IN306 - Bases de données

Christophe Garion

ISAE/DMIA - SUPAERO/IN
10 avenue Édouard Belin
31055 Toulouse Cedex 4

Pourquoi des bases de données ?

Qui connaît l'autre et se connaît lui-même, peut livrer cent batailles sans jamais être en péril. Qui ne connaît pas l'autre mais se connaît lui-même, pour chaque victoire, connaîtra une défaite. Qui ne connaît ni l'autre ni lui-même, perdra inéluctablement toutes les batailles.

Sun Tzu, L'Art de la Guerre

Pourquoi des bases de données ?

> *Qui connaît l'autre et se connaît lui-même, peut livrer cent batailles sans jamais être en péril. Qui ne connaît pas l'autre mais se connaît lui-même, pour chaque victoire, connaîtra une défaite. Qui ne connaît ni l'autre ni lui-même, perdra inéluctablement toutes les batailles.*
>
> Sun Tzu, L'Art de la Guerre

Le monde (de l'entreprise) croule sous les informations :

- hétérogènes, mais concernant les mêmes choses
- réparties sur plusieurs endroits, sur plusieurs personnes
- à recouper

Pourquoi des bases de données ?

> *Qui connaît l'autre et se connaît lui-même, peut livrer cent batailles sans jamais être en péril. Qui ne connaît pas l'autre mais se connaît lui-même, pour chaque victoire, connaîtra une défaite. Qui ne connaît ni l'autre ni lui-même, perdra inéluctablement toutes les batailles.*
>
> Sun Tzu, L'Art de la Guerre

Le monde (de l'entreprise) croule sous les informations :

- hétérogènes, mais concernant les mêmes choses
- réparties sur plusieurs endroits, sur plusieurs personnes
- à recouper

But des bases de données

Organiser tout cela (si possible de façon efficace et efficiente) !

Ordres de grandeur

Puisque l'on va parler de stockage d'information (pas seulement...), quelques ordres de grandeur :

unité	**taille**
1 B	mot de 8 bits = 256 valeurs possibles
1 GB	1024^3B = 1073741824 B
1 TB	1024^4B = 1099511627776 B
1 PB	1024^5B = 1125899906842624 B

Ordres de grandeur

Puisque l'on va parler de stockage d'information (pas seulement...), quelques ordres de grandeur :

unité	taille
1 B	mot de 8 bits = 256 valeurs possibles
1 GB	1024^3B = 1073741824 B
1 TB	1024^4B = 1099511627776 B
1 PB	1024^5B = 1125899906842624 B

Si un grain de riz ≡ 1 octet...

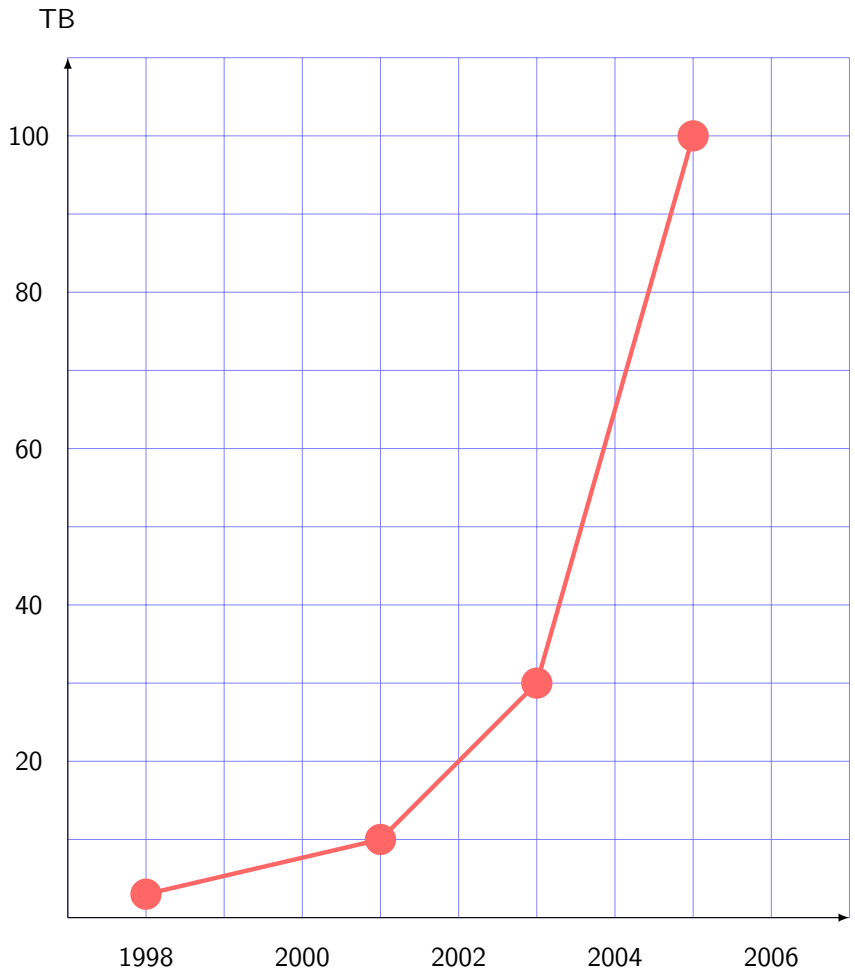
- 1 GB ≡ un sac de 25 kg de riz
- 1 PB permettent de recouvrir le centre de Londres sous 1m de riz

Quelques chiffres actuels...

qui	taille	remarques
Library of Congress	100 TB	
Ancestry.com	600 TB	
Facebook	1.5 PB	simplement les photos
Internet Archives	3 PB	100 TB/mois
NERSC	3 PB	
RapidShare	5 PB	
World Data Center for Climate	6 PB	220 TB de données accessibles sur le web
LHC	15 PB/an	
Google Inc.	?	20 PB de données chaque jour...
tout le travail écrit de l'humanité	$\approx$ 50 PB	
mémoire d'un être humain	$\approx 1,5$ TB...	

Évolution de la taille des BD

D'après WinterCorp (`http://www.wintercorp.com/`)



Déroulement du cours

séances	contenu
1-2	modèle entité-association
3	stockage des informations
3-4	modèles de bases de données, modèle relationnel
5	algèbre relationnelle
6-7	SQL
8-9	**BE**
10-11	aspects avancés de SQL, intégration de SQL dans un langage de programmation
12-13	dépendances fonctionnelles
14-15	gestion des transactions
15	base de données objet
16	**examen**

Première partie

Le modèle de Chen

1. Généralités et notions de base
2. Représentation des concepts
3. Schéma conceptuel d'une BD : le diagramme E/A

Première partie

Le modèle de Chen

L'information et sa représentation

Distinction entre :

- le monde **réel**
- le monde **perçu**
- le **représenté**

L'information et sa représentation

Distinction entre :

- le monde **réel**
- le monde **perçu**
- le **représenté**

Nécessité d'avoir :

- un **langage de représentation**, textuel ou graphique
- une **sémantique** qui permet de lever les ambiguïtés et les non-sens, représentée par un **modèle**

L'information et sa représentation

Distinction entre :

- le monde **réel**
- le monde **perçu**
- le **représenté**

Nécessité d'avoir :

- un **langage de représentation**, textuel ou graphique
- une **sémantique** qui permet de lever les ambiguïtés et les non-sens, représentée par un **modèle**

Ici, utilisation du modèle de Chen, ou modèle **entité-association**.

Chen, P. (1976).
The Entity-Relationship model - toward a unified view of data.
ACM Transactions on Database Systems, 1(1) :9–36.

Notions de base

Représentation de la connaissance **structurelle** et **descriptive**

entité	existence effective et autonome dans le monde réel
attribut	propriété liée à une entité
association	associe plusieurs entités entre elles

Notions de base

Représentation de la connaissance **structurelle** et **descriptive**

entité	existence effective et autonome dans le monde réel
attribut	propriété liée à une entité
association	associe plusieurs entités entre elles

Pour les attributs : **domaine** de valeurs [+ **contraintes**]

Notions de base

Représentation de la connaissance **structurelle** et **descriptive**

entité	existence effective et autonome dans le monde réel
attribut	propriété liée à une entité
association	associe plusieurs entités entre elles

Pour les attributs : **domaine** de valeurs [+ **contraintes**]

Exemple : entité Personne

Attention :

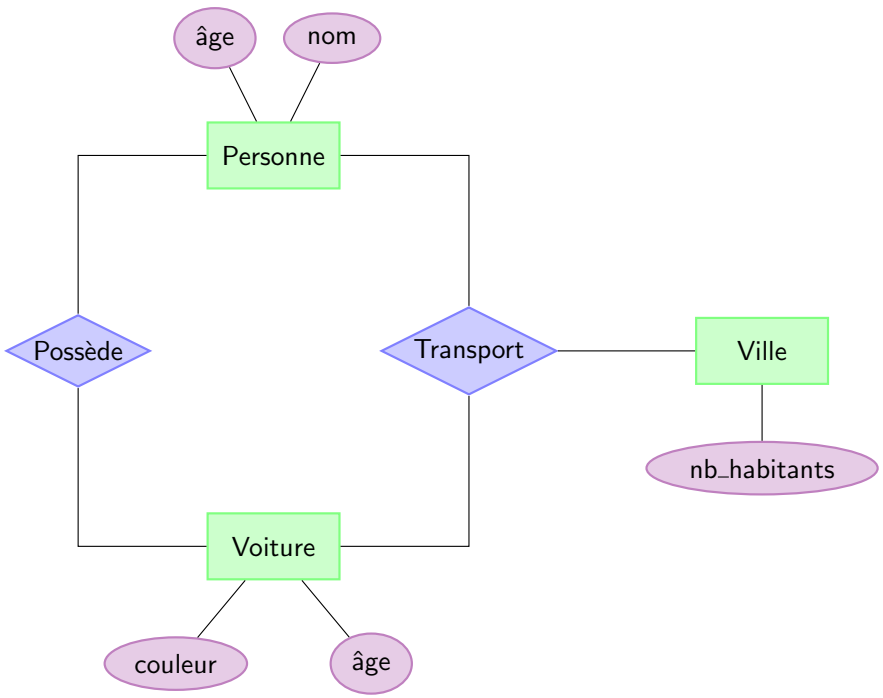
- attribut lié à une association
- entité/attribut

Structures des types

type d'entité	objet/classe
type d'attribut	$\mathcal{E} \mapsto \mathcal{D}$
type d'association	

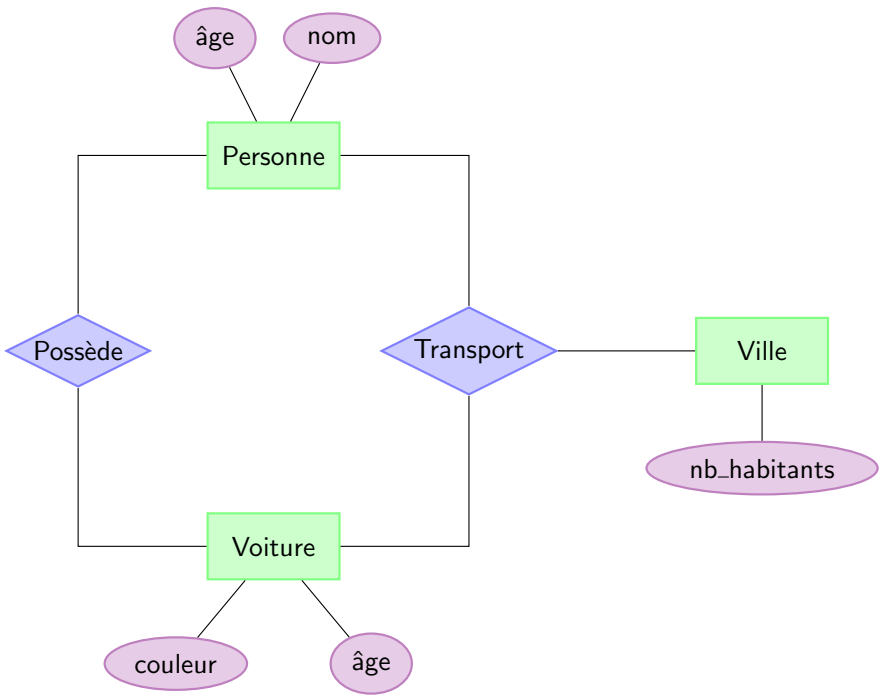
Structures des types

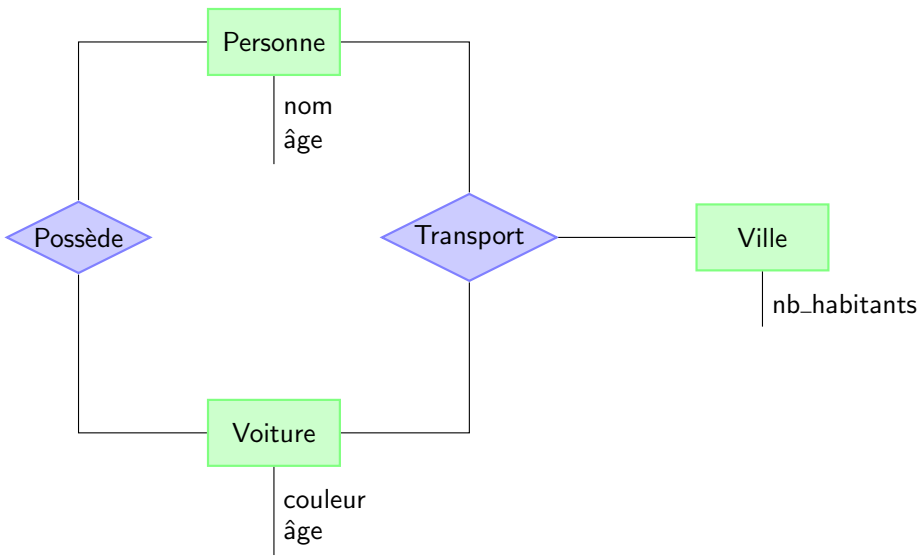
type d'entité objet/classe
type d'attribut $\mathcal{E} \mapsto \mathcal{D}$
type d'association



Structures des types

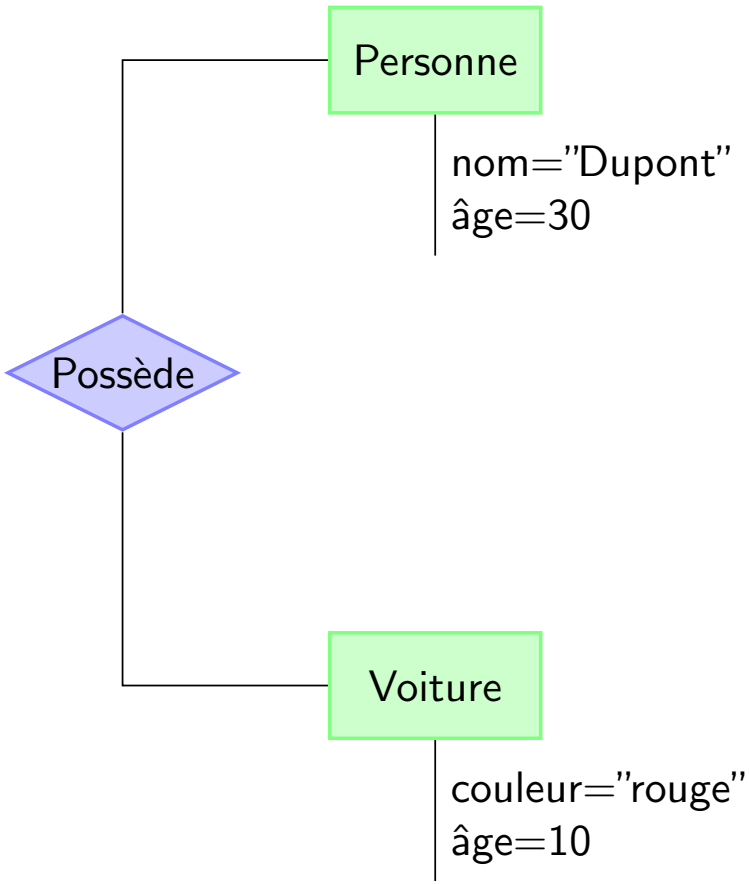
type d'entité objet/classe
type d'attribut $\mathcal{E} \mapsto \mathcal{D}$
type d'association





Occurences

Occurences d'entité, d'attribut, d'association



Première partie

Le modèle de Chen

Clé et clé primaire

Définition (clé et clé primaire)

Une clé est un attribut ou un ensemble d'attributs d'une classe d'entités ou d'association dont la connaissance d'une valeur identifie sans ambiguïté une occurrence unique de cette classe. Une clé primaire est une des clés possibles choisie pour servir de moyen d'identification d'une occurrence.

Clé et clé primaire

Définition (clé et clé primaire)

Une clé est un attribut ou un ensemble d'attributs d'une classe d'entités ou d'association dont la connaissance d'une valeur identifie sans ambiguïté une occurrence unique de cette classe. Une clé primaire est une des clés possibles choisie pour servir de moyen d'identification d'une occurrence.

- clé primaire : on peut la créer *ad hoc*
- « clé » d'une association $\equiv$ valeurs des clés primaires des entités
- les attributs clés $\equiv$ injections de la classe dans leurs domaines de valeurs.

Cardinalité de rôle

Définition (rôle)

Le rôle d'une classe d'entités E dans une classe d'association A est l'interprétation de la présence d'une entité de classe E dans une association de classe A.

Cardinalité de rôle

Définition (rôle)

Le rôle d'une classe d'entités E dans une classe d'association A est l'interprétation de la présence d'une entité de classe E dans une association de classe A.

Définition (cardinalité)

La cardinalité est une mesure de l'étendue des interventions possibles d'une entité quelconque d'une classe E dans des associations d'une classe A. La cardinalité est un intervalle $[min, max]$ (noté également $min..max$) sur $\mathbb{N}$, où min et max représentent respectivement le nombre minimal et le nombre maximal d'interventions possibles.

Cardinalité de rôle

Définition (rôle)

Le rôle d'une classe d'entités E dans une classe d'association A est l'interprétation de la présence d'une entité de classe E dans une association de classe A.

Définition (cardinalité)

La cardinalité est une mesure de l'étendue des interventions possibles d'une entité quelconque d'une classe E dans des associations d'une classe A. La cardinalité est un intervalle $[min, max]$ (noté également $min..max$) sur $\mathbb{N}$, où min et max représentent respectivement le nombre minimal et le nombre maximal d'interventions possibles.

- **fonctionnel** : ($min = 0$ ou $min = 1$) et $max = 1$.
- **hiérarchique** : ($min = 0$ ou $min = 1$) et $max = n$, $n > 1$.

Classes d'associations binaires

Définition (caractérisation d'une classe d'association binaire)

Soit A une classe d'association entre deux classes d'entités E et F. Cette classe est dite :

- **bijective** *si les rôles joués par E et F sont fonctionnels de type* $[1,1].x$
  On note alors $E \overset{1}{\longleftrightarrow}^{1} F$.
- **pseudo-bijective** *si les cardinalités des rôles joués par E et F sont de type* $[0,1]$.
  On note alors $E \overset{0}{\longleftrightarrow}^{1} F$ *ou* $E \overset{1}{\longleftrightarrow}^{0} F$.
- **hiérarchique de** E **vers** F *si le rôle joué par E est hiérarchique et le rôle joué par F est fonctionnel.*
  On note alors $E \overset{1}{\Longleftrightarrow}^{n} F$.
- **fonctionnelle de** E **vers** F *si le rôle joué par F est fonctionnel et celui de E hiérarchique.*
  On note alors $E \overset{n}{\Longleftrightarrow}^{1} F$.
- **maillée** *si les rôles joués par E et F sont hiérarchiques.*
  On note alors $E \overset{n}{\Longleftrightarrow}^{m} F$.

Première partie

Le modèle de Chen

Exemples de diagramme E/A

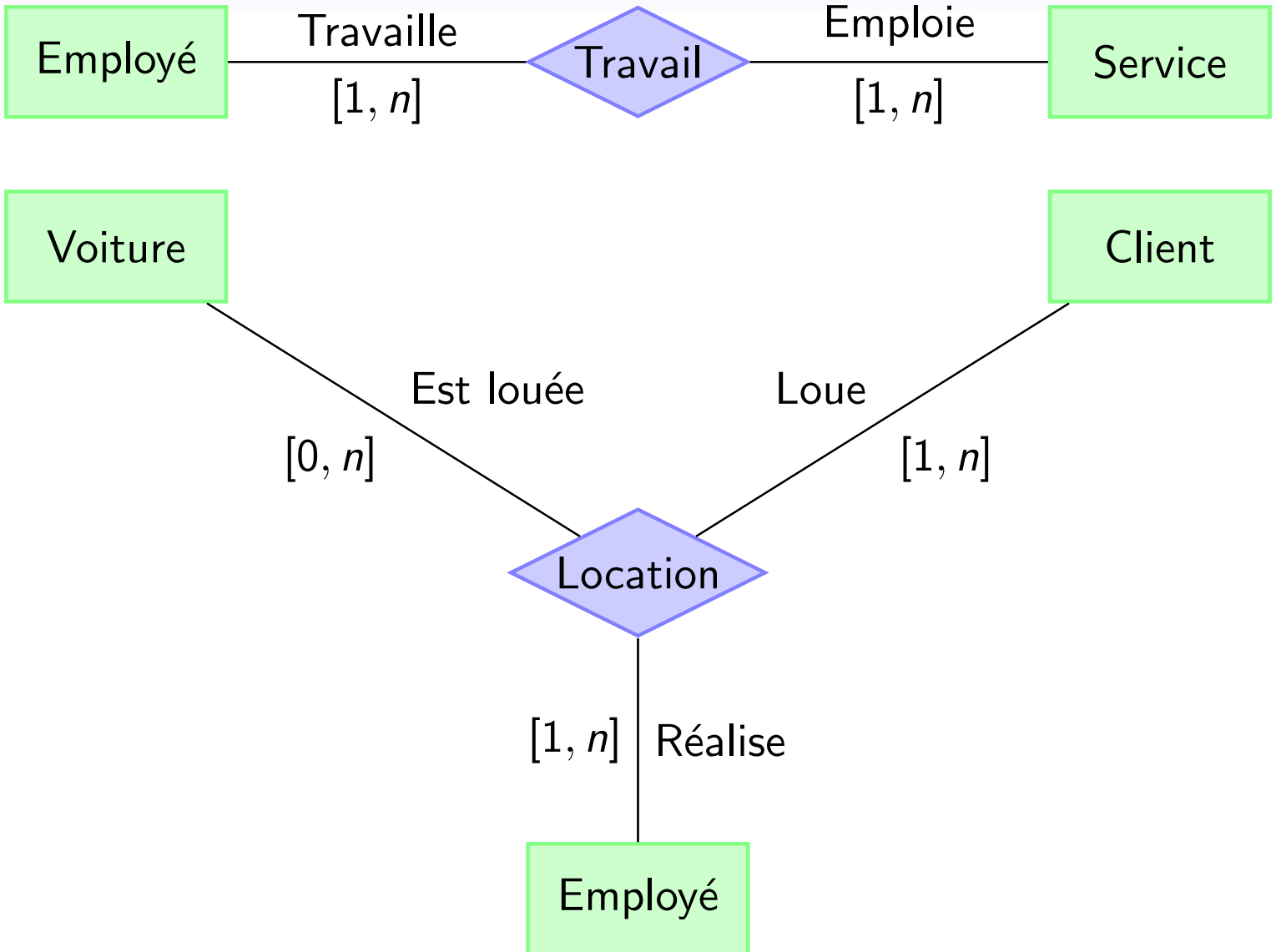


Diagramme E/A avec clé

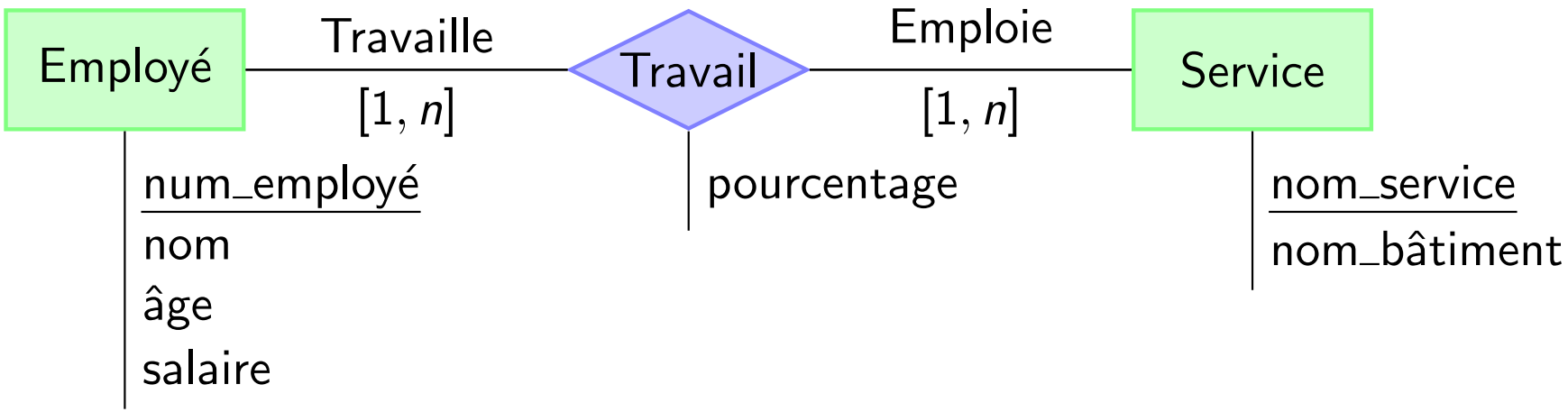


Schéma conceptuel

Définition (schéma conceptuel)

Un schéma conceptuel est composé des éléments suivants :

- *un diagramme entité-association ;*
- *un dictionnaire des attributs ;*
- *les domaines de valeurs des attributs.*

Problème de la location de voiture. . .

