décroissante. Cette méthode peut être combinée avec la descente des sélections et des jointures.
- On construit autant de plans d'exécution qu'il y a de relations. Chaque plan est basé sur un ordre linéaire gauche de jointure et commence avec une relation différente. L'ordre de jointure est construit itérativement en choisissant à chaque étape, la meilleure relation à joindre parmi les relations restantes, c.-à-d. celle pour laquelle le coût de jointure avec la jointure déjà construite est minimum. On choisit ensuite le plan de coût minimum.
- On recherche le meilleur plan par des méthodes de programmation dynamique.

Concurrence et reprise

Concept de transaction

Une **transaction** est un fragment de programme dont l'exécution fait passer une BD d'un état cohérent à un autre état cohérent.

Une transaction est une suite d'**événements** dont chacun peut être : soit une opération sur une **donnée**, soit le démarrage ou l'arrêt de la transaction.

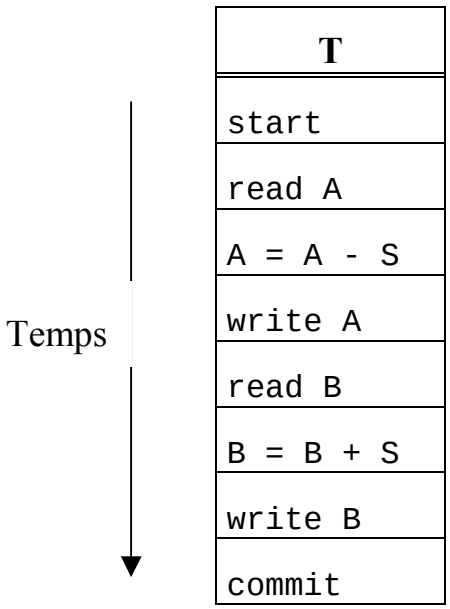
Les seules opérations considérées par un gestionnaire de transaction sont les opérations de **lecture** et d'**écriture** d'une donnée et nous le verrons, le **verrouillage** ou le **déverrouillage** d'une donnée.

Une donnée est un fragment d'une BD : une valeur d'attribut, un n-uplet, une table, etc.

Nous considérerons tout d'abord les types d'événement suivants :

événement	**signification**
`start`	démarrage de la transaction
`read D`	lecture d'une donnée D
`write D`	écriture d'une donnée D
`rollback`	annulation de la transaction
`commit`	confirmation de la transaction

Exemple de transaction : Transfert d'une somme S d'un compte A vers un compte □.



Propriétés d'une transaction

Plusieurs transactions peuvent se dérouler en même temps : on dit qu'elles sont **concurrentes**.

Un SGBD doit assurer que toute transaction possède les propriétés suivantes (dites propriétés **ACID**).

Atomicité — Soit toutes les modifications effectuées par une transaction sont enregistrées dans la BD, soit aucune ne l'est. Si une transaction est confirmée (`commit`) toutes les modifications qu'elle a effectuées sont enregistrées dans la BD et rendues visibles aux autres utilisateurs. Si une transaction est interrompue alors aucune de ces modifications n'est enregistrée dans la BD.

Cohérence — Une transaction fait passer une BD d'un état cohérent à un autre état cohérent. Un état cohérent est un état dans lequel les contraintes d'intégrité sont vérifiées.

Isolation — Une transaction se déroule sans être pertubée par les transactions concurrentes : tout se passe comme si elle se déroulait seule.

Durabilité — Une fois qu'une transaction a été confirmée le SGBD garantit qu'aucune modification qu'elle a effectuée ne sera perdue quelque soient les accidents qui surviendront : interruption, pannes du système d'exploitation, "crash" de disque, etc.

Problémes dus à la concurrence

Si aucun contrôle du déroulement des transactions n'est mis en place, plusieurs problémes peuvent se rencontrer :

- perte de mise à jour,
- lecture impropre :
  - lecture de données incohérentes,
  - lecture de données non confirmées,
- lecture non reproductible.

Perte de mise a jour

T_1	T_2	BD
		A = 10
read A		
	read A	
A = A + 10		
write A		A = 20
	A = A + 50	
	write A	A = 60

Après l'écriture de A par T_2 on devrait avoir A = 70 puisque T_1 a ajouté 50 à A et T_2 lui a ajouté 10. Mais l'ordre des opérations est tel que A = 60.