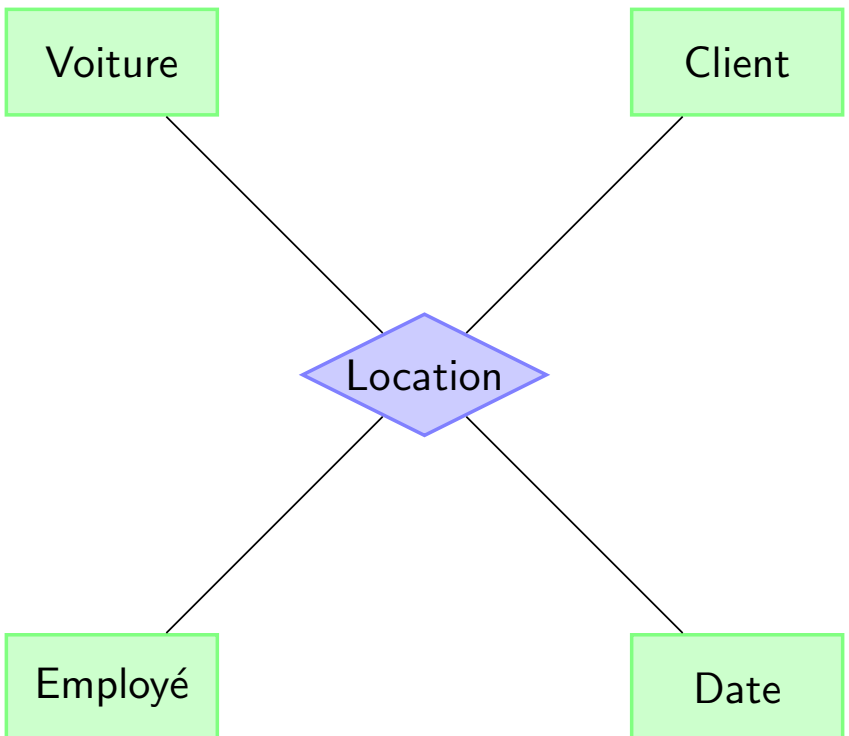
Location : loueur, voiture, employé. . .

. . . il faut la date !

Problème de la location de voiture...

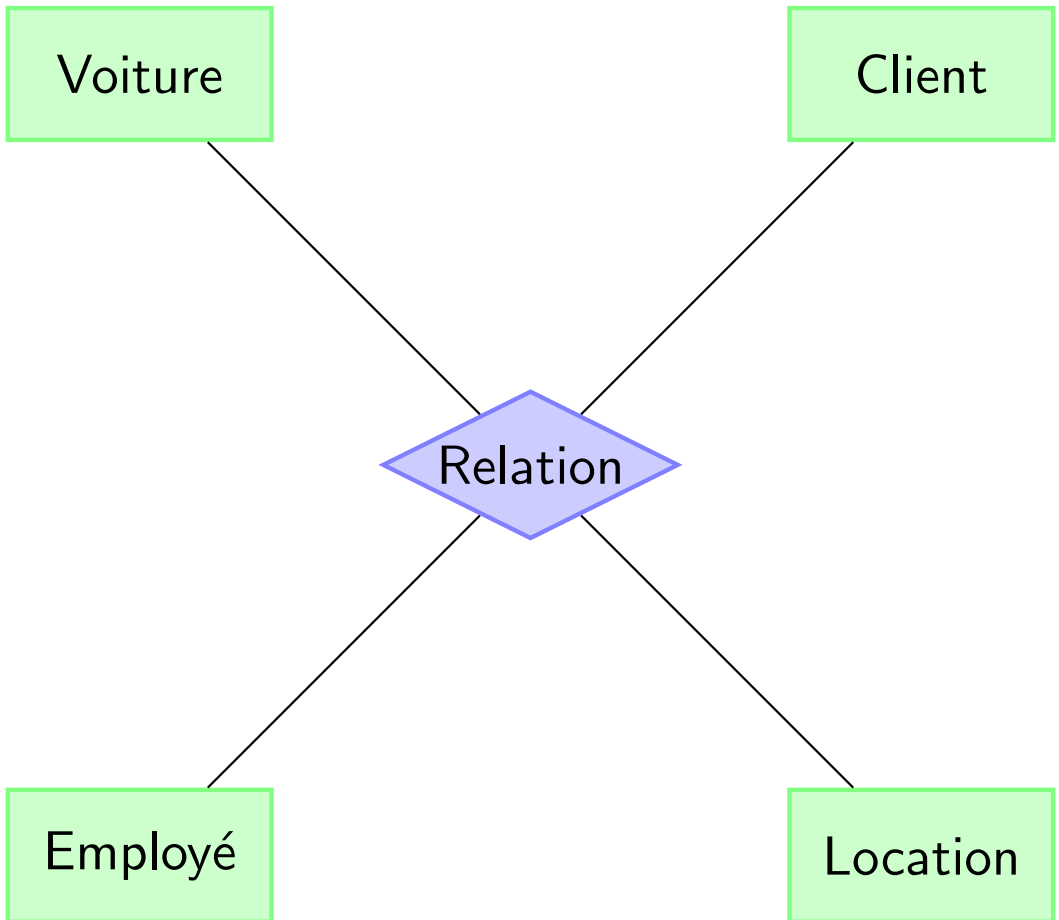
Location : loueur, voiture, employé...
...il faut la date !

Solution 1 :



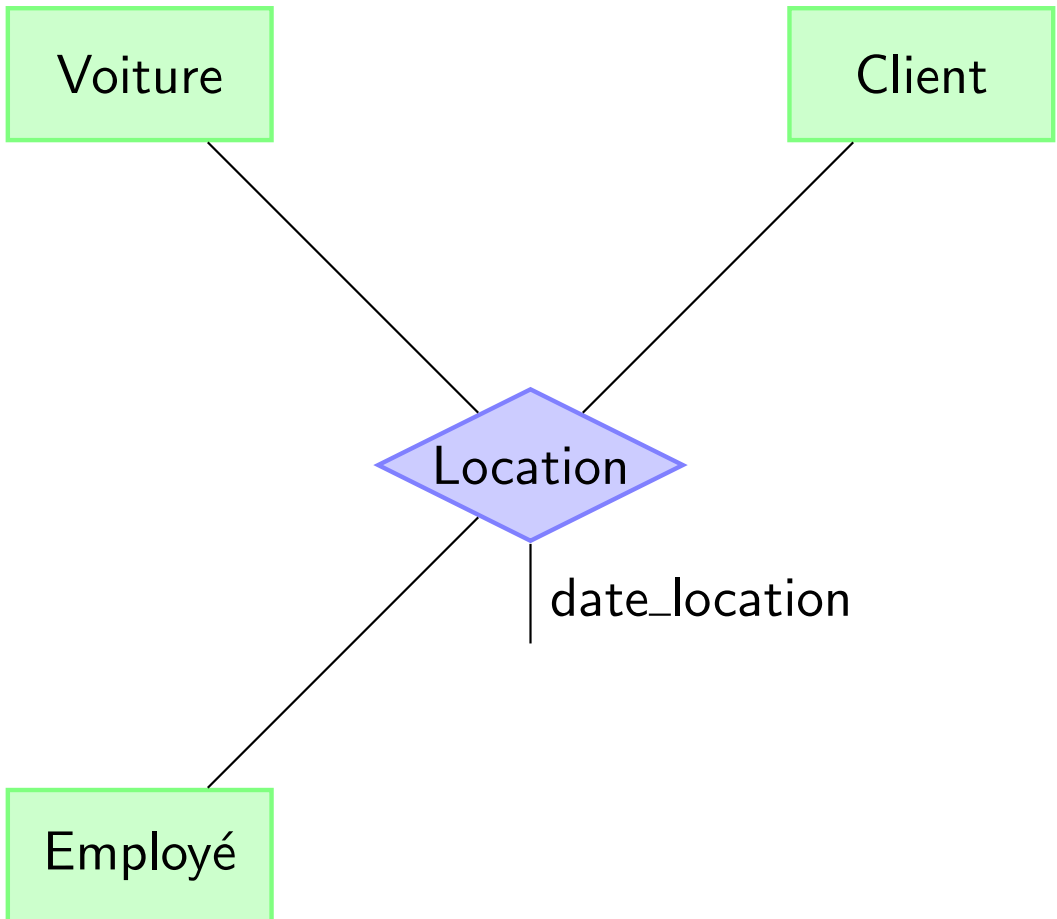
Problème de la location de voiture...

Solution 2 :

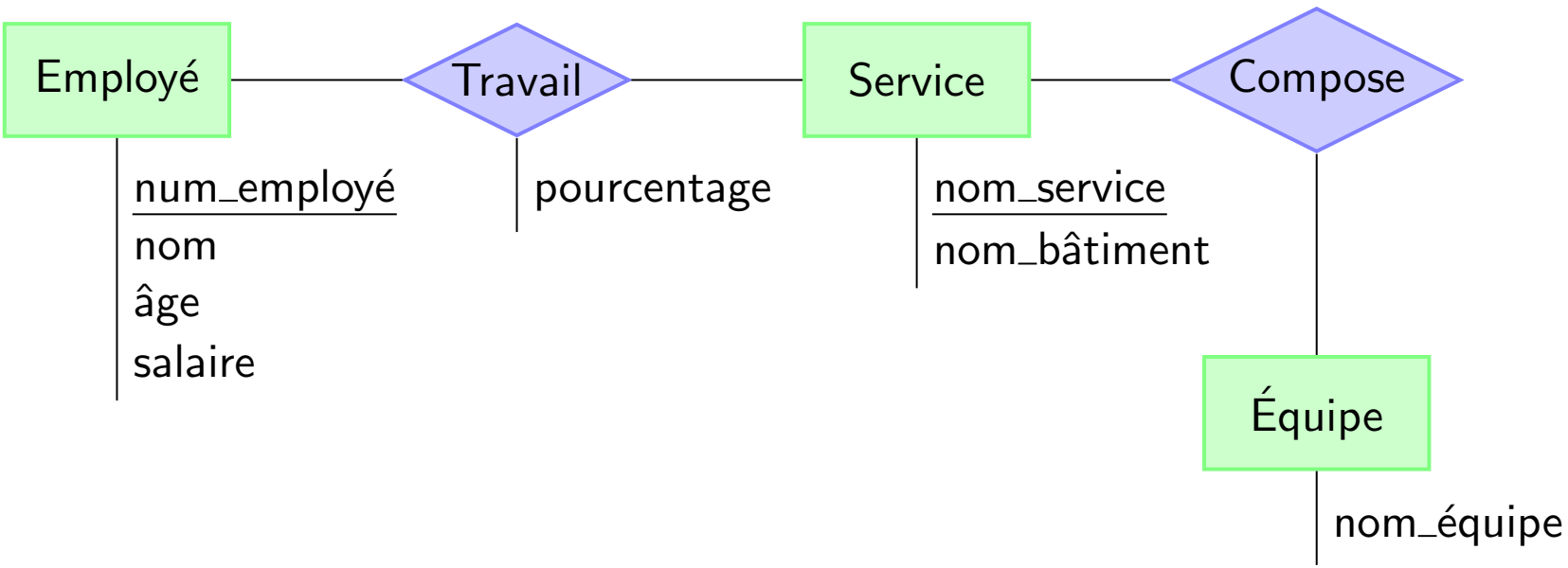


Problème de la location de voiture...

Solution 3 (celle que l'on retiendra pour des domaines de valeurs) :



Entités faibles : problématique

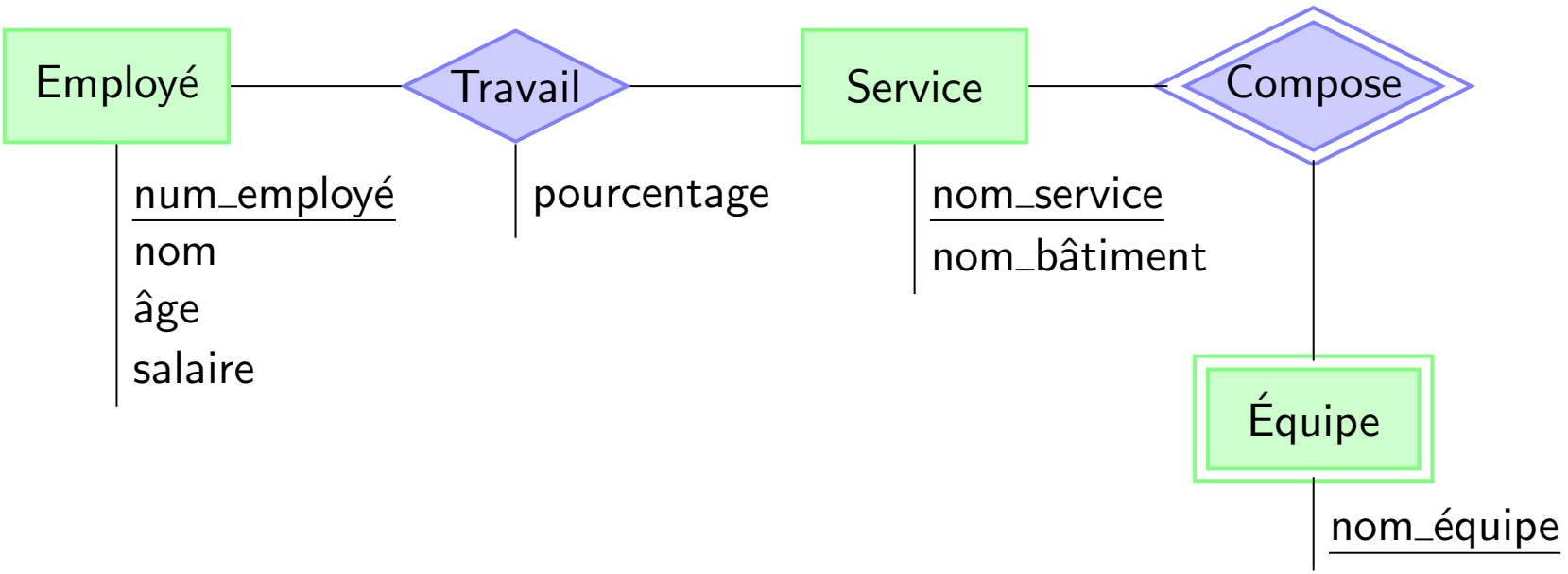


Entité faible : définition

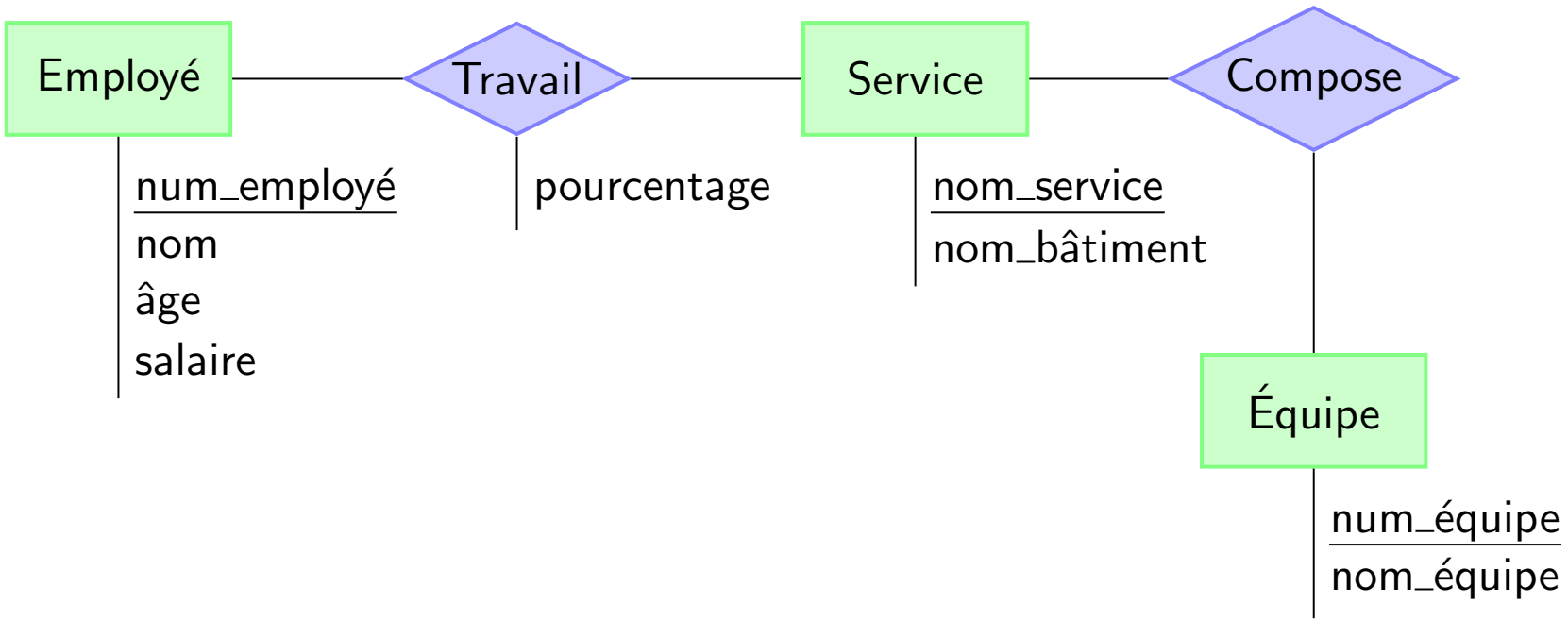
Définition

Une entité E est dite **faible** *si sa clé primaire est composé d'attributs propres à l'entité mais également de d'attributs clés d'autres entités lié à E par des associations fonctionnelles de E vers ces entités. On dit que ces associations sont des associations* **supports** *de E.*

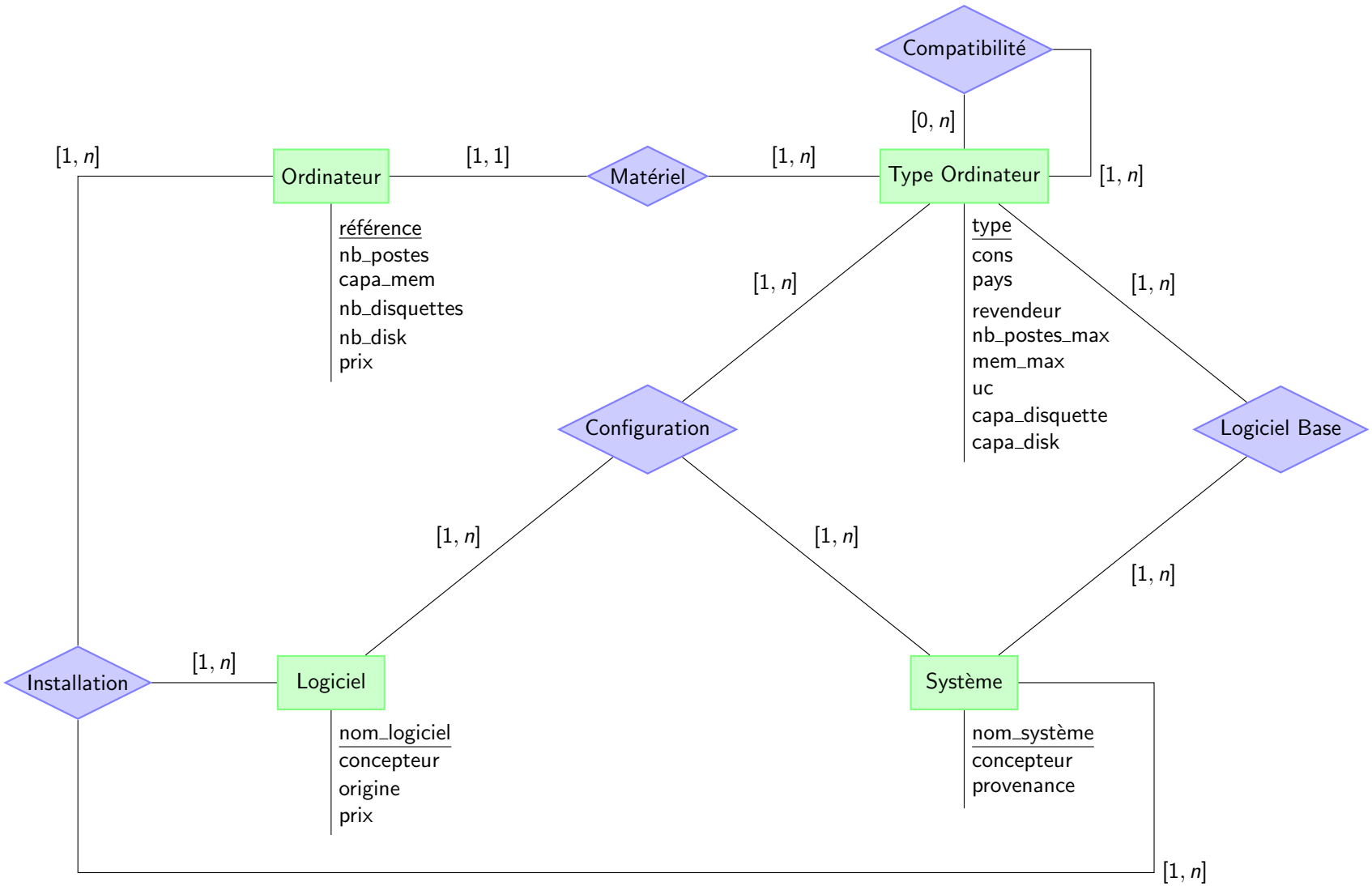
Entité faible : représentation



Entité faible : solution artificielle



Exemple de diagramme E/A



Deuxième partie

Stockage de l'information

4. Généralités sur le stockage de données
5. Impact du stockage secondaire
6. Organisation des données

Deuxième partie

Stockage de l'information

Remarques et bibliographie

Sujet compliqué :

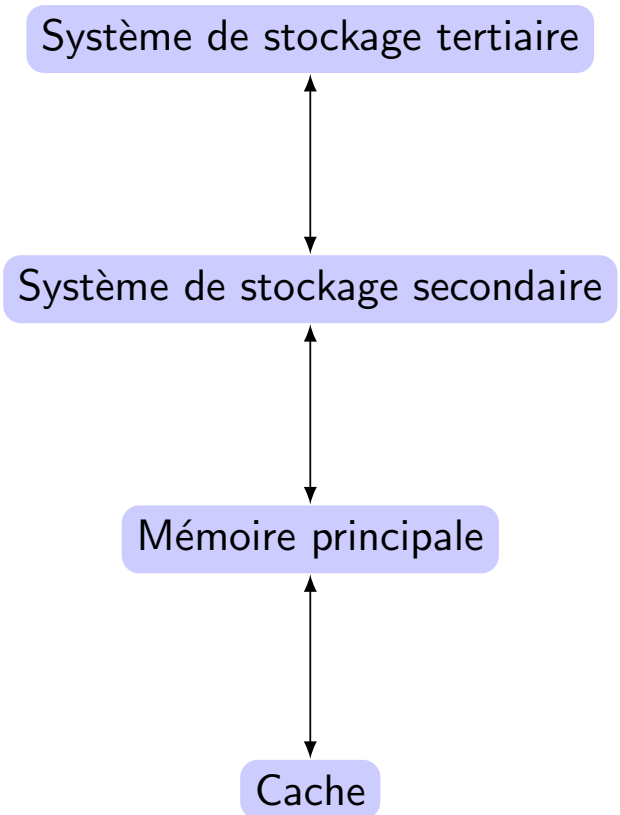
- transparent pour l'utilisateur **et** (souvent pour) l'administrateur d'une base
- ici : survol des notions

Garcia-Molina, H., Ullman, J., and Widom, J. (2002).
Database Systems : the Complete Book.
Prentice Hall.

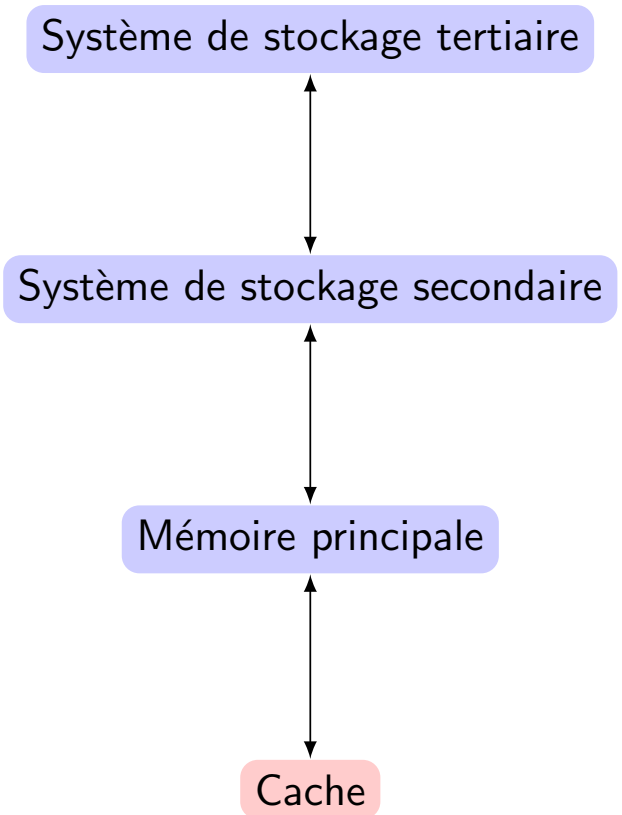
Tanenbaum, A. (2003).
Systèmes d'exploitation.
Dunod, 2^{nd} edition.
In French.

Tanenbaum, A. (2006).
Architecture de l'ordinateur.
Dunod.

Composants de stockage de données



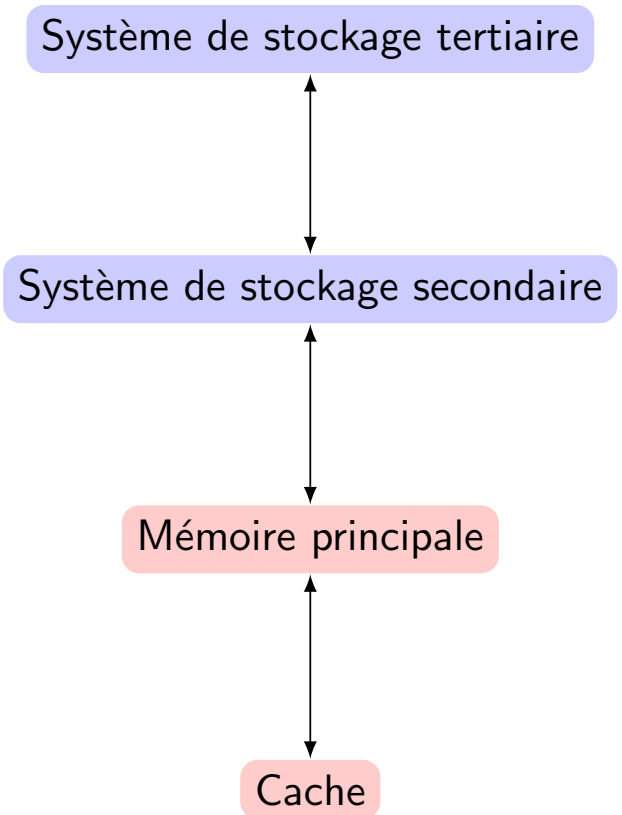
Composants de stockage de données



Cache :

- stockage **volatile**
- capacité de 1 MB
- temps d'accès : 1 ns

Composants de stockage de données



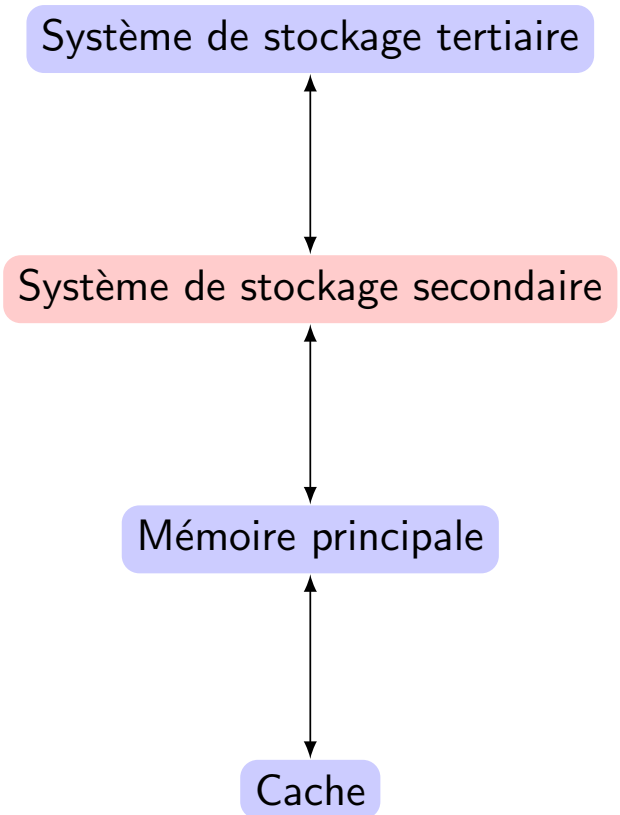
Cache :

- stockage **volatile**
- capacité de 1 MB
- temps d'accès : 1 ns

Mémoire principale :

- mémoire **volatile**
- appelée également RAM
- accédée par adresses
- capacité $>$ 1 GB
- temps d'accès : 10-100 ns

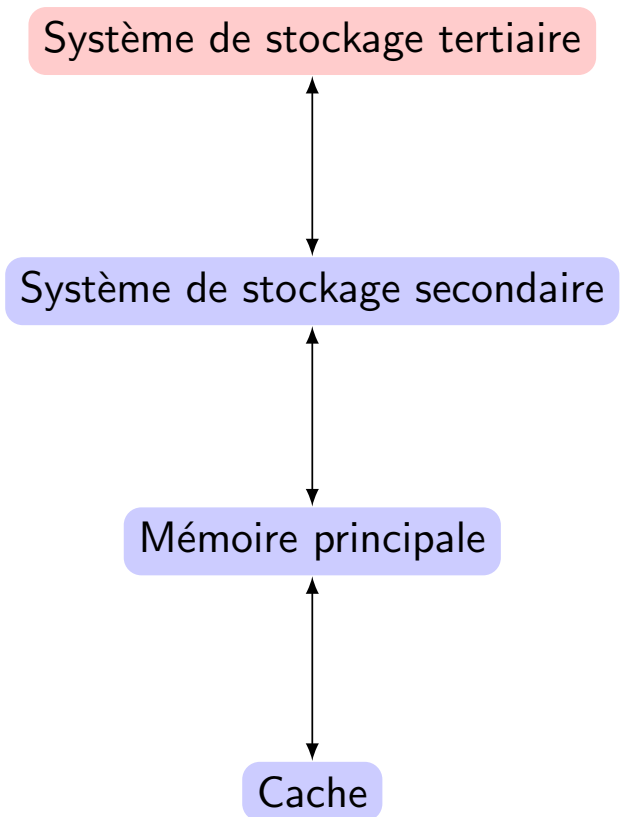
Composants de stockage de données



Stockage secondaire :

- mémoire **permanente**
- capacité $>$ 1TB
- temps d'accès : 15 ms

Composants de stockage de données



Stockage secondaire :

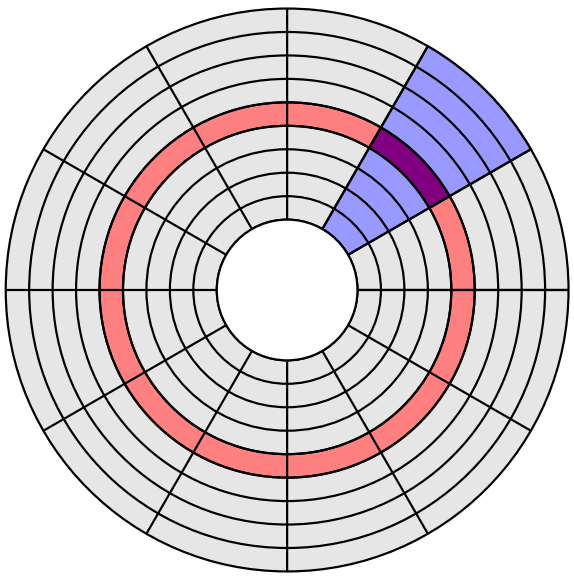
- mémoire **permanente**
- capacité $>$ 1TB
- temps d'accès : 15 ms

Stockage tertiaire :

- mémoire **permanente**
- bandes magnétiques
- disques magnétiques

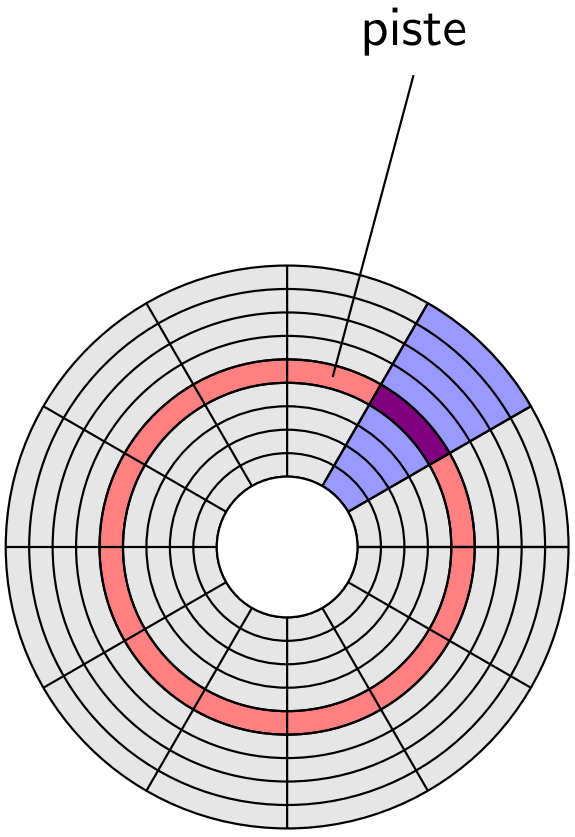
Stockage secondaire

Réalisé sur des disques durs magnétiques :



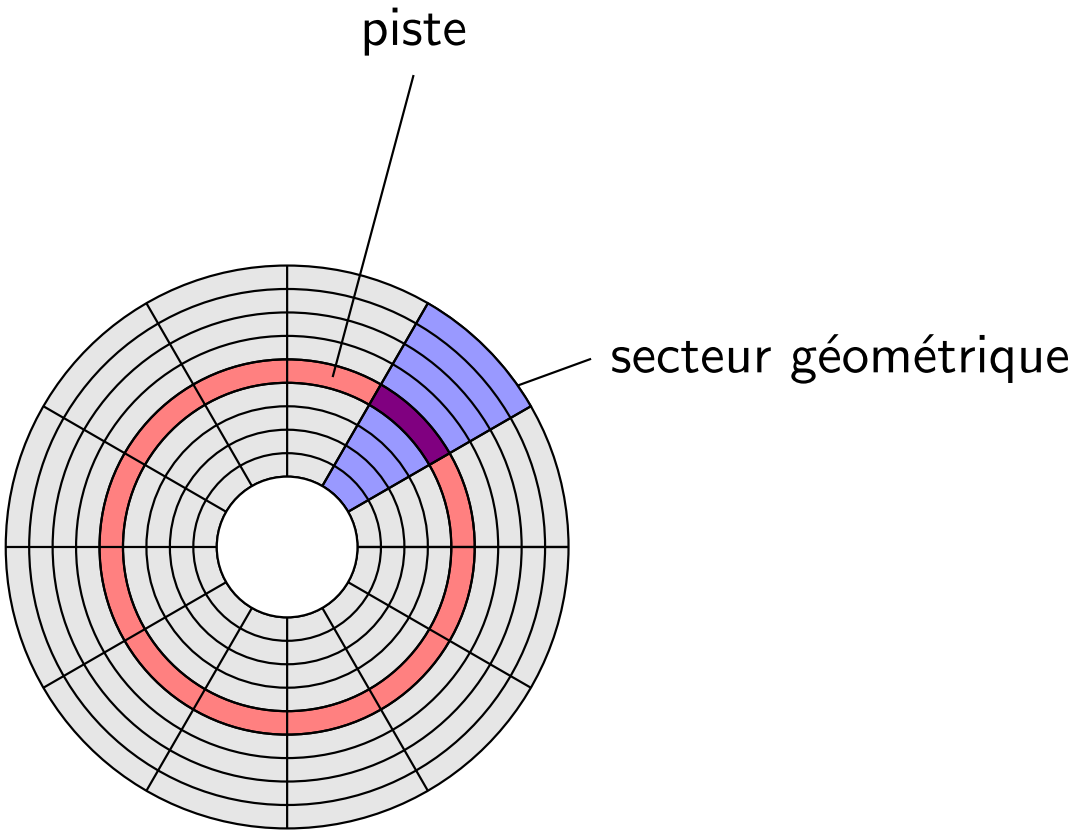
Stockage secondaire

Réalisé sur des disques durs magnétiques :



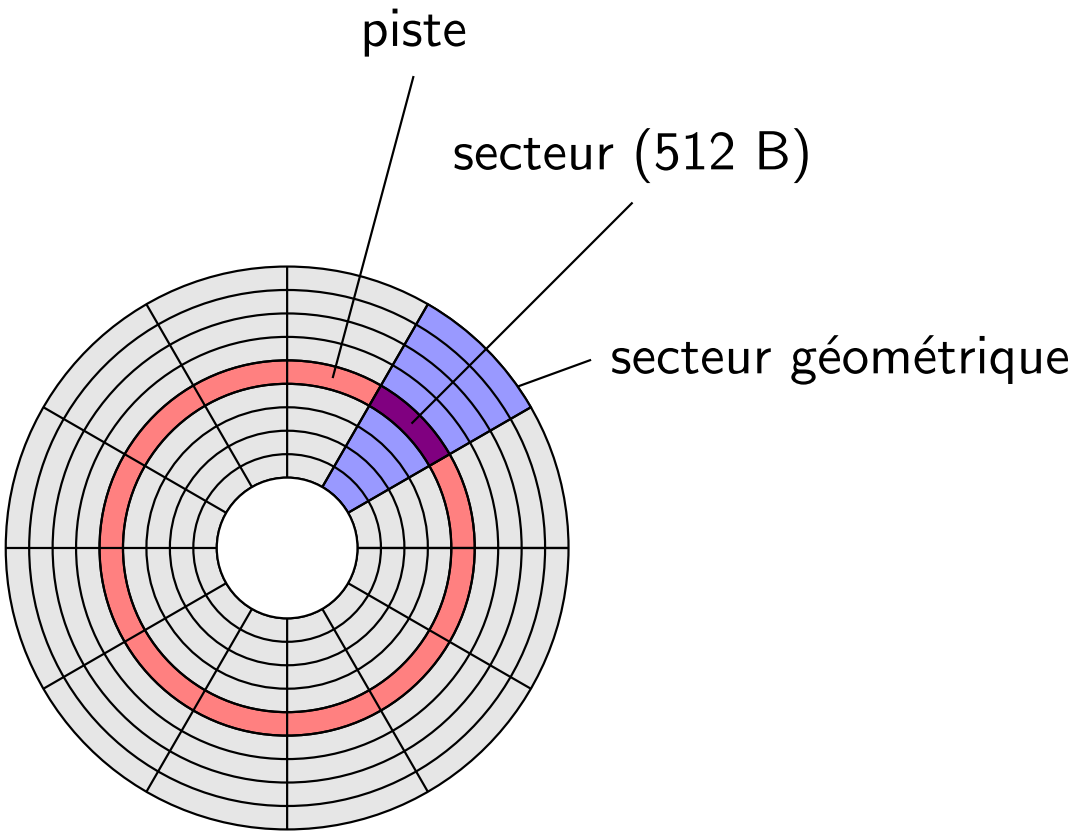
Stockage secondaire

Réalisé sur des disques durs magnétiques :



Stockage secondaire

Réalisé sur des disques durs magnétiques :



Caractéristiques des disques durs

Lecture/écriture

- unité **atomique** : le secteur (512 B)
- transfert vers la mémoire principale par **blocs** (4 kB la plupart du temps)
- lecture/écriture d'un bloc : 15 ms

Caractéristiques des disques durs

Lecture/écriture

- unité **atomique** : le secteur (512 B)
- transfert vers la mémoire principale par **blocs** (4 kB la plupart du temps)
- lecture/écriture d'un bloc : 15 ms

Mécanisme de mémoire virtuelle

- pour pouvoir faire tourner des applications « en parallèle »
- espace d'adressage sur 32 bits : **4 GB**
- utilise le stockage secondaire : le *swap* (Unix), système de pagination

Caractéristiques des disques durs

Lecture/écriture

- unité **atomique** : le secteur (512 B)
- transfert vers la mémoire principale par **blocs** (4 kB la plupart du temps)
- lecture/écriture d'un bloc : 15 ms

Mécanisme de mémoire virtuelle

- pour pouvoir faire tourner des applications « en parallèle »
- espace d'adressage sur 32 bits : **4 GB**
- utilise le stockage secondaire : le *swap* (Unix), système de pagination

Fichiers et systèmes de fichiers

- fichier : collection de données enregistrée sur un support physique
- fichiers **physiques** et fichiers **logiques**
- systèmes de fichiers : organisation (nom, emplacement), fourniture de primitives

Caractéristiques des disques durs

Lecture/écriture

- unité **atomique** : le secteur (512 B)
- transfert vers la mémoire principale par **blocs** (4 kB la plupart du temps)
- lecture/écriture d'un bloc : 15 ms

Mécanisme de mémoire virtuelle

- pour pouvoir faire tourner des applications « en parallèle »
- espace d'adressage sur 32 bits : **4 GB**
- utilise le stockage secondaire : le *swap* (Unix), système de pagination

Fichiers et systèmes de fichiers

- fichier : collection de données enregistrée sur un support physique
- fichiers **physiques** et fichiers **logiques**
- systèmes de fichiers : organisation (nom, emplacement), fourniture de primitives

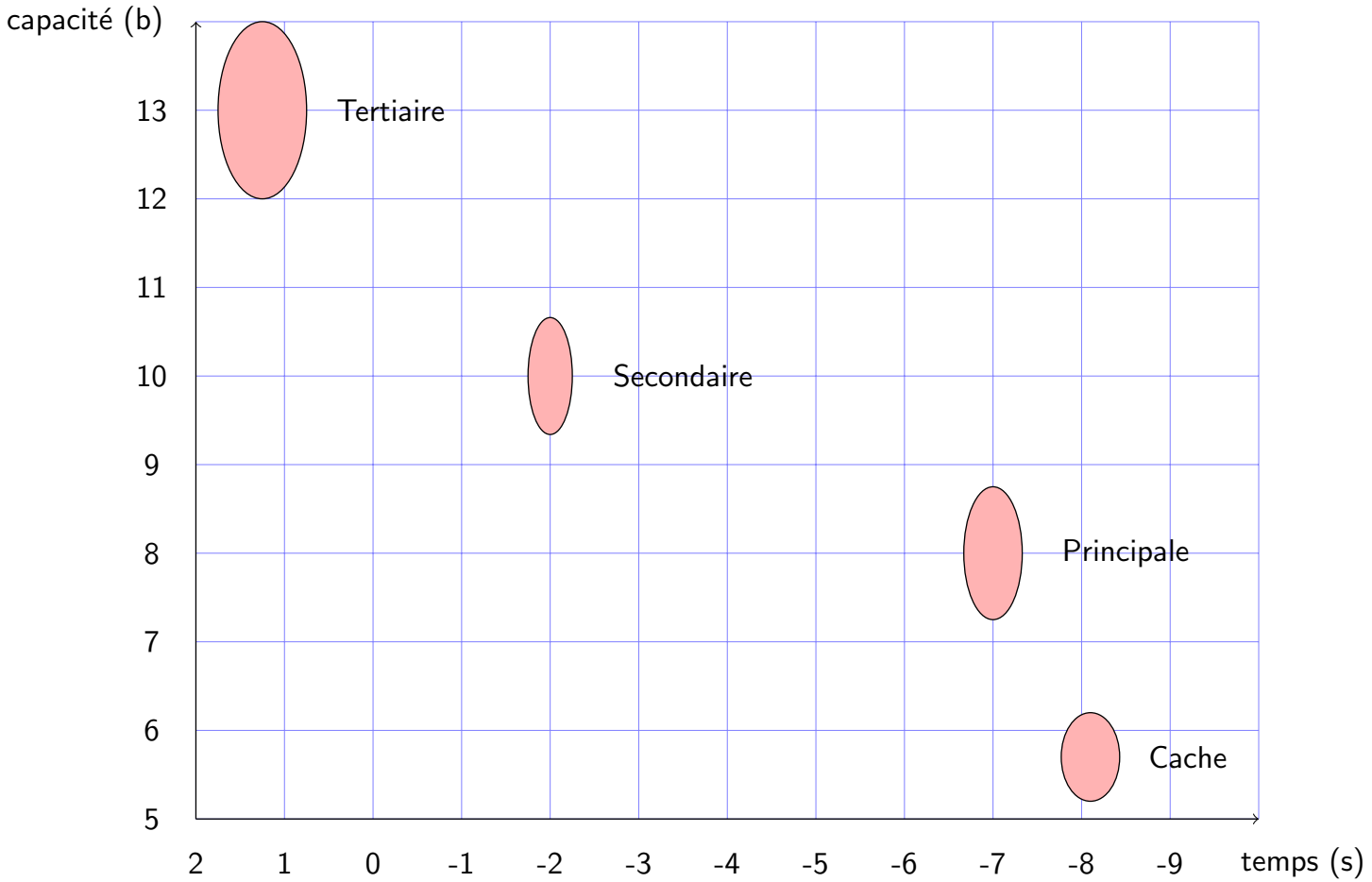
Gérer les erreurs (MTBF $\approx$ 6 mois !)