Lecture impropre (données incohérentes)

T_1	T_2	BD
		contrainte A + B = 200
		A = 120 B = 80
`read A`		
`A = A - 50`		
`write A`		A = 70
	`read A`	
	`read B`	
	`display A + B` (150 est affiché)	
`read B`		
`B = B + 50`		
`write B`		B = 130

T_1 est une transaction cohérente, vis à vis de la contrainte d'intégrité, puisqu'elle retranche à A ce qu'elle ajoute à B. T_2 devrait afficher la valeur 200. Mais l'ordre des opérations est tel que T_2 affiche 150.

Lecture impropre (données non confirmées)

T_1	T_2	BD
		A = 50
	`A = 70`	
	`write A`	A= 70
`read A` `(70 est lu)`		
	`rollback` (La valeur initiale de A est restaurée)	A = 50

T_1 a lu une valeur de A incorrecte car tout doit se passer comme si T_2 n'avait jamais changé A.

Lecture non reproductible

T_1	T_2	BD
		A = 10
	`read A` (10 est lu)	
`A = 20`		
`write A`		A = 20
	`read A` (20 est lu)	

T_2 qui ne modifie pas A doit obtenir à chaque lecture la même valeur pour cette donnée. Mais l'ordre des opérations est tel que T_2 lit 10 puis 20.

Exécution sérialisable

Le principe sur lequel repose le contrôle de concurrence est celui de la **sérialisabilité** de l'exécution d'un ensemble de transactions.

En effet, lorsque les transactions sont exécutées les unes après les autres, il n'y a pas de problèmes de concurrence. Malheureusement, cette solution est inapplicable car très coûteuse en temps calcul.

La solution adoptée consiste à exécuter un ensemble de transactions concurrentes de façon à ce que le résultat soit équivalent à une exécution en série de ces transactions : une telle exécution est dite **sérialisable**.

La sérialisabilité peut se définir de la façon suivante :

- L'exécution d'un ensemble de transactions est dite **en série** si, pour tout couple de transactions, tous les événements de l'une précèdent tous les événements de l'autre.
- Deux exécutions d'un même ensemble de transactions sont **équivalentes** si et seulement si :
  - elles sont constituées des mêmes événements,
  - elles produisent le même état final de la BD.
- Une exécution concurrente d'un ensemble de transactions est dite **sérialisable** si et seulement si il existe une exécution en série équivalente.

Condition de sérialisabilité

La recherche de l'exécution en série équivalente à une exécution concurrente d'un ensemble de transactions, peut se faire en exploitant les règles suivantes de **commutation** des opérations de lecture et d'écriture.

Soient Op_1 une opération d'une transaction T_1 et Op_2 une opération d'une transaction T_2.

- Si Op_1 et Op_2 n'opèrent pas sur la même donnée, alors Op_1 et Op_2 sont permutables.
- Si Op_1 et Op_2 opèrent sur la même donnée D :
  - Op_1 = `read D` et Op_2 = `read D` : Op_1 et Op_2 sont permutables car dans les deux cas T_1 et T_2 lisent la même valeur de D.
  - Op_1 = `read D` et Op_2 = `write D` : Op_1 et Op_2 ne sont pas permutables car dans un cas T_1 lit la valeur de `D` avant sa modification par T_2 et dans l'autre, la valeur de `D` modifiée par T_2.
  - Op_1 = `write D` et Op_2 = `read D` : Op_1 et Op_2 ne sont pas permutables pour des raisons similaires au cas précédent.
  - Op_1 = `write D` et Op_2 = `write D` : Op_1 et Op_2 ne sont pas permutables, car la prochaine opération de lecture ne lira pas la même valeur de D (celle écrite par T_1 dans un cas et celle écrite par T_2 dans l'autre).

Deux opérations sont dites **conflictuelles** si elles appartiennent à deux transactions différentes et si elles ne sont pas permutables**.**

D'après les règles ci-dessus, deux opérations appartenant à deux transactions différentes sont conflictuelles si et seulement si elles portent sur la même donnée et que l'une des deux au moins est une opération d'écriture.

Une exécution concurrente est sérialisable si elle peut être transformée en une exécution en série équivalente par une suite de permutations d'opérations non conflictuelles.