- *checksums*
- systèmes RAID

Deuxième partie

Stockage de l'information

Vue d'ensemble



Problème de l'utilisation du stockage secondaire

Problèmes

- mémoire principale trop petite
- 15 ms de transfert $\rightarrow$ des millions d'instructions !

Problème de l'utilisation du stockage secondaire

Problèmes

- mémoire principale trop petite
- 15 ms de transfert $\rightarrow$ des millions d'instructions !

Il va falloir optimiser

- à bas niveau (disque dur)

Problème de l'utilisation du stockage secondaire

Problèmes

- mémoire principale trop petite
- 15 ms de transfert $\rightarrow$ des millions d'instructions !

Il va falloir optimiser

- à bas niveau (disque dur)
- mais également au niveau algorithmique !

Problème de l'utilisation du stockage secondaire

Problèmes

- mémoire principale trop petite
- 15 ms de transfert $\rightarrow$ des millions d'instructions !

Il va falloir optimiser

- à bas niveau (disque dur)
- mais également au niveau algorithmique !

Exemple : tri de données

- algorithme « efficace » : *Quicksort* ($O(n \log n)$ en moyenne)
- pas efficace si les données ne sont pas toutes en mémoire principale
- utilisation du tri/fusion ($O(n \log n)$ mais $O(n)$ en mémoire)

Algorithme de tri/fusion

Algorithme 5.1 : Algorithme de tri/fusion d'une liste L de n éléments

entrées : une liste L d'éléments non triés de longueur n
sortie : une liste de longueur n contenant les éléments de L triés

```
1  si n = 1 alors
2  |   retourner l ;
3  sinon
4  |   pour i ← 1 to n/2 ou (n+1)/2 faire
5  |   |   S [i ] ← L[i ] ;
6  |   fin
7  |   pour i ← n/2 ou i ← (n+1)/2 to n faire
8  |   |   S' [i ] ← L[i ] ;
9  |   fin
10 |   Trier(S) ;
11 |   Trier(S') ;
12 |   tant que S ≠ ∅ et S' ≠ ∅ faire
13 |   |   e1 ← premier élément de S ;
14 |   |   e2 ← premier élément de S' ;
15 |   |   si e1 ≤ e2 alors
16 |   |   |   output = output ∪ e1 ;
17 |   |   |   retirer e1 de S ;
18 |   |   sinon
19 |   |   |   output = output ∪ e2 ;
20 |   |   |   retirer e2 de S' ;
21 |   |   fin
22 |   fin
23 |   si S = ∅ alors
24 |   |   output = output + S' ;
25 |   sinon
26 |   |   output = output + S ;
27 |   fin
28 |   retourner output ;
29 fin
```

Tri/fusion : exemple

étape	L_1	L_2	sortie
0	$\{2, 4, 8, 10\}$	$\{1, 5, 12, 13\}$	$\{\}$

Tri/fusion : exemple

étape	L_1	L_2	sortie
0	$\{2, 4, 8, 10\}$	$\{1, 5, 12, 13\}$	$\{\}$
1	$\{2, 4, 8, 10\}$	$\{5, 12, 13\}$	$\{1\}$

Tri/fusion : exemple

étape	L_1	L_2	sortie
0	$\{2, 4, 8, 10\}$	$\{1, 5, 12, 13\}$	$\{\}$
1	$\{2, 4, 8, 10\}$	$\{5, 12, 13\}$	$\{1\}$
2	$\{4, 8, 10\}$	$\{5, 12, 13\}$	$\{1, 2\}$

Tri/fusion : exemple

étape	L_1	L_2	sortie
0	$\{2, 4, 8, 10\}$	$\{1, 5, 12, 13\}$	$\{\}$
1	$\{2, 4, 8, 10\}$	$\{5, 12, 13\}$	$\{1\}$
2	$\{4, 8, 10\}$	$\{5, 12, 13\}$	$\{1, 2\}$
3	$\{8, 10\}$	$\{5, 12, 13\}$	$\{1, 2, 4\}$

Tri/fusion : exemple

étape	L_1	L_2	sortie
0	$\{2, 4, 8, 10\}$	$\{1, 5, 12, 13\}$	$\{\}$
1	$\{2, 4, 8, 10\}$	$\{5, 12, 13\}$	$\{1\}$
2	$\{4, 8, 10\}$	$\{5, 12, 13\}$	$\{1, 2\}$
3	$\{8, 10\}$	$\{5, 12, 13\}$	$\{1, 2, 4\}$
4	$\{8, 10\}$	$\{12, 13\}$	$\{1, 2, 4, 5\}$

Tri/fusion : exemple

étape	L_1	L_2	sortie
0	$\{2, 4, 8, 10\}$	$\{1, 5, 12, 13\}$	$\{\}$
1	$\{2, 4, 8, 10\}$	$\{5, 12, 13\}$	$\{1\}$
2	$\{4, 8, 10\}$	$\{5, 12, 13\}$	$\{1, 2\}$
3	$\{8, 10\}$	$\{5, 12, 13\}$	$\{1, 2, 4\}$
4	$\{8, 10\}$	$\{12, 13\}$	$\{1, 2, 4, 5\}$
5	$\{10\}$	$\{12, 13\}$	$\{1, 2, 4, 5, 8\}$

Tri/fusion : exemple

étape	L_1	L_2	sortie
0	$\{2, 4, 8, 10\}$	$\{1, 5, 12, 13\}$	$\{\}$
1	$\{2, 4, 8, 10\}$	$\{5, 12, 13\}$	$\{1\}$
2	$\{4, 8, 10\}$	$\{5, 12, 13\}$	$\{1, 2\}$
3	$\{8, 10\}$	$\{5, 12, 13\}$	$\{1, 2, 4\}$
4	$\{8, 10\}$	$\{12, 13\}$	$\{1, 2, 4, 5\}$
5	$\{10\}$	$\{12, 13\}$	$\{1, 2, 4, 5, 8\}$
6	$\{\}$	$\{12, 13\}$	$\{1, 2, 4, 5, 8, 10\}$

Tri/fusion : exemple

étape	L_1	L_2	sortie
0	$\{2, 4, 8, 10\}$	$\{1, 5, 12, 13\}$	$\{\}$
1	$\{2, 4, 8, 10\}$	$\{5, 12, 13\}$	$\{1\}$
2	$\{4, 8, 10\}$	$\{5, 12, 13\}$	$\{1, 2\}$
3	$\{8, 10\}$	$\{5, 12, 13\}$	$\{1, 2, 4\}$
4	$\{8, 10\}$	$\{12, 13\}$	$\{1, 2, 4, 5\}$
5	$\{10\}$	$\{12, 13\}$	$\{1, 2, 4, 5, 8\}$
6	$\{\}$	$\{12, 13\}$	$\{1, 2, 4, 5, 8, 10\}$
7	$\{\}$	$\{\}$	$\{1, 2, 4, 5, 8, 10, 12, 13\}$

Deuxième partie

Stockage de l'information

Organisation des données

- stockage des **attributs** : type informatique
- stockage des **enregistrements** : groupement logique d'informations

  ex : occurence d'entité

Organisation des données

- stockage des **attributs** : type informatique
- stockage des **enregistrements** : groupement logique d'informations

  ex : occurence d'entité

Enregistrements :

- stockés dans des blocs
- problème de la taille : plus petit, plus grand

Organisation des données

- stockage des **attributs** : type informatique
- stockage des **enregistrements** : groupement logique d'informations

  ex : occurence d'entité

Enregistrements :

- stockés dans des blocs
- problème de la taille : plus petit, plus grand

Accès aux données :

- mémoire principale : mémoire virtuelle
- sinon sur les supports **adressables** : adresse **physique** et adresse **logique** (table de correspondance)
- problème des pointeurs

Organisation des données

- stockage des **attributs** : type informatique
- stockage des **enregistrements** : groupement logique d'informations

  ex : occurence d'entité

Enregistrements :

- stockés dans des blocs
- problème de la taille : plus petit, plus grand

Accès aux données :

- mémoire principale : mémoire virtuelle
- sinon sur les supports **adressables** : adresse **physique** et adresse **logique** (table de correspondance)
- problème des pointeurs

Utilisation d'index

Troisième partie

Intérêt des systèmes de gestion de bases de données

Problèmes de la gestion de données

La gestion des données via des fichiers est problématique :

- redondance des informations
- dépendance logique
- dépendance physique

Problèmes de la gestion de données

La gestion des données via des fichiers est problématique :

- redondance des informations
- dépendance logique
- dépendance physique

Base de données :

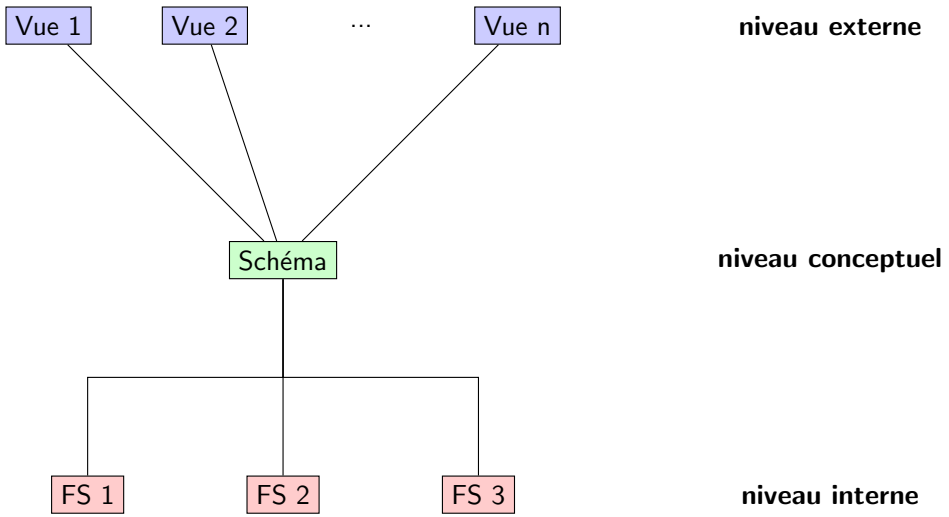
- représenter et exploiter les informations **logiques**
- système de fichiers élaboré avec primitives plus intéressantes

Systèmes de gestion de bases de données

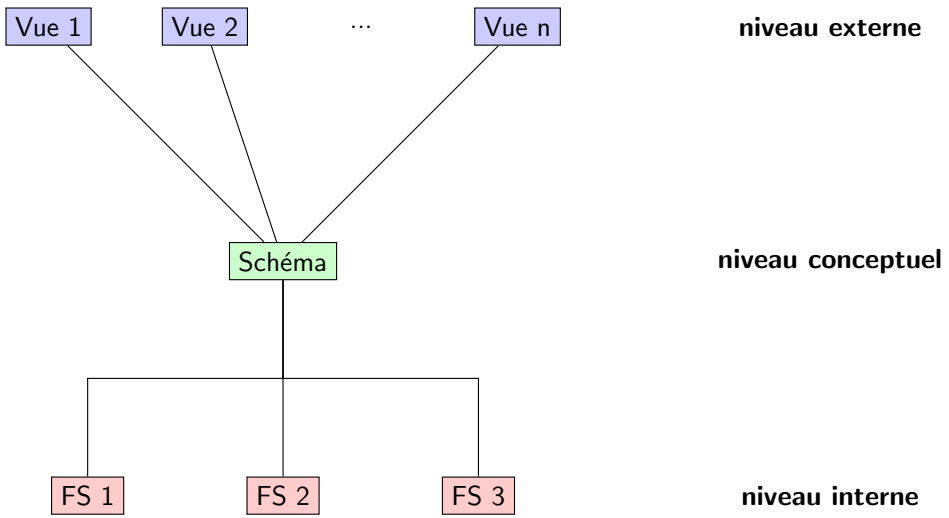
Spécifications du groupe CODASYL (*COnference on DAta SYstems Languages*) :

- le SGBD doit permettre l'accès à tout ou à une partie des fichiers suivant des critères donnés,
- le SGBD doit permettre des accès partagés aux données,
- le SGBD doit éviter (et éventuellement supprimer) les redondances,
- le SGBD doit réaliser une indépendance la plus grande possible entre les programmes qui exploitent la base de données et les fichiers qui représentent les données,
- le SGBD doit permettre une structuration optimale des données par rapport aux traitements qui seront effectués,
- le SGBD doit enfin permettre une réglementation de l'accès aux données.

Niveaux d'architecture

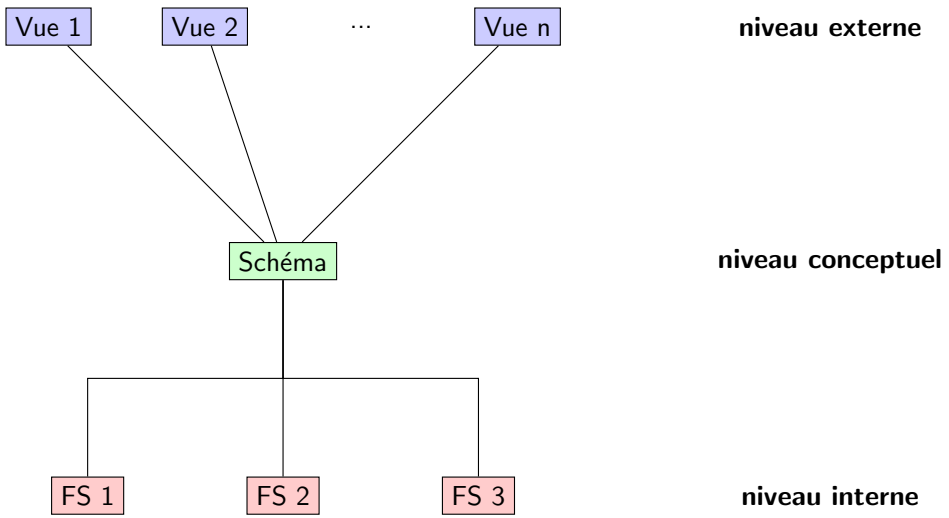


Niveaux d'architecture



externe fonctionne avec des vues (indépendance **logique**)

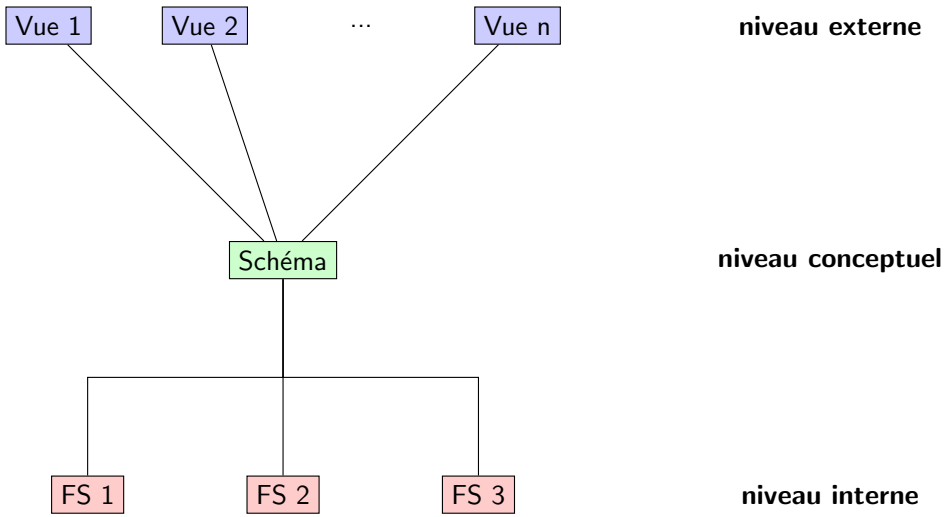
Niveaux d'architecture



externe fonctionne avec des vues (indépendance **logique**)

conceptuel regroupe toutes les informations de la base

Niveaux d'architecture



externe fonctionne avec des vues (indépendance **logique**)
conceptuel regroupe toutes les informations de la base
interne représentation physique (indépendance **physique**)

Langages associés au bases de données

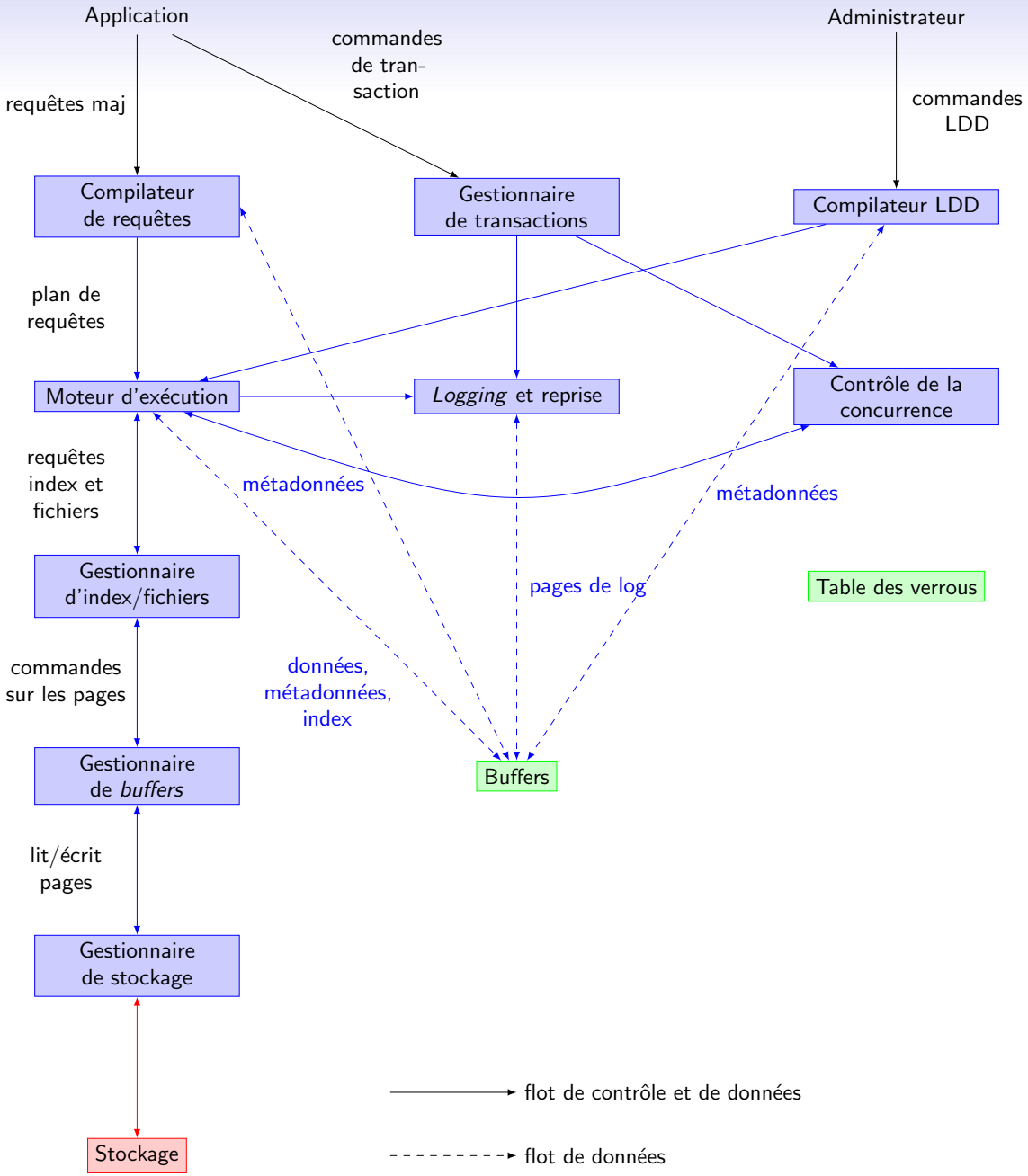
- **langage de définition de données**, encore appelé LDD ou DDL (*Data Definition Language*) : description des structures et des règles de cohérence
- **langage de manipulation de données**, encore appelé LMD ou DML (*Data Manipulation Language*) : manipulation des informations (ajout, suppression, modification et interrogation).

  On attend un langage d'interrogation **complet**.
- **langage de description des structures physiques** : niveau interne
- **langage d'expression de sélection** : langage d'interrogation
  - **procédural** : comment sélectionner des informations dans la base de données
  - **assertionnel** : quelles sont les informations à sélectionner

Caractéristiques des SGBD

- fonctions de base : dates, statistiques d'accès aux objets etc.
- indépendance des données et des programmes (**schéma**)
- non redondance
- partageabilité entre utilisateurs
- efficacité des échanges
- cohérence
- sécurité et fiabilité

Composants d'un SGBD



Quatrième partie

Le modèle relationnel

Quatrième partie

Le modèle relationnel

Modèle relationnel

Évolution du modèle hiérarchique dû à Codd.

Codd, E. J. (1970).
A relational model of data for large shared data banks.
Communications of the ACM, 13(6) :377–387.
Available on `http://www.acm.org/classics/nov95/toc.html`.

Codd, E. J. (1990).
The relational model for database management : version 2.
Addison-Wesley Publishing.

Modèle relationnel

Évolution du modèle hiérarchique dû à Codd.

Codd, E. J. (1970).
A relational model of data for large shared data banks.
Communications of the ACM, 13(6) :377–387.
Available on `http://www.acm.org/classics/nov95/toc.html`.

Codd, E. J. (1990).
The relational model for database management : version 2.
Addison-Wesley Publishing.

Très simple :

- relations : tableaux à deux dimensions
- éléments stockés : intersections ligne/colonne

Exemple

Ordinateur

ref	*nb_pos*	*capa_mem*	*nb_lect*	*nb_disk*	*prix*	*type*
10	1	512	2	1	10000	Micral 75
12	1	256	2	0	8000	Goupil G4
25	1	128	1	1	30000	Mac II

Type Ordinateur

type	*cons*	*pays*	*rev*	*nb_postes*	*mem_max*	*uc*	*capa_lec*	*capa_disk*
Micral 75	Bull	France	Camif	1	512	IN80486	1044	40
Mac II	Apple	USA	Apple	1	256	Mo68020	1044	60

Système

nom_sys	*concept*	*provenance*
MS_DOS	Microsoft	USA
UNIX	AT&T	USA

Logiciel de base

type	*nom_sys*
Micral 75	MS_DOS
Micral 75	UNIX
Micral 75	OS/2
Mac II	UNIX

Quatrième partie

Le modèle relationnel

En gros. . .

- une relation a un nom
- une colonne d'une relation est un **attribut**
- une ligne d'une relation est un **n-uplet** ou un **tuple**
- l'ordre des lignes et des colonnes n'a pas d'importance (ensemble)

En gros. . .

- une relation a un nom
- une colonne d'une relation est un **attribut**
- une ligne d'une relation est un **n-uplet** ou un **tuple**
- l'ordre des lignes et des colonnes n'a pas d'importance (ensemble)

Définition

Un domaine est un ensemble de valeurs que peut prendre un élément donné (souvent un attribut). Il sera représenté par un type.

Relations

Définition (relation)

Soient $\mathcal{R}$ une relation et $\mathcal{D}_1, \ldots, \mathcal{D}_n$ les domaines utilisés pour définir les colonnes de la relation (des domaines peuvent être identiques). Alors $\mathcal{R}$ peut être définie comme :

- *un sous-ensemble du produit cartésien $\mathcal{D}_1 \times \ldots \times \mathcal{D}_n$*
- *un ensemble $\{t_1, \ldots, t_m\}$ de n-uplets de valeurs tel que $\forall i \in \{1, \ldots, m\}$ $t_i = (d_{i,1}, \ldots, d_{i,n})$ où pour tout $j \in \{1, \ldots, n\}$ $d_{i,j}$ prend ses valeurs dans $\mathcal{D}_j$.*

On dira que m est la **cardinalité** *de $\mathcal{R}$.*
*On dira que n est l'***ordre** *de $\mathcal{R}$.*

Représentation d'une relation

	$\mathcal{D}_1$	...	$\mathcal{D}_i$	...	$\mathcal{D}_1$
t_1	$d_{1,1}$	...	$d_{1,i}$	...	$d_{1,n}$
...	...	...	...	...	...
t_j	$d_{j,1}$	...	$d_{j,i}$	...	$d_{j,n}$
...	...	...	...	...	...
t_m	$d_{m,1}$	...	$d_{m,i}$	...	$d_{m,n}$

Représentation d'une relation

	$\mathcal{D}_1$	...	$\mathcal{D}_i$	...	$\mathcal{D}_1$
t_1	$d_{1,1}$	...	$d_{1,i}$	...	$d_{1,n}$
...	...	...	...	...	...
t_j	$d_{j,1}$	...	$d_{j,i}$	...	$d_{j,n}$
...	...	...	...	...	...
t_m	$d_{m,1}$	...	$d_{m,i}$	...	$d_{m,n}$

Exemple :

	Micral 75	Mac II	Goupil G4
MS_Dos	×		
UNIX	×	×	
OS/2	×		

Attributs et superclés

Définition (attribut)

Un attribut est une fonction nommée d'une relation dans un de ses domaines.

Attributs et superclés

Définition (attribut)

Un attribut est une fonction nommée d'une relation dans un de ses domaines.

Définition (superclé)

Soit $\mathcal{R}$ une relation possédant n attributs $Att_1, \ldots, Att_n$. L'ensemble **non vide** *d'attributs $\{Att_{c_1}, \ldots, Att_{c_m}\} \subseteq \{Att_1, \ldots, Att_n\}$ est une* **superclé** *de $\mathcal{R}$ si et seulement si :*
Étant donnés deux tuples $t_1 = (d_{1,1}, \ldots, d_{1,n})$ et $t_2 = (d_{2,1}, \ldots, d_{2,n})$ de $\mathcal{R}$, alors $t_1 = t_2$ si et seulement si $\forall i \in \{1, \ldots, m\}$ $d_{1,c_i} = d_{2,c_i}$.

Propriétés des superclés et clé

- toute relation possède au moins une superclé
- tous les attributs d'une relation dépendent fonctionnellement des superclés de la relation
- si deux n-uplets contiennent la même valeur de clé, alors ils sont identiques.

Propriétés des superclés et clé

- toute relation possède au moins une superclé
- tous les attributs d'une relation dépendent fonctionnellement des superclés de la relation
- si deux n-uplets contiennent la même valeur de clé, alors ils sont identiques.

Définition (clé)

Soit $\{Att_{c_1}, \ldots, Att_{c_m}\}$ *une superclé d'une relation* $\mathcal{R}$. *Si* $\forall i \in \{1, \ldots, m\}$ $\{Att_{c_1}, \ldots, Att_{c_m}\} - \{Att_{c_i}\}$ *n'est pas une superclé de* $\mathcal{R}$, *alors* $\{Att_{c_1}, \ldots, Att_{c_m}\}$ *est une* **clé** *de* $\mathcal{R}$.

Schéma relationnel d'une base de données

Définition (schéma d'une relation)

Le schéma d'une relation est définie de façon intensionnelle par l'énoncé de :

- *son nom ;*
- *ses attributs ;*
- *des domaines de valeurs de ses attributs ;*

On note des deux façons suivantes la relation $\mathcal{R}$, d'attributs $Att_1, \ldots, Att_n$ de domaines respectifs $\mathcal{D}_1, \ldots, \mathcal{D}_n$:

$$\mathcal{R} \quad \{Att_1 : \mathcal{D}_1, \ldots, Att_n : \mathcal{D}_n\}$$
$$\mathcal{R}(\{Att_1 : \mathcal{D}_1, \ldots, Att_n : \mathcal{D}_n\})$$

Schéma relationnel d'une base de données

Définition (clé primaire)

Soit $\mathcal{R}$ une relation. La clé primaire de $\mathcal{R}$ est une clé $\{Att_1, \ldots, Att_m\}$ de $\mathcal{R}$. Pour représenter cette clé primaire, on souligne les attributs correspondants dans la relation.

Schéma relationnel d'une base de données

Définition (clé primaire)

Soit $\mathcal{R}$ une relation. La clé primaire de $\mathcal{R}$ est une clé $\{Att_1, \ldots, Att_m\}$ de $\mathcal{R}$. Pour représenter cette clé primaire, on souligne les attributs correspondants dans la relation.

Définition (schéma relationnel)

Le schéma relationnel d'une base de données est défini par l'ensemble des schémas des relations qui composent la base et un ensemble de contraintes appelé contraintes d'intégrité.

Clés étrangères

Définition (clé étrangère)

Soit $R(\Delta)$ une relation. On dit que $\{Att_1, \ldots, Att_n\} \subseteq \Delta$ sont des clés étrangères référençant les attributs $\{Att'_1, \ldots, Att'_n\}$ d'une relation R' ssi les valeurs prises par le tuple $\langle Att_1, \ldots, Att_n \rangle$ ne peuvent être que des valeurs du tuple $\langle Att'_1, \ldots, Att'_n \rangle$ apparaissant dans des n-uplets de R'. On le note de la façon suivante :

$$R \quad \{\ldots, \underline{Att_1, \ldots, Att_n}\}$$

$$\{Att_1, \ldots, Att_n\} \text{ référence } R' \ \{Att'_1, \ldots, Att'_n\}$$

Quatrième partie

Le modèle relationnel

Principe

- les **relations-entités** issues des classes d'entités ;
- les **relations-associations** issues des classes d'associations.

Principe

- les **relations-entités** issues des classes d'entités ;
- les **relations-associations** issues des classes d'associations.

Principe

1. *pour chaque classe d'entités, on définit une relation composée*
   - *des attributs et des domaines de la classe d'entités ;*
   - *de la clé primaire de la classe d'entités.*
2. *pour chaque classe d'associations, on définit une relation composée*
   - *des attributs et des domaines de la classe d'associations ;*
   - *de la clé primaire de la classe d'associations. Cette clé est composée des clés primaires de chacune des classes d'entités en liaison et des attributs propres de l'association.*

Algorithme de traduction d'une entité

Algorithme 9.1 : Algorithme permettant de traduire une classe d'entités en relation

entrée : une classe d'entités de nom E
sortie : une relation représentant la classe d'entités

construire une relation R :

- de nom E
- possédant pour attributs ceux de la classe d'entités E
- ayant pour clé primaire celle de la classe d'entités E

retourner R ;

Algorithme de traduction d'une association

Algorithme 9.2 : Algorithme permettant de traduire une classe d'associations en relation(s)

entrées : une classe d'associations A entre des classes d'entités de noms $E_1, \ldots, E_p$
sortie : une relation correspondant à A ou une relation correspondant à une classe d'entité modifiée

pour $i \in \{1, \ldots, p\}$ **faire**
  $r_i \leftarrow$ nom du rôle joué par E_i dans A ;
fin
si aucun des r_i **n'est de cardinalité** $[0,1]$ **ou** $[1,1]$ **alors**
  construire une relation :
  - de nom A
  - d'attributs ceux de la classe d'association A
  - possédant pour clé primaire le regroupement des clés étrangères que constituent les clés des relations $E_1, \ldots, E_p$

fin
si il existe r_j **tq sa cardinalité soit** $[0,1]$ **ou** $[1,1]$ **alors**
  - ajouter dans E_j les attributs de la classe d'associations A
  - ajouter dans E_j les clés des relations $E_1, \ldots, E_{j-1}, E_{j+1}, \ldots, E_p$ comme clés étrangères

fin
retourner la relation A si elle existe ou la relation E_j

Simplification du schéma relationnel

Algorithme 9.3 : Simplification d'un schéma relationnel

entrée : un schéma relationnel SR
sortie : un schéma relationnel dont les clés sont simplifiées et les relations éventuellement regroupées

pour chaque relation r **dont la clé primaire est composée de plusieurs attributs faire**
  si la clé de r **n'est pas minimale alors**
    simplifier la clé de r ;
  fin
fin
si deux relations E_1 **et** E_2 **ont la même clé primaire** $\{Att_1, \ldots, Att_n\}$ **alors**
  regrouper E_1 et E_2 en une seule relation possédant :
  - tous les attributs de E_1 et E_2 autres que la clé primaire
  - $\{Att_1, \ldots, Att_n\}$ comme clé primaire

fin
retourner SR

Traduction d'une entité faible

Algorithme 9.4 : Traduction d'une entité faible en relation

entrée : une entité faible E liée à des entités supports $E_1, \ldots, E_n$ par des associations supports $A_1, \ldots, A_p$
sortie : une relation R représentant E

```
1 placer les attributs de E dans R ;
2 pour chaque i ∈ {1, ..., n} faire
3 |   ajouter la clé de E_i à R ;
4 fin
5 retourner R ;
```

Cinquième partie

Dépendance fonctionnelles, normalisation de relations

Objectifs

Objectif

Construction de schémas relationnels optimaux

Objectifs

Objectif

Construction de schémas relationnels optimaux

type	*cons*	*pays*	*rev*	*nb_postes*	*mem_max*	*uc*	*capa_lec*	*capa_disk*
Micral 75	Bull	France	Camif	1	512	IN80486	1044	40
Micral 60	Bull	France	Camif	1	256	IN80486	1044	20
Mac II	Apple	USA	Apple	1	256	Mo68020	1044	60

Objectifs

Objectif

Construction de schémas relationnels optimaux

type	*cons*	*pays*	*rev*	*nb_postes*	*mem_max*	*uc*	*capa_lec*	*capa_disk*
Micral 75	Bull	France	Camif	1	512	IN80486	1044	40
Micral 60	Bull	France	Camif	1	256	IN80486	1044	20
Mac II	Apple	USA	Apple	1	256	Mo68020	1044	60

Plusieurs problèmes :

- information sur Bull apparaît deux fois
- mise-à-jour de Bull $\rightarrow$ incohérence ?
- effacement du troisième tuple $\rightarrow$ perte d'information

Cinquième partie

Dépendance fonctionnelles, normalisation de relations

Définition

Définition

Soient R une relation, $i \in \mathbb{N}$ tel que $i \geq 1$, $A_1, \ldots, A_n$ n attributs de R et B un attribut de R. Soient t_1 et t_2 deux tuples de R. Il existe une **dépendance fonctionnelle** *(notée DF) entre $A_1, \ldots, A_n$ et B ssi pour tous tuples t_1 et t_2 de R, si t_1 et t_2 s'accordent sur les valeurs des attributs $A_1, \ldots, A_n$, alors ils ont la même valeur pour l'attribut B.*
On le note $R \models A_1 \ldots A_n \longrightarrow B$.
On dit également que $A_1 \ldots A_n$ détermine fonctionnellement B dans R.

Conséquence

Définition

Soit S un ensemble de dépendances fonctionnelles sur une relation R. Une dépendance fonctionnelle f sur R est dit **conséquence** *de S si et seulement si $R \models S$ implique $R \models f$. On le note $S \models f$.*
Un ensemble de dépendances fonctionnelles T sur R est **conséquence** *de S si et seulement si pour toute dépendance $f \in T$, $S \models f$. On le note $S \models T$.*

Par exemple, $\{X \longrightarrow Y\} \models XZ \longrightarrow YZ$.

Cinquième partie

Dépendance fonctionnelles, normalisation de relations

Séparation/combinaison

Proposition

Soient R une relation et $A_1, \ldots, A_n, B_1, \ldots, B_m$ des attributs de R. Alors :

- *$A_1 \ldots A_n \longrightarrow B_1 \ldots B_m$ peut être remplacée par les m dépendances fonctionnelles $A_1 \ldots A_n \longrightarrow B_j$ pour $j \in \{1, \ldots, m\}$;*
- *les m dépendances fonctionnelles $A_1 \ldots A_n \longrightarrow B_j$ pour $j \in \{1, \ldots, m\}$ peuvent être remplacées par la dépendance fonctionnelle $A_1 \ldots A_n \longrightarrow B_1 \ldots B_m$.*

Dépendances fonctionnelles triviales

Définition

Soit $A_1 \dots A_n \longrightarrow B_1 \dots B_m$ *une dépendance fonctionnelle. On dit qu'elle est :*

- **triviale** *si* $\{B_1, \dots, B_m\} \subseteq \{A_1, \dots, A_n\}$;
- **non triviale** *si* $\exists j \in \{1, \dots, m\}$ *tel que* $B_j \notin \{A_1, \dots, A_n\}$;
- **complètement non triviale** *si pour tout* $j \in \{1, \dots, m\}$ $B_j \notin \{A_1, \dots, A_n\}$.

Système formel d'Armstrong

Définition (axiome de réflexivité)

$$\textit{Pour tout } \{B_1, \ldots, B_m\} \subseteq \{A_1, \ldots, A_n\} \quad \vdash A_1 \ldots A_n \longrightarrow B_1 \ldots B_m$$

Système formel d'Armstrong

Définition (axiome de réflexivité)

$$\textit{Pour tout } \{B_1, \ldots, B_m\} \subseteq \{A_1, \ldots, A_n\} \quad \vdash A_1 \ldots A_n \longrightarrow B_1 \ldots B_m$$

Définition (règle d'augmentation)

Si $A_1 \ldots A_n \longrightarrow B_1 \ldots B_m$ *alors* $A_1 \ldots A_n C_1 \longrightarrow B_1 \ldots B_m C_1$. *On note :*

$$A_1 \ldots A_n \longrightarrow B_1 \ldots B_m \ \vdash \ A_1 \ldots A_n C_1 \longrightarrow B_1 \ldots B_m C_1$$

Système formel d'Armstrong

Définition (règle de transitivité)

Si $A_1 \dots A_n \longrightarrow B_1 \dots B_m$ *et* $B_1 \dots B_m \longrightarrow C_1 \dots C_k$ *sont deux dépendances fonctionnelles de* R*, alors* $A_1 \dots A_n \longrightarrow C_1 \dots C_k$*. On la note :*

$$\{A_1 \dots A_n \longrightarrow B_1 \dots B_m, B_1 \dots B_m \longrightarrow C_1 \dots C_k\} \vdash A_1 \dots A_n \longrightarrow C_1 \dots C_k$$

Dérivation

Définition (dérivation de dépendance fonctionnelle)

Soient S un ensemble de dépendance fonctionnelle et f une dépendance fonctionnelle. On dit que S **dérive** *f ou que f* **est déduite de** *S, noté $S \vdash f$ si et seulement si une suite de dépendances fonctionnelles $f_1, \ldots, f_n$ telle que :*

- $f_n = f$
- $\forall i \in \{1, \ldots, n\}$ $f_i \in S$ *ou* f_i *est déduite de* $\{f_1, \ldots, f_{i-1}\}$ *en utilisant le système d'Armstrong.*

$f_1, \ldots, f_n$ est appelée une **dérivation** *de f à partir de S.*

Propriété du système d'Armstrong

Théorème

Soit S un ensemble de dépendances fonctionnelles sur une relation R.

- *si f peut être dérivée de S en utilisant les règles d'Armstrong, alors f est une dépendance fonctionnelle sur R conséquence de S (validité des règles).*
- *si f est une dépendance fonctionnelle sur R conséquence de S, alors f peut dérivée de S en utilisant les règles d'Armstrong (complétude des règles).*

On peut donc écrire :

$$S \models f \Leftrightarrow S \vdash f$$

Fermeture d'un ensemble d'attributs

Définition

Soient $\{A_1, \ldots, A_n\}$ *un ensemble d'attributs et* S *un ensemble de dépendances fonctionnelles.* $\{A_1, \ldots, A_n\}^+$, *appelé* **fermeture de** $\{A_1, \ldots, A_n\}$ *est l'ensemble des attributs* B *tels que toute relation satisfaisant les dépendances fonctionnelles de* S *satisfait également* $A_1 \ldots A_n \longrightarrow B$.

Algorithme de calcul de fermeture

Algorithme 11.1 : Calcul de la fermeture d'un ensemble d'attributs X relativement à un ensemble de dépendances fonctionnelles S

entrées : un ensemble d'attributs X et un ensemble de dépendances fonctionnelles S

sortie : la fermeture de X par rapport à S

```
1  closure ← X ;
2  répéter
3  |   card ← |closure | ;
4  |   pour chaque Y ⟶ Z ∈ S faire
5  |   |   si Y ⊆ closure alors
6  |   |   |   closure ← closure ∪ {Z} ;
7  |   |   fin
8  |   fin
9  jusqu'à card = |closure | ;
10 retourner closure ;
```

Validité et complétude de l'algorithme

Théorème

L'algorithme de calcul de fermeture est valide et complet. Plus précisément, soient S un ensemble de dépendances fonctionnelles sur une relation R et $\{A_1, \ldots, A_n\}$ des attributs de R. Alors $B \in \{A_1, \ldots, A_n\}^+$ si et seulement si $A_1 \ldots A_n \longrightarrow B$ peut être déduite de S.

Validité et complétude de l'algorithme

Théorème

L'algorithme de calcul de fermeture est valide et complet. Plus précisément, soient S un ensemble de dépendances fonctionnelles sur une relation R et $\{A_1, \ldots, A_n\}$ des attributs de R. Alors $B \in \{A_1, \ldots, A_n\}^+$ si et seulement si $A_1 \ldots A_n \longrightarrow B$ peut être déduite de S.

Remarque

Complexité de l'algorithme en $O(|S| \times |R|)$

DF élémentaires et redondantes

Définition

Soient R une relation et $A_1, \ldots, A_n, B$ des attributs de R tel que l'on ait $A_1 \ldots A_n \longrightarrow B$. $A_1 \ldots A_n \longrightarrow B$ est une dépendance fonctionnelle élémentaire si et seulement si il n'existe pas $\{A_i, \ldots, A_j\} \subset \{A_1, \ldots, A_n\}$ tel que l'on ait $A_i \ldots A_j \longrightarrow B$ pour R.

DF élémentaires et redondantes

Définition

Soient R une relation et $A_1, \ldots, A_n, B$ des attributs de R tel que l'on ait $A_1 \ldots A_n \longrightarrow B$. $A_1 \ldots A_n \longrightarrow B$ est une dépendance fonctionnelle élémentaire si et seulement si il n'existe pas $\{A_i, \ldots, A_j\} \subset \{A_1, \ldots, A_n\}$ tel que l'on ait $A_i \ldots A_j \longrightarrow B$ pour R.

Définition

$X \longrightarrow A$ est redondante pour l'ensemble de dépendances fonctionnelles S ssi $A \in \{X\}^+$ avec $\{X\}^+$ calculé à partir de $S - \{X \longrightarrow A\}$.

Calcul de couverture irredondante

Algorithme 11.2 : Calcul de la couverture irredondante d'un ensemble de dépendances fonctionnelles S

entrées : un ensemble de dépendances fonctionnelles S
sortie : la couverture irredondante de S

1 S1 $\leftarrow$ décomposer les DF de S pour n'avoir qu'un seul attribut à droite ;
2 S2 $\leftarrow$ éliminer les attributs en surnombre à gauche dans les DF de S1 ;
3 S3 $\leftarrow$ éliminer les DF redondantes dans S2 ;
4 **retourner** S3 ;

Cinquième partie

Dépendance fonctionnelles, normalisation de relations

Décomposition de relation

Objectif

$R = R_1 \bowtie \ldots \bowtie R_n$.

Décomposition de relation

Objectif

$R = R_1 \bowtie \ldots \bowtie R_n$.

Deux propriétés à vérifier :

- $R_1 \bowtie \ldots \bowtie R_n$ doit permettre de calculer exactement la relation initiale : en particulier, on doit retrouver tous les tuples de R (et pas plus !)
  - ➥ sans perte d'information (SPI)

Décomposition de relation

Objectif

$R = R_1 \bowtie \ldots \bowtie R_n$.