Exemple

T_1	T_2
`read A`	
`write A`	
	`read A`
`read B`	
	`write A`
`write B`	
	`read B`
	`write B`

(1)

T_1	T_2
`read A`	
`write A`	
`read B`	
`write B`	
	`read A`
	`write A`
	`read B`
	`write B`

(2)

L'exécution concurrente (1) est sérialisable car elle peut être transformée en (2) qui est une exécution en série, par les trois permutations d'opérations non conflictuelles suivantes :

- permutation de `read B` de T_1 avec `read A` de T_2,
- permutation de `write B` de T_1 avec `write A` de T_2,
- permutation de `write B` de T_1 avec `read A` de T_2,

Le verrouillage

Le gestionnaire de transactions d'un SGBD assure le respect des propriétés de cohérence et d'isolation.

La technique la plus simple pour éviter les problèmes dus à la concurrence est le **verrouillage** : chaque transaction verrouille les données qu'elle lit ou écrit pour interdire aux autres transactions d'y accéder.

Le verrouillage dégrade les performances d'un SGBD en imposant des temps d'attente. Il y a intérêt à limiter la taille des données à verrouiller, d'où le concept de **granularité de verrouillage**. On peut encore améliorer les performances en précisant la nature des opérations pour lesquelles le verrouillage est réalisé (lecture ou écriture par exemple) cela a conduit a la définition de **modes de verrouillage**.

Granularité

On peut verrouiller :

- une valeur d'attribut,
- un n-uplet et donc toutes ses valeurs,
- une page de fichier et donc tous ses n-uplets,
- une table et donc toutes ses lignes,
- la BD et donc toutes ses tables.

Modes de verrouillage

Les modes de verrouillage varient d'un SGBD à l'autre. Deux modes sont toujours définis :

- **partagé** (**S**) : demandé avant de lire une donnée,
- **exclusif** (□) : demandé avant de modifier une donnée.

Les règles qui régissent ces deux modes sont les suivantes :

1. Un verrou partagé ne peut être obtenu sur une donnée que si les verrous déjà placés sur cette donnée sont eux même partagés.
2. Un verrou exclusif ne peut être obtenu sur une donnée que si aucun verrou n'est déjà placé sur cette donnée.

Les règles qui régissent les modes de verrouillage peuvent être exprimées par une **matrice de compatibilité**. Pour les modes S et X cette matrice est la suivante :

MC	S	X
S	oui	non
X	non	non

où MC(i, j) indique si un verrou en mode i peut être obtenu sur une donnée quand un verrou en mode j y est déjà placé.

Opérations de verrouillage

Les deux opérations suivantes gèrent le verrouillage et s'ajoutent aux opérations élémentaires d'une transaction :

Opération	**Signification**
`lock m D`	demande d'un verrou en mode m sur la donnée D
`unlock D`	déverrouillage d'une donnée D

Verrouillage à deux phases

Une transaction est **bien formée** si :

- elle obtient un verrou sur une donnée avant de lire ou d'écrire cette donnée,
- elle libère tous ses verrous avant de se terminer.

Une transaction est à **deux phases** si elle est bien formée et si après avoir libéré un verrou elle n'en acquiert plus : on distingue une **phase d'acquisition** des verrous et une phase de **libération des verrous**.

Le protocole habituel consiste à lever les verrous aprés la confirmation de la transaction (`commit`).

Il est démontré que l'exécution d'un ensemble de transactions à deux phases est sérialisable, c.-à-d. est équivalent à une exécution en série. L'ordre de sérialisation est celui de l'acquisition du dernier verrou.

En conséquence il ne peut y avoir ni pertes de mise à jour, ni lectures impropres, ni lectures non reproductibles.

Elimination des problèmes de concurrence

Perte de mise a jour

T_1	T_2	BD
		A = 10
`lock X A`		
`read A`		
	`lock X A`	
`A = A + 10`	*attente*	
`write A`	*attente*	A = 20
`unlock A`	*attente*	
	`read A`	
	`A = A + 50`	
	`write A`	
	`unlock A`	A = 70

Le verrouillage de A par T_1 oblige T_2 à attendre que T_1 ait terminé sa mise à jour de A avant d'entamer la sienne. Il n'y a donc pas de perte de mise à jour.