Deux propriétés à vérifier :

- $R_1 \bowtie \ldots \bowtie R_n$ doit permettre de calculer exactement la relation initiale : en particulier, on doit retrouver tous les tuples de R (et pas plus !)
  - ➥ sans perte d'information (SPI)
- les dépendances fonctionnelles de R doivent être préservées par la jointure des sous-relations (SPD)
  - ➥ sans perte de dépendances (SPD)

SPI : algorithme de poursuite

Algorithme 12.1 : Algorithme appliquant la méthode de poursuite

entrées : une décomposition $(R_1, \ldots, R_n$ d'une relation $R(A_1, \ldots, A_m)$ et un ensemble de dépendances fonctionnelles S sur R
sortie : vrai si la décomposition est SPI

```
1  construire une matrice T de dimensions m × n où T_{i,j} représente la valeur de A_i pour R_j ;
2  pour i ← 1 to m faire
3  |    pour j ← 1 to n faire
4  |    |    si A_i ∈ R_j alors T_{i,j} ← a_i ;
5  |    |    sinon T_{i,j} ← x_{i,j} ;
6  |    fin
7  fin
8  répéter
9  |    pour chaque X ⟶ Y ∈ S faire
10 |    |    pour i ← 1 to m faire
11 |    |    |    pour j ← (i + 1) to m faire
12 |    |    |    |    si T_i et T_j sont identiques sur les attributs de X alors
13 |    |    |    |    |    pour chaque A_k ∈ Y faire
14 |    |    |    |    |    |    si T_{i,k} ou T_{j,k} est une constante alors
15 |    |    |    |    |    |    |    remplacer la variable par la constante a_k ;
16 |    |    |    |    |    |    sinon
17 |    |    |    |    |    |    |    choisir x_{i,k} comme variable commune dans les deux lignes ;
18 |    |    |    |    |    |    fin
19 |    |    |    |    |    fin
20 |    |    |    |    fin
21 |    |    |    fin
22 |    |    fin
23 |    fin
24 jusqu'à (a_1, ..., a_n) ∈ T ou que le tableau n'évolue plus ;
25 si (a_1, ..., a_n) ∈ T alors retourner vrai ;
26 sinon retourner faux ;
```

Algorithme de poursuite : exemple

$R(A, B, C, D)$, $A \longrightarrow B$ et $B \longrightarrow CD$.
Décomposition de R en (A, B) et (B, C, D).

Algorithme de poursuite : exemple

$R(A, B, C, D)$, $A \longrightarrow B$ et $B \longrightarrow CD$.
Décomposition de R en (A, B) et (B, C, D).

	A	B	C	D
(A, B)	a_1	a_2	$x_{1,3}$	$x_{1,4}$
(B, C, D)	$x_{2,1}$	a_2	a_3	a_4

Algorithme de poursuite : exemple

$R(A, B, C, D)$, $A \longrightarrow B$ et $B \longrightarrow CD$.
Décomposition de R en (A, B) et (B, C, D).

	A	B	C	D
(A, B)	a_1	a_2	$x_{1,3}$	$x_{1,4}$
(B, C, D)	$x_{2,1}$	a_2	a_3	a_4

	A	B	C	D
(A, B)	a_1	a_2	a_3	a_4
(B, C, D)	$x_{2,1}$	a_2	a_3	a_4

Algorithme de poursuite : exemple

$R(A, B, C, D)$, $A \longrightarrow B$ et $B \longrightarrow CD$.
Décomposition de R en (A) et (B, C, D).

Algorithme de poursuite : exemple

$R(A, B, C, D)$, $A \longrightarrow B$ et $B \longrightarrow CD$.
Décomposition de R en (A) et (B, C, D).

	A	B	C	D
(A)	a_1	$x_{1,2}$	$x_{1,3}$	$x_{1,4}$
(B, C, D)	$x_{2,1}$	a_2	a_3	a_4

SPI : quelques théorèmes

Théorème

Soit $R(\Delta)$ une relation telle que $X \longrightarrow Y$ soit vraie. Alors $R(\Delta) = R(X, Y) \bowtie_X R(\Delta - \{Y\})$.

Théorème

Soit R une relation et $\{R_1, R_2\}$ une décomposition de R. Si $R_1 \cap R_2 \longrightarrow R_1 - R_2$ ou $R_1 \cap R_2 \longrightarrow R_2 - R_1$ est une dépendance fonctionnelle sur R, alors $\{R_1, R_2\}$ est SPI.

Théorème

Soit R une relation et $\{R_1, \ldots, R_n\}$ une décomposition SPI de R par rapport à un ensemble de DF S. Si $\{R_{1,1}, R_{1,2}\}$ est une décomposition SPI de R_1 par rapport à F, alors $\{R_{1,1}, R_{1,2}, R_2, \ldots, R_n\}$ est une décomposition SPI de R par rapport à F.

Décomposition SPD

Définition (décomposition SPD)

Une décomposition R est sans perte de dépendances par rapport à un ensemble de dépendances fonctionnelles S s'il existe $F \subseteq S^+$ tel que :

- *pour toute $X \longrightarrow Y \in F$ il existe $R_i \in R$ telle que $XY \subseteq R_i$*
- $F^+ = S^+$

Décomposition SPD : algorithme

Algorithme 12.2 : Algorithme SPD

entrées : une décomposition $\{R_1, \ldots, R_n\}$ d'une relation $R(A_1, \ldots, A_m)$ et un ensemble de dépendances fonctionnelles S sur R

sortie : vrai si la décomposition est SPD

```
1 spd ← vrai ;
2 tant que spd et il existe X ⟶ Y ∈ S non traitée faire
3 |   Z ← X ;
4 |   répéter
5 |   |   pour i ← 1 to n faire Z ← Z ∪ ((Z ∩ R_i)^+ ∩ R_i) ;
6 |   jusqu'à Y ⊆ Z ou Z ne change pas ;
7 |   si Y ⊈ Z alors spd ← faux ;
8 fin
9 retourner spd ;
```

Première forme normale

Définition (1NF)

Soit R une relation d'attributs $A_1, \ldots, A_n$. R est sous première forme normale (ou 1NF) ssi pour tout $i \in \{1, \ldots, n\}$ les A_i a un domaine atomique.

Seconde forme normale

Définition (2NF)

Soit R une relation. R est sous seconde forme normale (ou 2NF) ssi R est sous première forme normale et si tout attribut B n'appartenant pas à une clé de la relation est en dépendance élémentaire avec toutes les clés de la relation.

Troisième forme normale

Définition (3NF)

Soit R une relation. R est sous troisième forme normale (ou 3NF) ssi R est sous seconde forme normale et si tout attribut B n'appartenant pas à une clé de la relation ne dépend pas d'un attribut non clé.

Théorème

Toute relation admet une décomposition en 3NF avec jointure conservatrice et préservation des dépendances fonctionnelles.

Sixième partie

Algèbre relationnelle

Introduction

- langage d'extraction et de modification du contenu d'une BDR
  - langage procédural
  - langage assertionnel

Introduction

- langage d'extraction et de modification du contenu d'une BDR
  - langage procédural
  - langage assertionnel
- algèbre relationnelle : base pour tous les langages assertionnels

Introduction

- langage d'extraction et de modification du contenu d'une BDR
  - langage procédural
  - langage assertionnel
- algèbre relationnelle : base pour tous les langages assertionnels
- algèbre : un ensemble d'éléments atomiques muni d'opérateurs sans effets de bord
  - éléments atomiques : relations et variables
  - différents types d'opérateurs

Introduction

- langage d'extraction et de modification du contenu d'une BDR
  - langage procédural
  - langage assertionnel
- algèbre relationnelle : base pour tous les langages assertionnels
- algèbre : un ensemble d'éléments atomiques muni d'opérateurs sans effets de bord
  - éléments atomiques : relations et variables
  - différents types d'opérateurs
- requête : expression de l'algèbre relationnelle

Sixième partie

Algèbre relationnelle

Hypothèse sur les relations

Hypothèse

Soient R et S deux relations. Pour appliquer une opération ensembliste à R et S, il faut et il suffit que :

- *R et S aient des schémas avec des attributs identiques et de même domaine ;*
- *les colonnes de R et S doivent être ordonnées de la même façon.*

Union, intersection et différence

Syntaxe (union)

(MA) $R \cup S$

(IN) ***UNION***(R,S)

Syntaxe (intersection)

(MA) $R \cap S$

(IN) $INTER(R,S)$

Syntaxe (différence)

(MA) $R - S$

(IN) $DIFF(R,S)$

Sixième partie

Algèbre relationnelle

Projection

Syntaxe

(MA) $\pi_{A_i,A_j,\ldots,A_l}(R)$

(IN) `PROJ(R, Ai, Aj,..., Al)`

Projection

Syntaxe

(MA) $\pi_{A_i,A_j,\ldots,A_l}(R)$

(IN) `PROJ(R, Ai, Aj,..., Al)`

Ordinateur

ref	*nb_pos*	*capa_mem*	*nb_lect*	*nb_disk*	*prix*	*type*
10	1	512	2	1	10000	Micral 75
12	1	256	2	0	8000	Goupil G4
25	1	128	1	1	30000	Mac II

Projection

Syntaxe

(MA) $\pi_{A_i,A_j,\ldots,A_l}(R)$

(IN) `PROJ(R, Ai, Aj,..., Al)`

Ordinateur

ref	*nb_pos*	*capa_mem*	*nb_lect*	*nb_disk*	*prix*	*type*
10	1	512	2	1	10000	Micral 75
12	1	256	2	0	8000	Goupil G4
25	1	128	1	1	30000	Mac II

$\pi_{ref,\ type}(Ordinateur)$

Projection

Syntaxe

(MA) $\pi_{A_i,A_j,\ldots,A_l}(R)$

(IN) `PROJ(R, Ai, Aj,..., Al)`

Ordinateur

ref	*nb_pos*	*capa_mem*	*nb_lect*	*nb_disk*	*prix*	*type*
10	1	512	2	1	10000	Micral 75
12	1	256	2	0	8000	Goupil G4
25	1	128	1	1	30000	Mac II

$\pi_{ref,\ type}(Ordinateur)$

ref	*type*
10	Micral 75
12	Goupil G4
25	Mac II

Sélection

Syntaxe

Soit P une expression logique impliquant les attributs de R. La syntaxe de l'opérateur de sélection est la suivante :

(MA) $\sigma_P(R)$

(IN) `SEL(R, P)`

Sélection

Syntaxe

Soit P une expression logique impliquant les attributs de R. La syntaxe de l'opérateur de sélection est la suivante :

(MA) $\sigma_P(R)$

(IN) `SEL(R, P)`

Logiciel

nom	*classe*	*concept*	*provenance*	*revendeur*	*prix*
Turbo Pascal	Compil	Borland	USA	Camif	1000
Oracle	SGBD	Oracle	USA	Oracle	15000

Sélection

Syntaxe

Soit P une expression logique impliquant les attributs de R. La syntaxe de l'opérateur de sélection est la suivante :

(MA) $\sigma_P(R)$

(IN) `SEL(R, P)`

Logiciel

nom	*classe*	*concept*	*provenance*	*revendeur*	*prix*
Turbo Pascal	Compil	Borland	USA	Camif	1000
Oracle	SGBD	Oracle	USA	Oracle	15000

$\sigma_{classe = \text{"SGBD"} \ \text{ET} \ prix < 1000}(Logiciel)$

Produit cartésien

Syntaxe

(MA)	$R \times S$
(IN)	`PROD(R, S)`

Produit cartésien

Syntaxe

(MA) $R \times S$

(IN) *PROD(R, S)*

R

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

S

D	E
d_1	e_1
d_2	e_2
d_3	e_3

Produit cartésien

Syntaxe

(MA) $R \times S$

(IN) *PROD(R, S)*

R

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

S

D	E
d_1	e_1
d_2	e_2
d_3	e_3

$R \times S$

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_1	b_1	c_1	d_2	e_2
a_1	b_1	c_1	d_3	e_3
a_2	b_2	c_2	d_1	e_1
a_2	b_2	c_2	d_2	e_2
a_2	b_2	c_2	d_3	e_3

Jointure naturelle

Syntaxe

(MA) $R \bowtie S$

(IN) **JOIN**(R, S)

Jointure naturelle

Syntaxe

(MA) $R \bowtie S$

(IN) ***JOIN***`(R, S)`

R

A	B
1	2
3	4

S

B	C	D
2	5	6
4	7	8
9	10	11

Jointure naturelle

Syntaxe

(MA) $R \bowtie S$

(IN) **JOIN**(R, S)

R

A	B
1	2
3	4

S

B	C	D
2	5	6
4	7	8
9	10	11

$R \bowtie S$

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

θ-jointure

Syntaxe

Soit P une expression logique impliquant les attributs de R ou de S. La syntaxe de l'opérateur de θ-jointure est la suivante :

(MA) $R \bowtie_P S$

(IN) `JOIN(R, S, P)`

θ-jointure

Syntaxe

Soit P une expression logique impliquant les attributs de R ou de S. La syntaxe de l'opérateur de θ-jointure est la suivante :

(MA) $R \bowtie_P S$

(IN) `JOIN(R, S, P)`

Théorème

$$\bowtie_P = \sigma_P \circ \times$$

θ-jointure

$$Système \bowtie_{type = \texttt{"Micral75"} \text{ AND } Système.nom\_sys = Logiciel\ de\ Base.nom\_sys} Logiciel\ de\ Base$$

θ-jointure

$Système \bowtie_{type = \text{"Micral75"} \text{ AND } Système.nom\_sys = Logiciel\ de\ Base.nom\_sys} Logiciel\ de\ Base$

1. construction de $Système \times Logiciel\ de\ Base$:

Système.nom_sys	*concept*	*provenance*	*type*	*Logiciel de Base.nom_sys*
MS_DOS	Microsoft	USA	Micral 75	MS_DOS
MS_DOS	Microsoft	USA	Micral 75	UNIX
MS_DOS	Microsoft	USA	Mac II	UNIX
UNIX	AT&T	USA	Micral 75	MS_DOS
UNIX	AT&T	USA	Micral 75	UNIX
UNIX	AT&T	USA	Mac II	UNIX

θ-jointure

$Système \bowtie_{type = \text{"Micral75"} \text{ AND } Système.nom\_sys = Logiciel\ de\ Base.nom\_sys} Logiciel\ de\ Base$

1. construction de $Système \times Logiciel\ de\ Base$:

Système.nom_sys	*concept*	*provenance*	*type*	*Logiciel de Base.nom_sys*
MS_DOS	Microsoft	USA	Micral 75	MS_DOS
MS_DOS	Microsoft	USA	Micral 75	UNIX
MS_DOS	Microsoft	USA	Mac II	UNIX
UNIX	AT&T	USA	Micral 75	MS_DOS
UNIX	AT&T	USA	Micral 75	UNIX
UNIX	AT&T	USA	Mac II	UNIX

2. sélection des tuples suivant le critère
   $type = \text{"Micral75"} \text{ AND } Système.nom\_sys = Logiciel\ de\ Base.nom\_sys$

Système.nom_sys	*concept*	*provenance*	*type*	*Logiciel de Base.nom_sys*
MS_DOS	Microsoft	USA	Micral 75	MS_DOS
UNIX	AT&T	USA	Micral 75	UNIX

Opérateur de renommage

Syntaxe

(MA) $\rho_{S(A_{i_1},\ldots,A_{i_n})}(R)$

(IN) `REN(R, S, (Ai1,..., Ain))`

Attention, il n'y a pas d'effet de bord.

Mises à jour

- l'insertion d'un tuple se fera au moyen de l'opérateur d'union. Par exemple :
  $\textit{Logiciel de Base} \cup (\texttt{MacII}, \texttt{MS\_DOS})$
  insère un nouveau tuple dans la relation *Logiciel de Base* ;
- la suppression d'un tuple se fera au moyen de l'opérateur de différence. Par exemple :
  $\textit{Logiciel de Base} - (\texttt{MacII}, \texttt{UNIX})$
  supprime un tuple dans la relation *Logiciel de Base* ;
- la modification de tuple se fait au moyen d'une suppression suivie d'une insertion.

Sixième partie

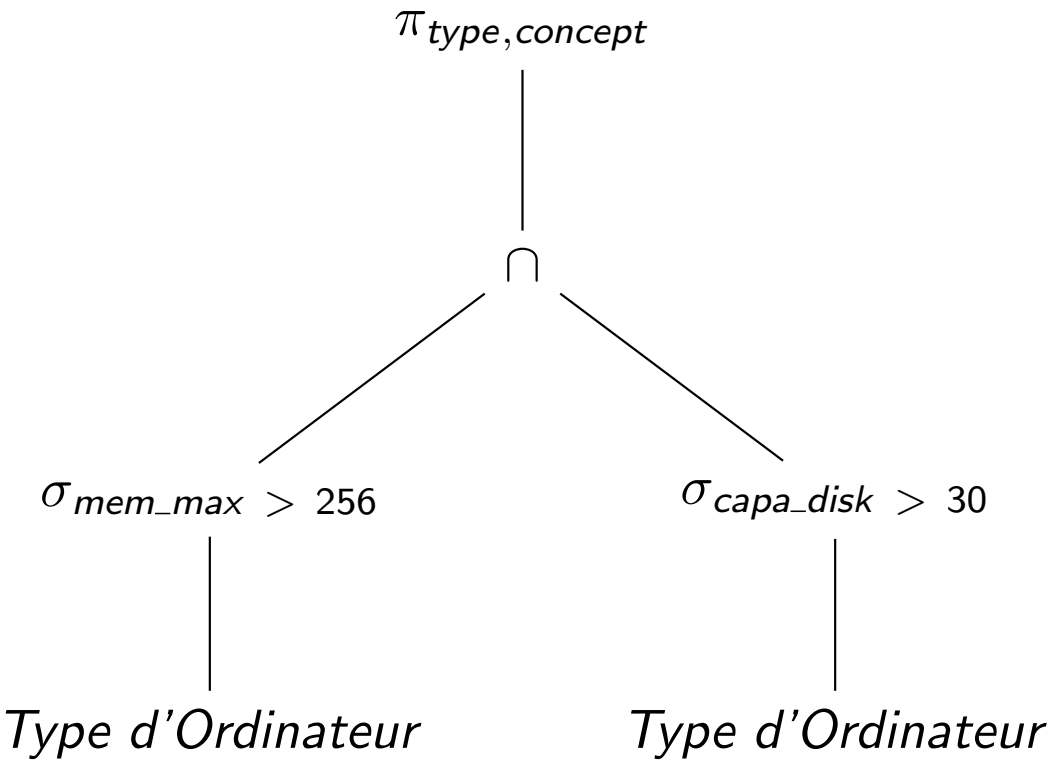
Algèbre relationnelle

Exemple

Quels sont les types d'ordinateurs et les concepteurs associés tels que la mémoire maximale soit supérieure à 256 Mo et la capacité du disque soit supérieure à 30 Go ?

1. sélection sur la table *Type d'Ordinateur* des ordinateurs dont la mémoire maximale est supérieure à 256 Mo, soit $\sigma_{mem\_max > 256}(Type\ d'Ordinateur)$;
2. sélection sur la table *Type d'Ordinateur* des ordinateurs dont la capacité du disque est supérieure à 30 Go, soit $\sigma_{capa\_disk > 30}(Type\ d'Ordinateur)$;
3. intersection des deux relations obtenues en 2 et 3 ;
4. projection de la relation obtenue en 4 sur les attributs *type* et *concept*.

Représentation sous forme d'arbre



Notation linéaire

Ajout d'un opérateur d'affectation

```
R := SEL(Type Ordinateur, mem_max>256)
S := SEL(Type Ordinateur, capa_disk>30)
T := INTER(R,S)
Reponse := PROJ(T, type, concept)
```

Sixième partie

Algèbre relationnelle

Opérateurs d'agrégation

Ce ne sont pas des opérateurs de l'algèbre relationnelle. Ils transforment une relation en une **valeur** unique.

- **SUM** qui permet de sommer une colonne contenant des valeurs numériques ;
- **AVG** qui permet de moyenner une colonne contenant des valeurs numériques ;
- **MIN** et **MAX** qui retourne respectivement la plus petite et la plus grande valeur d'une colonne ;
- **COUNT** retourne le nombre de valeurs d'une colonne.

Regroupements

Ils permettent d'appliquer un opérateur d'agrégation à des partitions d'une table.

Regroupements

Ils permettent d'appliquer un opérateur d'agrégation à des partitions d'une table.
Ces partitions sont en fait des classes d'équivalence d'une relation par rapport à un attribut.

Regroupements

Ils permettent d'appliquer un opérateur d'agrégation à des partitions d'une table.
Ces partitions sont en fait des classes d'équivalence d'une relation par rapport à un attribut.

Syntaxe

$$\gamma_L\ (R)$$

où L est une liste d'éléments qui peuvent être :

- *un attribut de la relation R. On dit que c'est un* **attribut de regroupement** *;*
- *un opérateur d'agrégation appliqué à un attribut de la relation. Le nom de l'attribut correspondant dans la relation résultat est précisé par le signe* $\rightarrow$*. L'attribut ainsi créé est appelé* **attribut d'agrégat**.

Regroupements

Définition

La relation retournée par $\gamma_L(R)$ est construite de la façon suivante :

1. *partitionnement des tuples de R. Chaque partition est composée des tuples ayant un assignement particulier pour les attributs de regroupement de L ;*
2. *pour chaque groupe, construire* **un** *tuple constitué :*
   1. *des attributs de regroupement de L ;*
   2. *des agrégations calculées sur les tuples du groupe des attributs d'agrégats de L*

Regroupements

Par exemple, considérons $\gamma_{classe,\ \texttt{MIN}(prix)\rightarrow prix\_bas}$ $(Logiciel)$

Regroupements

Par exemple, considérons $\gamma_{classe,\ \texttt{MIN}(prix)\rightarrow prix\_bas}\ (Logiciel)$

nom	*classe*	*concept*	*provenance*	*revendeur*	*prix*
Turbo Pascal	Compil	Borland	USA	Camif	1000
ANSI C	Compil	Borland	USA	Fnac	2000
ADA	Compil	IBM	USA	Fnac	1500
Oracle	SGBD	Oracle	USA	Oracle	15000
PostgreSQL	SGBD	GPL	USA	GPL	0

Regroupements

Par exemple, considérons $\gamma_{classe,\ \texttt{MIN}(prix)\rightarrow prix\_bas}\ (Logiciel)$

nom	*classe*	*concept*	*provenance*	*revendeur*	*prix*
Turbo Pascal	Compil	Borland	USA	Camif	1000
ANSI C	Compil	Borland	USA	Fnac	2000
ADA	Compil	IBM	USA	Fnac	1500
Oracle	SGBD	Oracle	USA	Oracle	15000
PostgreSQL	SGBD	GPL	USA	GPL	0

Regroupements

Par exemple, considérons $\gamma_{classe,\ \texttt{MIN}(prix)\rightarrow prix\_bas}\ (Logiciel)$

nom	*classe*	*concept*	*provenance*	*revendeur*	*prix*
Turbo Pascal	Compil	Borland	USA	Camif	1000
ANSI C	Compil	Borland	USA	Fnac	2000
ADA	Compil	IBM	USA	Fnac	1500
Oracle	SGBD	Oracle	USA	Oracle	15000
PostgreSQL	SGBD	GPL	USA	GPL	0

Regroupements

Par exemple, considérons $\gamma_{classe,\ \texttt{MIN}(prix)\rightarrow prix\_bas}$ (*Logiciel*)

nom	*classe*	*concept*	*provenance*	*revendeur*	*prix*
Turbo Pascal	Compil	Borland	USA	Camif	**1000**
ANSI C	Compil	Borland	USA	Fnac	2000
ADA	Compil	IBM	USA	Fnac	1500
Oracle	SGBD	Oracle	USA	Oracle	15000
PostgreSQL	SGBD	GPL	USA	GPL	0

Regroupements

Par exemple, considérons $\gamma_{classe,\ \texttt{MIN}(prix)\rightarrow prix\_bas}$ (*Logiciel*)

nom	*classe*	*concept*	*provenance*	*revendeur*	*prix*
Turbo Pascal	Compil	Borland	USA	Camif	**1000**
ANSI C	Compil	Borland	USA	Fnac	2000
ADA	Compil	IBM	USA	Fnac	1500
Oracle	SGBD	Oracle	USA	Oracle	15000
PostgreSQL	SGBD	GPL	USA	GPL	0

Regroupements

Par exemple, considérons $\gamma_{classe,\ \texttt{MIN}(prix)\rightarrow prix\_bas}\ (Logiciel)$

nom	*classe*	*concept*	*provenance*	*revendeur*	*prix*
Turbo Pascal	Compil	Borland	USA	Camif	**1000**
ANSI C	Compil	Borland	USA	Fnac	2000
ADA	Compil	IBM	USA	Fnac	1500
Oracle	SGBD	Oracle	USA	Oracle	15000
PostgreSQL	SGBD	GPL	USA	GPL	**0**

Regroupements

Par exemple, considérons $\gamma_{classe,\ \texttt{MIN}(prix)\rightarrow prix\_bas}$ (*Logiciel*)

nom	*classe*	*concept*	*provenance*	*revendeur*	*prix*
Turbo Pascal	Compil	Borland	USA	Camif	**1000**
ANSI C	Compil	Borland	USA	Fnac	2000
ADA	Compil	IBM	USA	Fnac	1500
Oracle	SGBD	Oracle	USA	Oracle	15000
PostgreSQL	SGBD	GPL	USA	GPL	**0**

classe	*prix_bas*
Compil	1000
SGBD	0

Sixième partie

Algèbre relationnelle

Conclusion

- requêtes très simples
- pas de structures de contrôle compliquées
- opérateurs étendus
- optimisation des requêtes :
  - $\pi_{nb\_lec}\ (\sigma_{nb\_lec} \leq 2\ (Ordinateur))$
  - $\sigma_{nb\_lec} \leq 2\ (\pi_{nb\_lec}\ (Ordinateur))$

Septième partie

SQL (*Structured Query Language*)

18. **Le SGBD PostgreSQL**
19. **Le langage de manipulation de données**
    - Obtention d'informations
    - Mise à jour des informations
20. **Le langage de description de données**
    - Création de relations
    - Modification du schéma d'une relation
    - Destruction d'une relation
21. **Le langage de contrôle de données**
22. **Contraintes et triggers**
    - Clés primaires et clés étrangères
    - Contraintes sur les attributs et tuples
    - Triggers
23. **PostgreSQL et standard SQL**

Introduction

Pour interroger une base de données relationnelle, on dispose de deux types de langages :

- les langages procéduraux comme le langage algébrique vu précédemment. Ce langage fondé sur une algèbre permet de décrire comment obtenir une relation correspondant au résultat d'une requête ;
- les langages déclaratifs, qui permettent d'exprimer la requête sous forme d'une assertion sans expliquer comment la trouver.

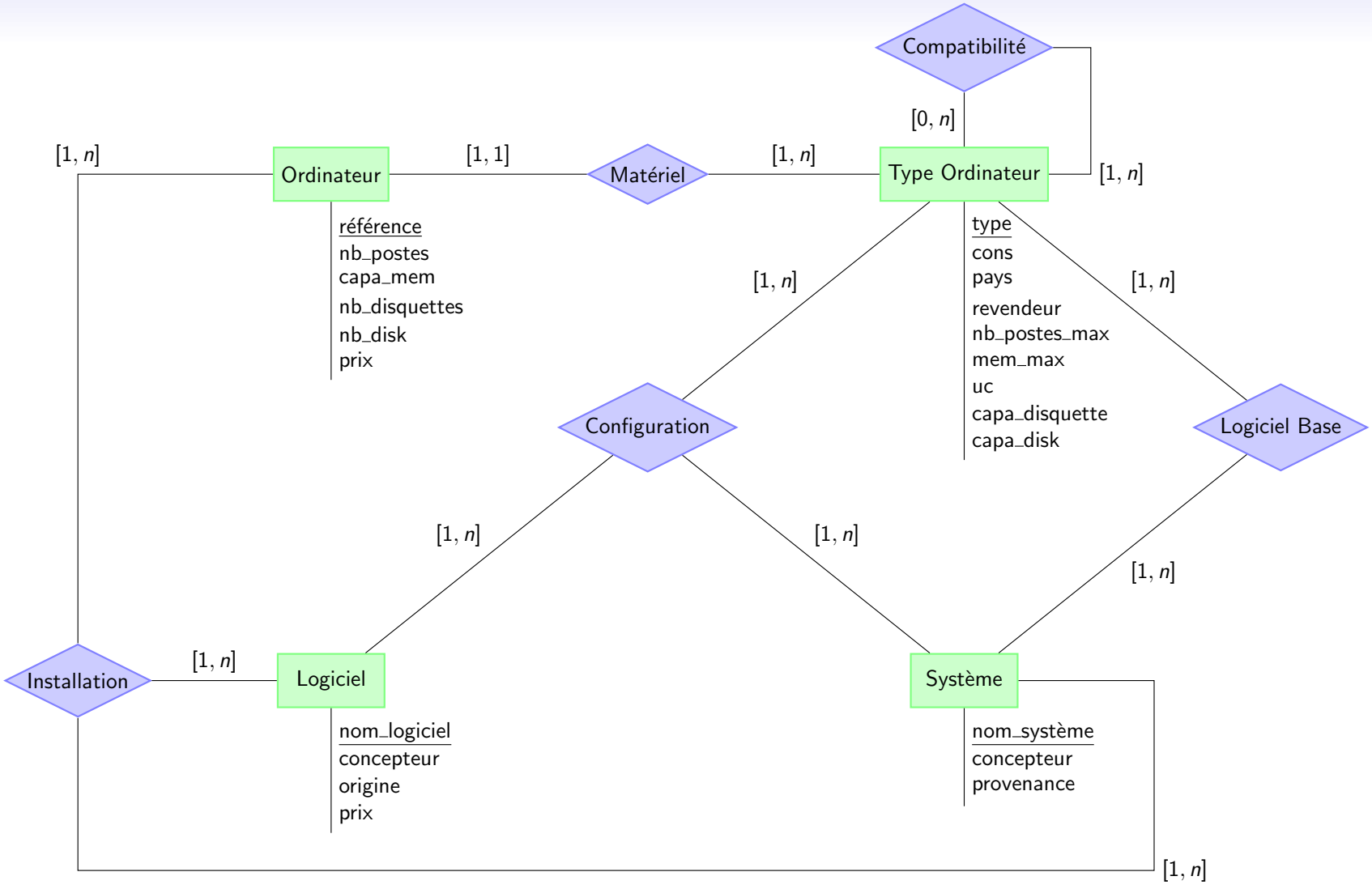
SQL est le langage déclaratif le plus utilisé actuellement.

Le langage SQL

- c'est un langage de **définition** et de **manipulation** de bases de données relationnelles ;
- c'est un langage standardisé (norme ANSI) ;
- principaux mots-clés :

DDL	DML	DCL
ALTER	**DELETE**	**GRANT**
CREATE	**INSERT**	**REVOKE**
COMMENT	**SELECT**	
DESCRIBE	**UPDATE**	
DROP		
RENAME		

Exemple utilisé : diagramme entité-association



Exemple utilisé : relations

- TypeOrdinateur(type, cons, pays, revendeur, nb_postes, mem_max, uc, capa_lec, capa_disk)
- Ordinateur(ref, nb_pos, capa_mem, nb_lect, nb_disk, prix, type)
- Système(nom_sys, concept, provenance)
- Logiciel(nom, classe, concept, provenance, revendeur, prix
- LogicielBase(type, nom_sys)
- Configuration(nom_log, nom_sys, type, mem_min, disk_min)
- Installation(ref, nom_log, nom_sys)
- Compatibilité(type_ref, type_comp)

Exemple utilisé : contenu des tables

TypeOrdinateur

type	cons	pays	rev	nb_postes	mem_max	uc	capa_lec	capa_disk
Micral 75	Bull	France	Camif	1	512	IN80486	1044	40
Mac II	Apple	USA	Apple	1	256	Mo68020	1044	60

Ordinateur

ref	nb_pos	capa_mem	nb_lect	nb_disk	prix	type
10	1	512	2	1	10000	Micral 75
12	1	256	2	0	8000	Goupil G4
25	1	128	1	1	30000	Mac II

Système

nom_sys	concept	provenance
MS_DOS	Microsoft	USA
UNIX	AT&T	USA

Logiciel

nom	classe	concept	provenance	revendeur	prix
Turbo Pascal	Compil	Borland	USA	Camif	1000
Oracle	SGBD	Oracle	USA	Oracle	15000

Exemple utilisé : contenu des tables

LogicielBase

type	nom_sys
Micral 75	MS_DOS
Micral 75	UNIX
Mac II	UNIX

Configuration

nom_log	nom_sys	type	mem_min	disk_min
Turbo Pascal	MS_DOS	Goupil G4	256	0
Oracle	UNIX	Micral 75	1500	1

Installation

ref	nom_log	nom_sys
10	Turbo Pascal	MS_DOS
10	DBaseIV	MS_DOS
25	Oracle	UNIX

Compatibilité

type_ref	type_comp
Micral 75	Goupil G4
Mac II	Mac Classic
IBM PS2	Micral 75

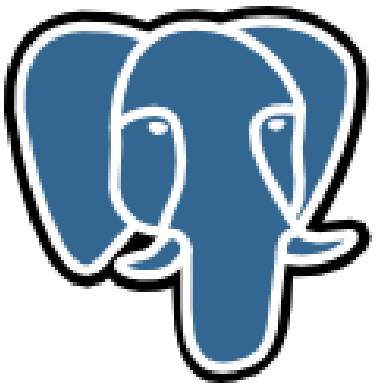
Septième partie

SQL (*Structured Query Language*)

Présentation de PostgreSQL

Caractéristiques :

- SGBD relationnel (objet)
- libre
- développé à Berkeley



Utilisateurs :

- Skype
- *State Departement*
- Fujitsu
- Cisco
- ...

PostgreSQL à SUPAERO

- une machine serveur : `postgresql`
- une base de données par élève de SID :
  - nom connexion : nom de l'élève
  - mot de passe : prénom de l'élève
  - nom de la base de données : dbnomélève
- une interface : pgadmin
- produits libres, disponibles sur toutes les plateformes

pgAdmin.
`http://www.pgadmin.org/`.

PostgreSQL.
`http://www.postgresql.org/`.

Septième partie

SQL (*Structured Query Language*)

Le bloc de qualification

La structure de base est le bloc de qualification :

```
SELECT Ai,..., An -- colonnes
FROM R            -- relation
WHERE F           -- assertion
GROUP BY A        -- regroupement
HAVING H          -- assertion
ORDER BY T        -- tri
;
```

Projection et sélection simple

Projection simple :

```
SELECT Ai,..., An
FROM R;
```

```
SELECT ref, type
FROM Ordinateur;
```

Projection et sélection simple

Projection simple :

```
SELECT Ai,..., An
FROM R;

SELECT ref, type
FROM Ordinateur;
```

Sélection simple :

```
SELECT *
FROM R
WHERE F;

SELECT * FROM Logiciel
WHERE (classe='SGBD') and (prix<10000);
```

Expression logique de sélection

Elle concerne des **constantes** et les **noms de colonnes** de R.

Expression logique de sélection

Elle concerne des **constantes** et les **noms de colonnes** de R.

Différents opérateurs peuvent intervenir :

- <, <=, =, >=, >, <> ;
- **AND**, **OR**, **NOT** ;
- **IN**, par exemple **IN** ('MS_DOS', 'UNIX') ;
- **BETWEEN**, par exemple **BETWEEN** 15 **AND** 30 ;
- **LIKE** en utilisant les jokers.

Expression logique de sélection

Elle concerne des **constantes** et les **noms de colonnes** de R.

Différents opérateurs peuvent intervenir :

- <, <=, =, >=, >, <> ;
- **AND**, **OR**, **NOT** ;
- **IN**, par exemple `IN ('MS_DOS', 'UNIX')` ;
- **BETWEEN**, par exemple `BETWEEN 15 AND 30` ;
- **LIKE** en utilisant les jokers.

Il existe des caractères spéciaux :

- _ remplace n'importe quel caractère ;
- % remplace n'importe quelle séquence de caractères ;
- **NULL** (attention, ce n'est pas une constante).

Tri et entêtes de colonnes

On peut changer le nom des entêtes de colonnes grâce à **AS** :

```
SELECT prix / 6.55957 AS "Prix Euro"
FROM Ordinateur;
```

Tri et entêtes de colonnes

On peut changer le nom des entêtes de colonnes grâce à **AS** :

```
SELECT prix / 6.55957 AS "Prix Euro"
FROM Ordinateur;
```

On peut trier les résultats d'une requête grâce à **ORDER BY**, **ASC** et **DESC** :

```
SELECT *
FROM Ordinateur
ORDER BY prix DESC, capa_mem ASC;
```

Si on ne connait pas le nom d'une colonne dans le résultat (exemple d'une agrégation), on peut utiliser le numéro de la colonne.

Opérations de produit

On peut effectuer des opérations de produits :

```
SELECT A1,..., An
FROM R1, R2, R3
WHERE F;
```

Par exemple :

```
SELECT Ordinateur.type, Ordinateur.prix,
       TypeOrdinateur.cons
FROM Ordinateur, TypeOrdinateur
WHERE (Ordinateur.ref = 10) AND
      (Ordinateur.type = TypeOrdinateur.type);
```

Alias

On peut faciliter l'écriture avec des alias :

```
SELECT N1.A1,..., Nn.An
FROM R1 (AS) N1,..., Rn (AS) Nn
WHERE (N1.A1 =...);
```

Par exemple :

```
SELECT O.type, O.prix, T.cons
FROM Ordinateur O, TypeOrdinateur T
WHERE (O.ref = 10) AND (O.type = T.type);
```

Produit et jointure

On peut utiliser l'opération de produit pour exprimer une jointure :

```
SELECT S.provenance
FROM Systeme S, LogicielBase L
WHERE (S.nom_sys = L.nom_sys)
      AND (L.type = 'Micral 75');
```

Produit et jointure

On peut utiliser l'opération de produit pour exprimer une jointure :

```
SELECT S.provenance
FROM Systeme S, LogicielBase L
WHERE (S.nom_sys = L.nom_sys)
      AND (L.type = 'Micral 75');
```

Depuis SQL2, on peut écrire une jointure explicitement :

```
SELECT S.provenance
FROM Systeme S JOIN LogicielBase L
     ON S.nom_sys = L.nom_sys
WHERE L.type = 'Micral 75';
```

Produit et jointure

Il existe différents types de jointures :

- **`CROSS JOIN`** qui est un produit cartésien ;
- **`INNER JOIN`** qui est une jointure classique ;
- **`LEFT OUTER JOIN`** qui permet de conserver **tous** les enregistrements de la première table ;
- **`RIGHT OUTER JOIN`** qui permet de conserver **tous** les enregistrements de la seconde table ;
- **`FULL OUTER JOIN`** qui permet de conserver **tous** les enregistrements.

Produit et jointure

Il existe différents types de jointures :

- **CROSS JOIN** qui est un produit cartésien ;
- **INNER JOIN** qui est une jointure classique ;
- **LEFT OUTER JOIN** qui permet de conserver **tous** les enregistrements de la première table ;
- **RIGHT OUTER JOIN** qui permet de conserver **tous** les enregistrements de la seconde table ;
- **FULL OUTER JOIN** qui permet de conserver **tous** les enregistrements.

Les conditions de jointures peuvent être :

- **ON** suivi d'une condition quelconque ;
- **USING** suivi des noms de colonnes communs aux deux tables ;
- **NATURAL** (qui précéde le mot-clé **JOIN**).

Opérations ensemblistes

Les opérations **INTERSECT**, **UNION** et **EXCEPT** permettent de travailler avec deux relations compatibles ayant la même structure.
Par exemple :

```
SELECT type_comp FROM Compatibilite
WHERE type_ref = 'Micral 75'
INTERSECT
(SELECT type_comp FROM Compatibilite
 WHERE type_ref = 'Mac II');
```

Opérateurs d'agrégation

Ces opérateurs permettent d'effectuer des opérations arithmétiques sur les résultats :

- **COUNT** compte les valeurs d'une colonne ;
- **COUNT**(*) compte les lignes d'une table ;
- **SUM** additionne les valeurs d'une colonne numérique ;
- **AVG** calcule la moyenne des valeurs d'une colonne ;
- **MIN** extrait la plus petite valeur d'une colonne ;
- **MAX** extrait la plus grande valeur d'une colonne.

On peut utiliser **DISTINCT** dans les requêtes.

Opérateurs d'agrégation

Syntaxe générale :

```
SELECT OP1(Ai),...,OPk(Aj)
FROM R
WHERE F;
```

Par exemple :

```
SELECT AVG(prix)
FROM Logiciel ;
```

Regroupements

On peut partionner une table suivant certains attributs :

```
SELECT A1, OP1(A2)
FROM R
WHERE F
GROUP BY P;
```

Regroupements

On peut partionner une table suivant certains attributs :

```
SELECT A1, OP1(A2)
FROM R
WHERE F
GROUP BY P;
```

Attention, on ne peut grouper que selon le critère suivant : **seuls** les attributs apparaissant dans P peuvent apparaître sans opérateur d'agrégation dans la clause **SELECT**.

Regroupements

On peut partionner une table suivant certains attributs :

```
SELECT A1, OP1(A2)
FROM R
WHERE F
GROUP BY P;
```

Attention, on ne peut grouper que selon le critère suivant : **seuls** les attributs apparaissant dans P peuvent apparaître sans opérateur d'agrégation dans la clause **SELECT**.

Par exemple :

```
SELECT type, COUNT(ref)
FROM Ordinateur
GROUP BY type;
```

Clause HAVING

On peut imposer un critère de sélection aux regroupements :

```
SELECT A1,...,Ap
FROM R
WHERE F
GROUP BY P
HAVING L;
```

Les mêmes conditions que celles portant sur **SELECT** dans le cas d'un **GROUP BY** s'appliquent.

Clause HAVING

On peut imposer un critère de sélection aux regroupements :

```
SELECT A1,...,Ap
FROM R
WHERE F
GROUP BY P
HAVING L;
```

Les mêmes conditions que celles portant sur **SELECT** dans le cas d'un **GROUP BY** s'appliquent.

Par exemple :

```
SELECT O.type AS "Type", COUNT(O.ref) AS "Nombre exemplaires"
FROM Ordinateur O
WHERE capa_mem >= 512
GROUP BY O.type
HAVING count(O.ref) >= 2
ORDER BY 2 ASC;
```

Sous-requêtes

On peut utiliser une requête dans une requête. On parle alors de **sous-requête**. Elle peut être de trois types :

- elle retourne une **relation** et apparaît dans **FROM**. Il faut utiliser un nom d'alias ;
- elle retourne une **constante** et apparaît dans **WHERE** ;
- elle retourne une **relation** et apparaît dans **WHERE**. Dans ce cas, on peut l'utiliser de différentes façons :
  - `SELECT... WHERE prix < 10000;`
  - `SELECT... WHERE prix < (SELECT prix FROM...);`
  - `SELECT... WHERE prix IN (SELECT prix FROM...);`
  - `SELECT... WHERE prix < ALL (SELECT prix FROM...);`
  - `SELECT... WHERE prix < ANY (SELECT prix FROM...);`
  - `SELECT... WHERE EXISTS (SELECT... WHERE prix=5);`

  Les opérateurs **IN** et **EXISTS** peuvent être précédés de **NOT**.

  L'opérateur **IN** peut servir pour représenter une jointure.

Sous-requêtes

```
SELECT provenance
FROM Systeme
WHERE nom_sys IN
    (SELECT nom_sys
     FROM LogicielBase
     WHERE type = 'Micral 75')
```

```
SELECT nom, classe
FROM Logiciel
WHERE NOT EXISTS
    (SELECT nom_log
     FROM Installation
     WHERE nom = nom_log);
```

Insertion

En extension :

```
INSERT
INTO R(Ai, Aj,..., Ap)
VALUES (vi, vj,..., vp);
```

Insertion

En extension :

```
INSERT
INTO R(Ai, Aj,..., Ap)
VALUES (vi, vj,..., vp);
```

En intension :

```
INSERT
INTO R(Ai, Aj,..., Ap)
SELECT Ci, Cj,..., Cp
FROM ...;
```

Modification

```
UPDATE R
SET Ai = vi,..., Ap = vp
WHERE F;
```

Modification

```
UPDATE R
SET Ai = vi,..., Ap = vp
WHERE F;
```

Par exemple :

```
UPDATE Ordinateur
SET capa_mem = capa_mem * 2
WHERE type = 'Micral 75';
```

Suppression de n-uplets

```
DELETE
FROM R
WHERE F;
```

Suppression de n-uplets

```
DELETE
FROM R
WHERE F;
```

Par exemple :

```
DELETE
FROM Ordinateur
WHERE type = 'Goupil G4';
```

Septième partie

SQL (*Structured Query Language*)

Structures manipulées

Le langage de description de données permet de créer et d'administrer :

- les tables ;
- les vues ;
- les index.

Structures manipulées

Le langage de description de données permet de créer et d'administrer :

- les tables ;
- les vues ;
- les index.

On dispose de types de base : **CHAR**(n), (n), **NUMBER**(n, d), **SMALLINT**, **INTEGER**, **FLOAT**, **DATE**, **TIME** et **TIMESTAMP**.

Création de table

On utilise l'opérateur **CREATE** :

```
CREATE TABLE Nom (
      ATT1 Type1,
      ATT2 Type2,
      ...
);
```

Création de table

On utilise l'opérateur **CREATE** :

```
CREATE TABLE Nom (
      ATT1 Type1,
      ATT2 Type2,
      ...
);
```

On peut insérer des lignes à la création :

```
CREATE TABLE Nom (nom_col1 Type1,...)
     AS SELECT col1... FROM R
     WHERE F;
```

Modification d'une table

Ajout d'un attribut à une relation :

```
ALTER TABLE R ADD(
    attribut Type [NULL/NOT NULL]
);
```

Modification d'une table

Ajout d'un attribut à une relation :

```
ALTER TABLE R ADD(
    attribut Type [NULL/NOT NULL]
);
```

Changer le type ou l'indétermination d'un attribut :

```
ALTER TABLE R MODIFY(
    attribut NouveauType [NULL/NOT NULL]
);
```

Modification d'une table

Ajout d'un attribut à une relation :

```
ALTER TABLE R ADD(
    attribut Type [NULL/NOT NULL]
);
```

Changer le type ou l'indétermination d'un attribut :

```
ALTER TABLE R MODIFY(
    attribut NouveauType [NULL/NOT NULL]
);
```

Détruire une colonne n'intervenant pas par ailleurs :

```
ALTER TABLE R DROP nom_col;
```

Destruction d'une relation

On utilise l'opération **DROP TABLE** Relation;

Destruction d'une relation

On utilise l'opération **`DROP TABLE`** `Relation;`

Attention, on supprime :

- le contenu de la relation ;
- le schéma associé à la relation.