Lecture impropre (données incohérentes)

T_1	T_2	BD
		contrainte A + B = 200
		A = 120 B = 80
`lock X A`		
`read A`		
`A = A - 50`		
`write A`		A = 70
	`lock S A`	
`lock X B`	*attente*	
`read B`	*attente*	
`B = B + 50`	*attente*	
`write B`	*attente*	B = 130
`unlock A`	*attente*	
	`read A`	
`unlock B`		
	`lock S B`	
	`read B`	
	`display A + B` (200 est affiché)	

Le verrouillage de A par T_1 empêche T_2 de lire A avant que T_1 n'ait terminé sa mise à jour de A et de B. T_2 affiche donc une valeur cohérente de A + B soit 200.

Lecture impropre (données non confirmées)

T_1	T_2	BD
		A = 50
	`lock X A`	
	`A = 70`	
	`write A`	A= 70
`lock S A`		
attente	`rollback` (La valeur initiale de A est restaurée. A est déverrouillée.)	A = 50
`read A` (50 est lu)		

Le verrouillage de A par T_2 empêche T_1 de lire A avant que T_2 n'ait été annulée. T_1 lit donc bien la valeur qu'avait A avant que T_2 ne commence.

Lecture non reproductible

T_1	T_2	BD
		A = 10
	`lock S A`	
	`read A` (10 est lu)	
`lock X A`		
attente	`read A` (10 est lu)	
attente	`unlock A`	
`A = 20`		
`write A`		A = 20

Le verrouillage de A par T_2 durant le temps de ses deux lectures empêche T_1 à de modifier A entre ces deux lectures. T_2 lit donc bien deux fois la même valeur de A soit 10.

Reclassement d'un verrou

Soit une transaction qui lit puis modifie une donnée D. Si elle fonctionne en verrouillage à deux phases, elle devra demander un verrou en mode exclusif dès la lecture de D. Il n'y aura donc pas de concurrence possible entre la lecture de D et sa mise à jour.

Afin d'augmenter le degré de concurrence, on introduit la possibilité de reclasser un verrou à l'aide des deux opérations suivantes :

Opération	**Signification**
`upgrade D`	surclassement en X du verrou sur D
`downgrade D`	déclassement en S du verrou sur D

Le reclassement d'un verrou est soumis aux mêmes règles que l'obtention d'un verrou en mode S ou X.

Par exemple :

T
`lock S A`
`read A`
`A = A + 10`
`upgrade A`
`write A`
`downgrade A`

Interblocage

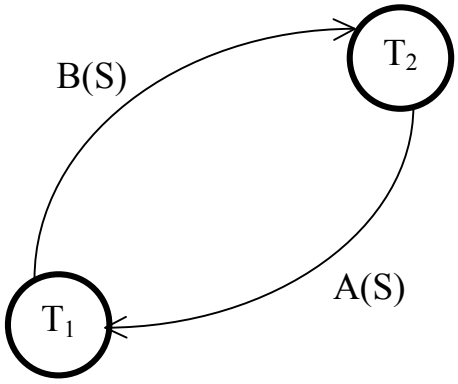
L'inconvénient de la technique de verrouillage est le risque d'**interblocage** (ou **verrou mortel**) entre plusieurs transactions.

Voici un exemple classique d'interblocage :

T_1	T_2
`lock X A`	
	`lock X B`
`lock S B`	
attente	`lock S A`
attente	*attente*

Une situation d'interblocage se traduit par un cycle dans le **graphe d'attente** des transactions. Chaque nœud de ce graphe est une transaction et il existe un arc orienté d'un nœud T_i vers nœud T_j étiqueté par D(m) si T_i attend que T_j libère son verrou sur D pour y placer verrou en mode m.

Le graphe d'attente de l'exemple ci-dessus est le suivant :



Il contient un cycle qui confirme l'interblocage.

Résolution de l'interblocage

Deux techniques peuvent être utilisées pour résoudre l'interblocage :

- la **prévention** : on adopte un protocole qui évite les verrous mortels, par exemple en accordant à une transaction tous les verrous dont elle a besoin avant son démarrage, ou bien en fixant un ordre sur les données et en imposant aux transactions de respecter cet ordre dans leur demandes de verrous.
- la **détection** : on laisse les situations d'interblocage se produire et l'on inspecte à intervalles réguliers le graphe d'attente pour détecter si un interblocage s'est produit. Dans ce cas on défait l'une des transactions bloquées et on la relance un peu plus tard.

Les interblocages étant relativement rares, la technique de détection est souvent la plus économique.

Reprise après panne

Les pannes d'un SGBD peuvent être classées en trois catégories :

- **Abandon de transaction**. Il peut survenir à la suite d'une erreur de programme, d'un abandon explicite de l'utilisateur ou être provoqué par le gestionnaire de transactions.
- **Panne d'ordinateur**, par exemple à la suite d'une coupure de courant. Les données enregistrées en mémoire centrale sont perdues mais pas celles enregistrées sur disque.
- **Panne de disque**. Une partie ou toutes les données enregistrées sur ce disque sont perdues.

Le gestionnaire de reprise d'un SGBD assure le respect des propriétés d'atomicité et de durabilité des transactions.

Nous étudions dans la suite la reprise après abandon de transaction ou arrêt de l'ordinateur. Cette reprise "à chaud", peut être réalisée automatiquement et rapidement, en s'appuyant sur la tenue d'un journal qui garde en mémoire tous les événements d'une transaction.

La reprise après panne de disque est plus longue à mettre en œuvre. Elle nécessite de réaliser des copies régulières de la BD et un archivage des mises à jour entre deux copies.

Outils pour la reprise

Journal (log)

Le **journal** est un fichier séquentiel qui contient la suite des événements concernant la vie des transactions et les modifications de la BD, depuis un instant donné.

Nous considérerons les événements suivants :

Evénement	**Signification**
`(T start)`	la transaction T a commencé
`(T D a n)`	la donnée D a été remplacée : elle est passée de la valeur a à la valeur n
`(T commit)`	la transaction T a été confirmée
`(check point)`	point de reprise

Un journal est enregistré sur une **mémoire stable**, c.-à-d. une mémoire qui théoriquement ne peut pas être détruite. Par exemple, sur des disques RAID ("Redundant Array of Inexpensive Disks") en associant à chaque disque logique deux disques physiques dont l'un est le mirroir de l'autre.

Exemple de journal

T_1	T_2	Journal
start		(T1 start)
lock X A		
read A (A = 15)		
A = A - 1		
write A		(T1 A 15 14)
lock S B		
unlock A		
	start	(T2 start)
	lock X A	
	read A	
read B		
	A = A * 2	
	write A	(T2 A 14 28)
	commit	(T2 commit)
B = B / A		

Point de reprise

Un **point de reprise** est une marque dans le journal qui indique qu'au moment où elle a été faite :

- aucune transaction n'était en cours,
- toutes les données écrites par des transactions antérieures au point de reprise avaient été transférées sur disque.

Pour obtenir un point de reprise, il faut :

1. interdire le début de nouvelles transactions,
2. laisser se terminer les transactions en cours,
3. écrire sur disque (dans le tampon) toutes les données encore présentes en mémoire centrale,
4. écrire `(checkpoint)` dans le journal,
5. écrire le dernier bloc du journal sur le disque.

Etude de cas

	T_1	T_2
(1)	`start`	
(2)	`lock X A`	
(3)	`read A`	
(4)	`A = A - 1`	
(5)	`write A`	
(6)	`lock S B`	
(7)	`unlock A`	
(8)		`start`
(9)		`lock X A`
(10)		`read A`
(11)	`read B`	
(12)		`A = A * 2`
(13)		`write A`
(14)		`commit`
(15)	`B = B / A`	
(16)	interruption !	

Analyse

T_1 a été interrompue, elle n'a donc pas réalisé toutes ses opérations. Pour respecter la propriété d'atomicité T_1 doit être défaite, ce qui implique les opérations suivantes :

- Le verrou que T_1 a placé sur B doit être levé pour que B soit accessible.
- A doit retrouver la valeur qu'elle avait avant sa mise à jour par T_1 (5).
- La valeur de A lue par T_2 en (10) n'est alors plus la bonne, et par conséquent les mises à jour faites par T_2 pourront être incorrectes : il faut donc défaire T_2.
- D'autres transactions ont pu lire A : elles devront aussi être défaites (cascades de transactions à défaire).

Reprise à partir du journal

Les transactions sont à deux phases.

Afin d'éviter d'avoir à défaire des transactions en cascade, les verrous posés par une transaction ne sont libérés qu'après sa confirmation (`commit`).

Rappelons qu'une écriture dans la BD se fait au travers d'un tampon. Une valeur écrite par l'opération `write D` pourra donc résider dans le tampon et n'être écrite sur le disque que bien plus tard.

Il faut distinguer deux modes de mise à jour :

- **Mise à jour immédiate**. Les écritures dans la BD sont effectuées immédiatement. Une transaction non confirmée a donc pu écrire dans la BD.
- **Mise à jour différée**. Les nouvelles valeurs des données sont stockées dans le journal. Leurs écritures dans la BD ne sont effectuées qu'après le `commit`. Une transaction non confirmée n'a donc rien écrit dans la BD.

Classification des transactions après une panne

Quand une panne survient, on peut classer les transactions en trois catégories :

1. **Les transactions terminées avant le dernier point de reprises**. Leurs mises à jour sont enregistrées sur disque, par définition d'un point de reprise.
2. **Les transactions commencées après le dernier point de reprise mais terminées avant la panne**. Les nouvelles valeurs des données peuvent résider dans le tampon sans avoir été écrites sur disque. Il faudra donc refaire les mises à jour de ces transactions en réécrivant les nouvelles valeurs des données qui devront avoir été mémorisées dans le journal.
3. **Les transactions en cours au moment de la panne**. La propriété d'atomicité implique que leurs effets soient annulés. Dans le mode de mise à jour immédiate, il faudra défaire les mises à jour de ces transactions en réécrivant les anciennes valeurs des données qui devront avoir été mémorisées dans le journal.

Mise à jour immédiate

Déroulement d'une transaction et journalisation

- Pour chaque événement `start` d'une transaction T : écrire `(start T)` dans le journal.
- Pour chaque opération `write D` d'une transaction T :
  - Ecrire `(T D ancienne-valeur nouvelle-valeur)` dans le journal, puis écrire le dernier bloc du journal dans la mémoire stable.
  - Ecrire D dans la BD.
- Pour chaque événement `commit` d'une transaction T :
  - Ecrire `(commit T)` dans le journal, puis écrire le dernier bloc du journal dans la mémoire stable.

Reprise

- On déroule le journal à l'envers jusqu'au dernier point de reprise, en notant les transactions confirmées.
- On déroule le journal à l'envers jusqu'au dernier point de reprise.
  Pour chaque opération `(T D a n)` d'une transaction non confirmée T :

  Ecrire l'ancienne valeur a de D dans la BD.
- On déroule le journal à l'endroit, depuis le dernier point de reprise.
  Pour chaque opération `(T D a n)` d'une transaction confirmée T :

  Ecrire la nouvelle valeur n de D dans la BD.

Mise à jour différée

Déroulement d'une transaction et journalisation

- Pour chaque événement `start` d'une transaction T écrire `(start T)` dans le journal.
- Pour chaque opération `write D` d'une transaction T :
  écrire `(T D ancienne-valeur nouvelle-valeur)` dans le journal.
- Pour chaque événement `commit` d'une transaction T :
  - Ecrire `(commit T)` dans le journal.
  - Ecrire le dernier bloc du journal dans la mémoire stable. : la transaction est confirmée.
  - Ecrire dans la BD les données modifiées par la transaction T.

Reprise

- On déroule le journal à l'envers jusqu'au dernier point de reprise, en notant les transactions confirmées.
- On déroule le journal à l'endroit à partir du point de reprise.
  Pour chaque opération `(T D n)` d'une transaction T confirmée :
  Ecrire la valeur n de D dans la BD.

Bibliographie

De très nombreux ouvrages sur les bases de données en général et relationnelles en particulier, sont disponibles. Voici ceux qui ont servi à la préparation de ce cours.

Besancenot J. et al., Les systèmes transactionnels : concepts, normes et produits, Hermès, 1997.

Date C. J., Introduction aux bases de données, International Thomson Publishing, France, 1998.

Delmal P., SQL 2 De la théorie à l'application, De Boeck Université, 1995.

Gardarin G., Bases de Données objet □ relationnel, Eyrolles, Paris, 1999.

Gray J., Reuter A., Transactions Processing Concepts and Techniques, Morgan □aufman, 1993.

Silberschatz A., □orth H. F., Sudarshan S., Database System Concepts, Mc-Gra□ Hill, 1997.

Ullman J. D., Database and □no□ledge-Base (t□o volumes), Computer Science Press, (1988, 1989).