Septième partie

SQL (*Structured Query Language*)

Gestion des droits d'accès

Comme dans un système de fichiers, les SGBD fondés sur SQL proposent un mécanisme de droit d'accès.

Un utilisateur d'une base de données est répertorié par le SGBD par :

- un identifiant interne ;
- un mot de passe ;
- un nom d'usage.

Le créateur d'une relation possède tous les droits sur cette relation.

Octroi des droits d'usage

Structure de base :

```
GRANT [ALL | Liste DML + ALTER]
ON table_1,..., table_n
TO [PUBLIC | Liste utilisateurs]
   [WITH GRANT OPTION];
```

`WITH GRANT OPTION` permet aux utilisateurs d'octroyer le droit à d'autres utilisateurs.

Retrait des droits d'usage

Structure de base :

```
REVOKE [ALL | Liste DML + ALTER]
ON table_1,..., table_n
TO [PUBLIC | Liste utilisateurs]
```

On peut également utiliser des vues externes.

Septième partie

SQL (*Structured Query Language*)

Motivation

Les informations stockées dans une base de données doivent respecter des **contraintes d'intégrité**.

Ces contraintes peuvent concerner la valeur d'un attribut.

Elles peuvent être plus complexes : action à réaliser lors du déclenchement d'un événement particulier par exemple.

Les contraintes de clés doivent pouvoir être représentées : clé **primaire**, clé **étrangère**.

Clé primaire

```
CREATE TABLE Table (
  ATT1  Type1,
  ...
  ATTi  Typei PRIMARY KEY,
  ...
);
```

Clé primaire

```
CREATE TABLE Table (
  ATT1  Type1,
  ...
  ATTi  Typei PRIMARY KEY,
  ...
);
```

```
CREATE TABLE Table (
  ATT1  Type1,
  ...
  ATTn  Typen,
  PRIMARY KEY (ATTi,...,ATTj)
);
```

Clés primaires

```
CREATE TABLE Configuration (
  nom       VARCHAR(20),
  nom_sys   VARCHAR(20),
  type      VARCHAR(20),
  mem_min   INTEGER DEFAULT 4,
  disk_min  INTEGER DEFAULT 1,
  PRIMARY KEY (nom, nom_sys, type)
);
```

Clés étrangères

```
CREATE TABLE Table1 (
ATT1 Type1,
...
ATTi Typei REFERENCES Table2(ATT2j),
...
);
```

Clés étrangères

```
CREATE TABLE Table1 (
ATT1 Type1,
...
ATTi Typei REFERENCES Table2(ATT2j),
...
);
```

```
CREATE TABLE T1 (
ATT1 Type1,
...
ATTn Typen,
FOREIGN KEY (ATTi,...,ATTj) REFERENCES
                            T2(ATT2l,..., ATT2k),
...
);
```

Clés étrangères

Hypothèse

Pour que l'attribut a_R de la relation R puisse apparaître comme clé étrangère de l'attribut a_T de la relation T, il faut et il suffit que :

- *a_R soit déclaré* **`UNIQUE`** *ou apparaissent dans la clé primaire de R ;*
- *les valeurs de a_T apparaissant dans des tuples de T doivent apparaître dans des tuples de R.*

Clés étrangères

```
CREATE TABLE Configuration (
  nom       VARCHAR(20),
  nom_sys   VARCHAR(20),
  type      VARCHAR(20),
  mem_min   INTEGER DEFAULT 4,
  disk_min  INTEGER DEFAULT 1,
  PRIMARY KEY (nom, nom_sys, type),
  FOREIGN KEY (nom) REFERENCES Logiciel(nom),
  FOREIGN KEY (nom_sys) REFERENCES Systeme(nom_sys),
  FOREIGN KEY (type) REFERENCES TypeOrdinateur(type)
);
```

Politiques de gestion des clés étrangères

Il existe plusieurs politiques de gestion des clés étrangères :

- politique par défaut : on n'autorise pas de modifications illicites (dans les deux sens) ;
- politique en cascade : on répercute les changements (que dans un sens, référence vers table « utilisatrice ») ;
- politique **`NULL`** : on met les champs de la table « utilisatrice » concernés à **`NULL`**.

Politiques de gestion des clés étrangères

Il existe plusieurs politiques de gestion des clés étrangères :

- politique par défaut : on n'autorise pas de modifications illicites (dans les deux sens) ;
- politique en cascade : on répercute les changements (que dans un sens, référence vers table « utilisatrice ») ;
- politique **NULL** : on met les champs de la table « utilisatrice » concernés à **NULL**.

```
CREATE TABLE T1 (
...
  FOREIGN KEY ATT1 REFERENCES T2(...)
  ON DELETE [SET NULL|CASCADE]
  ON UPDATE [SET NULL|CASCADE]
);
```

On peut utiliser également des transactions.

Contraintes sur un attribut

Contrainte **NOT NULL**

```
CREATE TABLE Table (
  ...
  ATT1 Type1 NOT NULL,
  ...
);
```

Contraintes sur un attribut

Contrainte **NOT NULL**

```
CREATE TABLE Table (
  ...
  ATT1 Type1 NOT NULL,
  ...
);
```

Contrainte **CHECK**

```
CREATE TABLE Table (
  ...
  ATTi Typei ...
       CHECK F,
  ...
);
```

Contrainte sur un tuple

```
CREATE TABLE Table (
  ...
  ATTi Typei ...
  ...

  CHECK F
);
```

où F est une expression logique dans laquelle un ou plusieurs attributs de **Table** apparaissent. Si un attribut d'une autre relation apparaît dans F, celui-ci doit provenir d'une clause **SELECT** incluse dans F.

Exemple de contraintes

```
CREATE TABLE Configuration (
  nom       VARCHAR(20),
  nom_sys   VARCHAR(20),
  type      VARCHAR(20),
  mem_min   INTEGER DEFAULT 4,
  disk_min  INTEGER DEFAULT 1,
  PRIMARY KEY (nom, nom_sys, type),
  FOREIGN KEY (nom) REFERENCES Logiciel(nom)
  DEFERRABLE INITIALLY DEFERRED,
  FOREIGN KEY (nom_sys) REFERENCES Systeme(nom_sys)
  DEFERRABLE INITIALLY IMMEDIATE,
  FOREIGN KEY (type) REFERENCES TypeOrdinateur(type),
  CHECK ((mem_min >= 0) AND (disk_min >= 0))
);
```

Vérification des contraintes

Les contraintes sur un attribut ou sur un tuple sont vérifiées :

- à la modification d'un tuple ou d'un attribut ;
- à l'insertion d'un tuple.

Vérification des contraintes

Les contraintes sur un attribut ou sur un tuple sont vérifiées :

- à la modification d'un tuple ou d'un attribut ;
- à l'insertion d'un tuple.

Que faire si ces contraintes concernent une autre relation par exemple ?

Triggers

```
CREATE TRIGGER NomTrigger
[AFTER|BEFORE] [UPDATE [OF att]|INSERT|DELETE] ON Table
REFERENCING
  [OLD ROW AS NomAncienTuple],
  [NEW ROW AS NomNouveauTuple],
  [OLD TABLE AS NomAncienneTable],
  [NEW TABLE AS NomNouvelleTable]
[FOR EACH STATEMENT|FOR EACH ROW]
WHEN (C)
  BEGIN [ATOMIC]
    Action1;
    ...
    ActionP;
  END;
```

Triggers : exemple

```
CREATE TRIGGER TriggerMoyenneLogiciel
        AFTER UPDATE OF prix ON Logiciel
        REFERENCING
                OLD TABLE AS AncienneTable,
                NEW TABLE AS NouvelleTable
        FOR EACH STATEMENT
        WHEN (10000 > (SELECT AVG(prix) FROM Logiciel))
        BEGIN
                DELETE FROM Logiciel
                WHERE (nom, classe, concep, provenance, revendeur, prix)
                IN NouvelleTable;
                INSERT INTO Logiciel
                (SELECT * FROM AncienneTable);
        END;
```

Septième partie

SQL (*Structured Query Language*)

PostgreSQL et respect du standard SQL

Quelques limitations :

- pas de choix des niveaux d'isolation des transactions ;
- pas d'assertions ;
- pas de requêtes dans les clauses **CHECK** (intentionnel) ;
- pas de type de données complexe : BLOB...
- une partie des fonctionnalités des types XML ;
- restrictions dans les *triggers* et syntaxe différente (cf. section suivante)
- certains langages ne sont pas disponibles : ADA, COBOL etc.

PostgreSQL 8.4 Documentation.
`http://www.postgresql.org/docs/8.4/interactive/`.

Fonctions utilisateur

On peut stocker des fonctions du côté serveur :

```
CREATE OR REPLACE FUNCTION bestStudent(id1 INTEGER, id2 INTEGER)
                  RETURNS INTEGER AS $BESTSTUDENT$
   DECLARE
      moy1 REAL;
      moy2 REAL;
   BEGIN
      moy1 := AVG(note) FROM Notes WHERE id = id1;
      moy2 := AVG(note) FROM Notes WHERE id = id2;

      IF (moy1 > moy2) THEN
         RETURN id1;
      ELSE
         RETURN id2;
      END IF;
   END;
$BESTSTUDENT$ language plpgsql;
```

Fonctions triggers

```
CREATE OR REPLACE FUNCTION trigger_moyenne() RETURNS trigger AS $trigger_moyenn
  DECLARE
    moy FLOAT;
  BEGIN
      moy := AVG(prix) FROM Logiciel;
      IF (moy > 10000) THEN
          RAISE EXCEPTION 'problem with average price';
      END IF;
      RETURN NULL;
  END;
$trigger_moyenne$ LANGUAGE plpgsql;
```

Déclaration d'un trigger

```
CREATE TRIGGER TriggerMoyenneLogiciel
AFTER UPDATE ON Logiciel
FOR EACH STATEMENT
EXECUTE PROCEDURE trigger_moyenne();
```

Huitième partie

Intégration de SQL dans un langage de programmation

Huitième partie

Intégration de SQL dans un langage de programmation

Introduction

Utilisation de SQL interactive jusqu'à présent. Mais :

- besoin de structures de contrôle complexes
- besoin de traitement complexes des données
- besoin de modélisation et de programmation de *processes* métiers
- besoin de construction d'IHM
- ...

Introduction

Utilisation de SQL interactive jusqu'à présent. Mais :

- besoin de structures de contrôle complexes
- besoin de traitement complexes des données
- besoin de modélisation et de programmation de *processes* métiers
- besoin de construction d'IHM
- ...

Une première solution : les PSM (*Persistent Storage Modules*)

Introduction

Utilisation de SQL interactive jusqu'à présent. Mais :

- besoin de structures de contrôle complexes
- besoin de traitement complexes des données
- besoin de modélisation et de programmation de *processes* métiers
- besoin de construction d'IHM
- ...

Une première solution : les PSM (*Persistent Storage Modules*)

Mais cela ne suffit pas : définitions d'interfaces entre SQL et COBOL, C, C++, Ada etc.

Problèmes posés

Problème

Structures de données SQL $\neq$ structures de données langage hôte

Problèmes posés

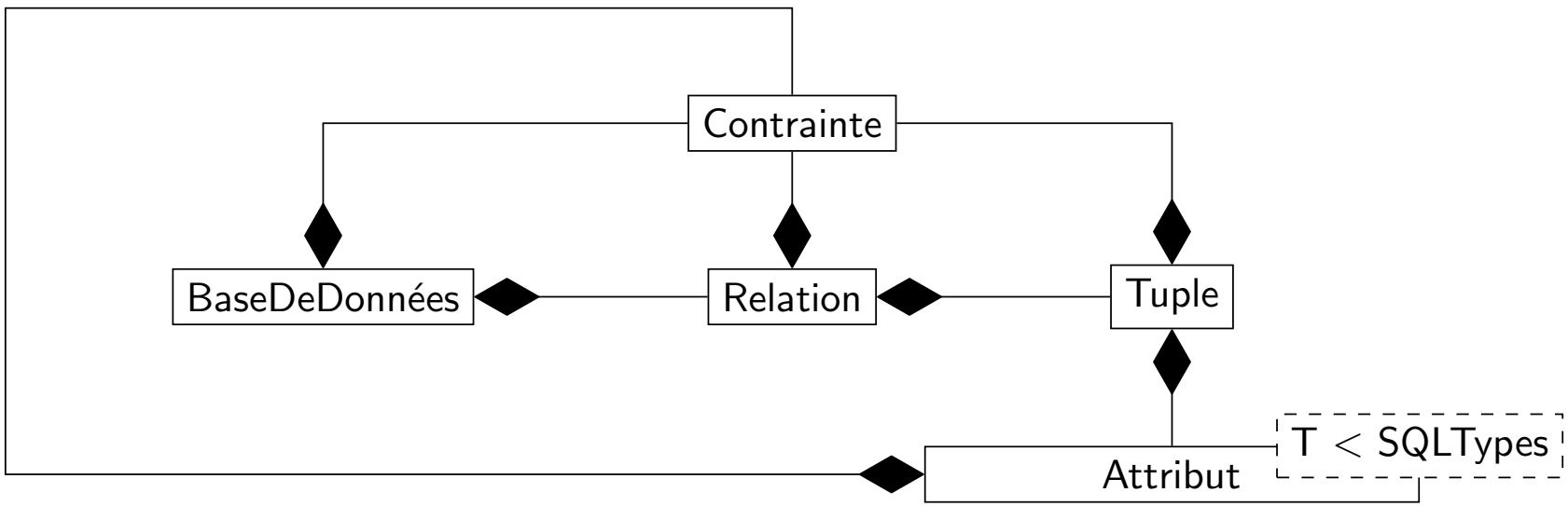
Problème

Structures de données SQL $\neq$ structures de données langage hôte

En particulier, 2 problèmes :

- *impedance mismatch*
- représentation dans le langage d'une base de données

Exemple : « méta-modélisation » UML



Intégration de SQL dans un langage hôte

Deux axes avec chacun deux solutions :

- statiquement ou dynamiquement
  - statique : utilisation d'un compilateur ou d'un préprocesseur
  - dynamique : instructions SQL construites et exécutées lors de l'exécution du programme hôte

Intégration de SQL dans un langage hôte

Deux axes avec chacun deux solutions :

- statiquement ou dynamiquement
  - statique : utilisation d'un compilateur ou d'un préprocesseur
  - dynamique : instructions SQL construites et exécutées lors de l'exécution du programme hôte
- intégré ou appelé
  - intégré : instructions SQL dans le code source
  - appelé : instructions SQL encapsulées dans des appels

Huitième partie

Intégration de SQL dans un langage de programmation

Exemple simple

- SQL statique et intégré
- utilisation de ecpg, fourni avec PostgreSQL

Exemple simple

- SQL statique et intégré
- utilisation de ecpg, fourni avec PostgreSQL

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Trying to connect to database\n");
5      EXEC SQL CONNECT TO 'bdpolyIN306@serv-sun1.isae.fr' USER garion;
6      printf("Connected...\n");
7
8      printf("Trying to disconnect\n");
9      EXEC SQL DISCONNECT ALL;
10     printf("Disconnected...\n");
11
12     return 0;
13 }
```

Code C obtenu

```
1  /* Processed by ecpg (4.2.1) */
2  /* These include files are added by the preprocessor */
3  #include <ecpgtype.h>
4  #include <ecpglib.h>
5  #include <ecpgerrno.h>
6  #include <sqlca.h>
7  /* End of automatic include section */
8
9  #line 1 "connexionC.pgc"
10 #include <stdio.h>
11
12 int main() {
13     printf("Trying to connect to database\n");
14     { ECPGconnect(__LINE__, 0, "bdpolyIN306@serv-sun1.isae.fr" , "garion" , NULL
15 #line 5 "connexionC.pgc"
16
17     printf("Connected...\n");
18
19     printf("Trying to disconnect\n");
20     { ECPGdisconnect(__LINE__, "ALL");}
21 #line 9 "connexionC.pgc"
```

Partage de variables et requêtes

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Trying to connect to database\n");
5      EXEC SQL CONNECT TO 'bdpolyIN306@serv-sun1.isae.fr' USER garion;
6      printf("Connected...\n");
7
8      EXEC SQL BEGIN DECLARE SECTION;
9        int prix;
10     EXEC SQL END DECLARE SECTION;
11
12     EXEC SQL SELECT prix INTO :prix FROM logiciel WHERE nom = 'Oracle';
13
14     printf("Result: %d\n", prix);
15
16     printf("Trying to disconnect\n");
17     EXEC SQL DISCONNECT ALL;
18     printf("Disconnected...\n");
19
20     return 0;
21 }
```

Requête avec cardinalité de la réponse > 1

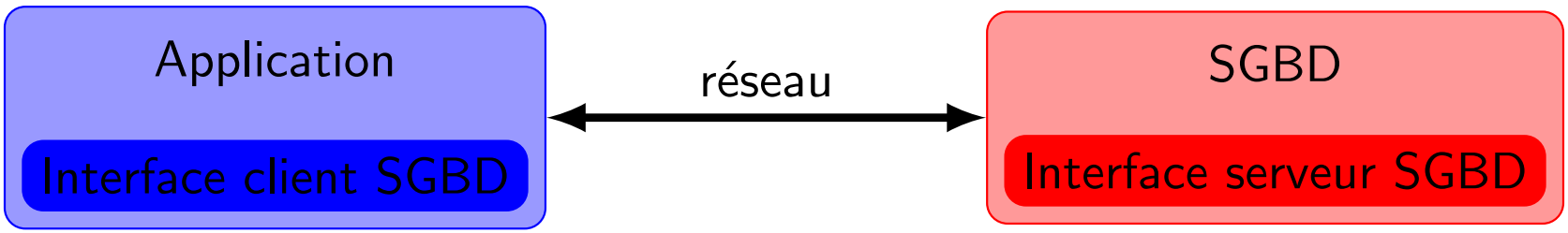
```
 1  int main() {
 2      printf("Trying to connect to database\n");
 3      EXEC SQL CONNECT TO 'bdpolyIN306@serv-sun1.isae.fr' USER garion;
 4      printf("Connected...\n");
 5
 6      EXEC SQL BEGIN DECLARE SECTION;
 7        char nom[20];
 8        int prix;
 9      EXEC SQL END DECLARE SECTION;
10
11      EXEC SQL DECLARE curseur CURSOR FOR SELECT nom, prix FROM logiciel;
12
13      EXEC SQL OPEN curseur;
14
15      while (sqlca.sqlcode == 0) {
16         EXEC SQL FETCH NEXT FROM curseur INTO :nom, :prix;
17
18         if (sqlca.sqlcode == 0) {
19            printf("Software: %s, price: %d\n", nom, prix);
20         }
21      }
22
23      EXEC SQL CLOSE curseur;
```

Huitième partie

Intégration de SQL dans un langage de programmation

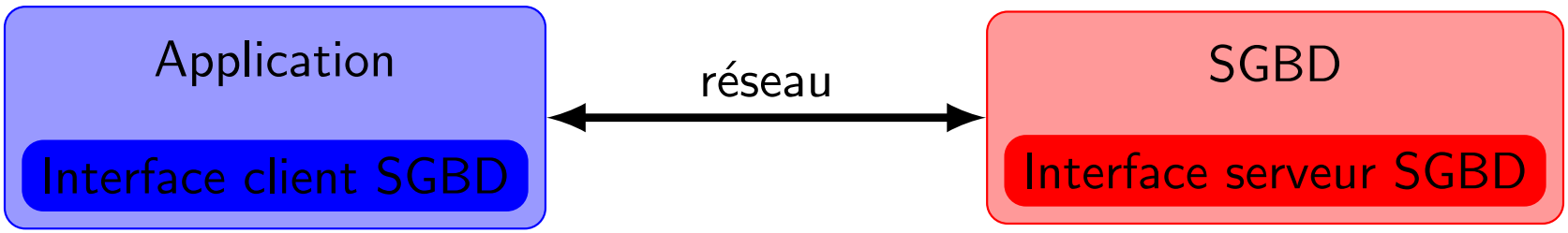
Mode client/serveur

Paradigme classique en informatique : serveur web, mail, serveur X...



Mode client/serveur

Paradigme classique en informatique : serveur web, mail, serveur X...

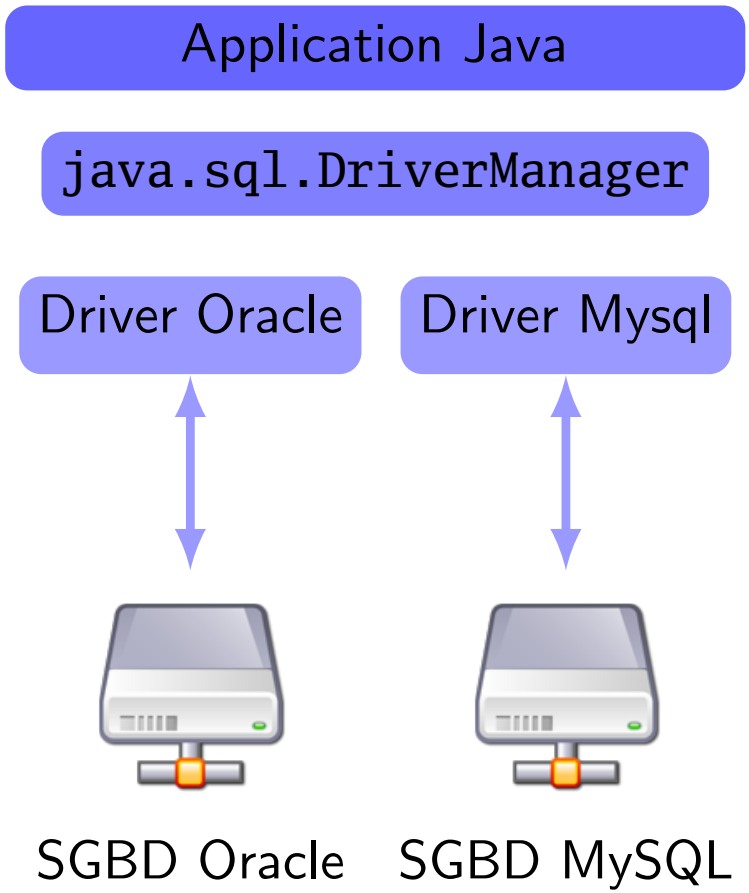


- jusqu'à récemment, les applications client/serveur étaient réalisées par le même constructeur ;
- puis apparition de systèmes à base de *drivers* : ODBC (Microsoft) et JDBC (*Java DataBase Connectivity*).

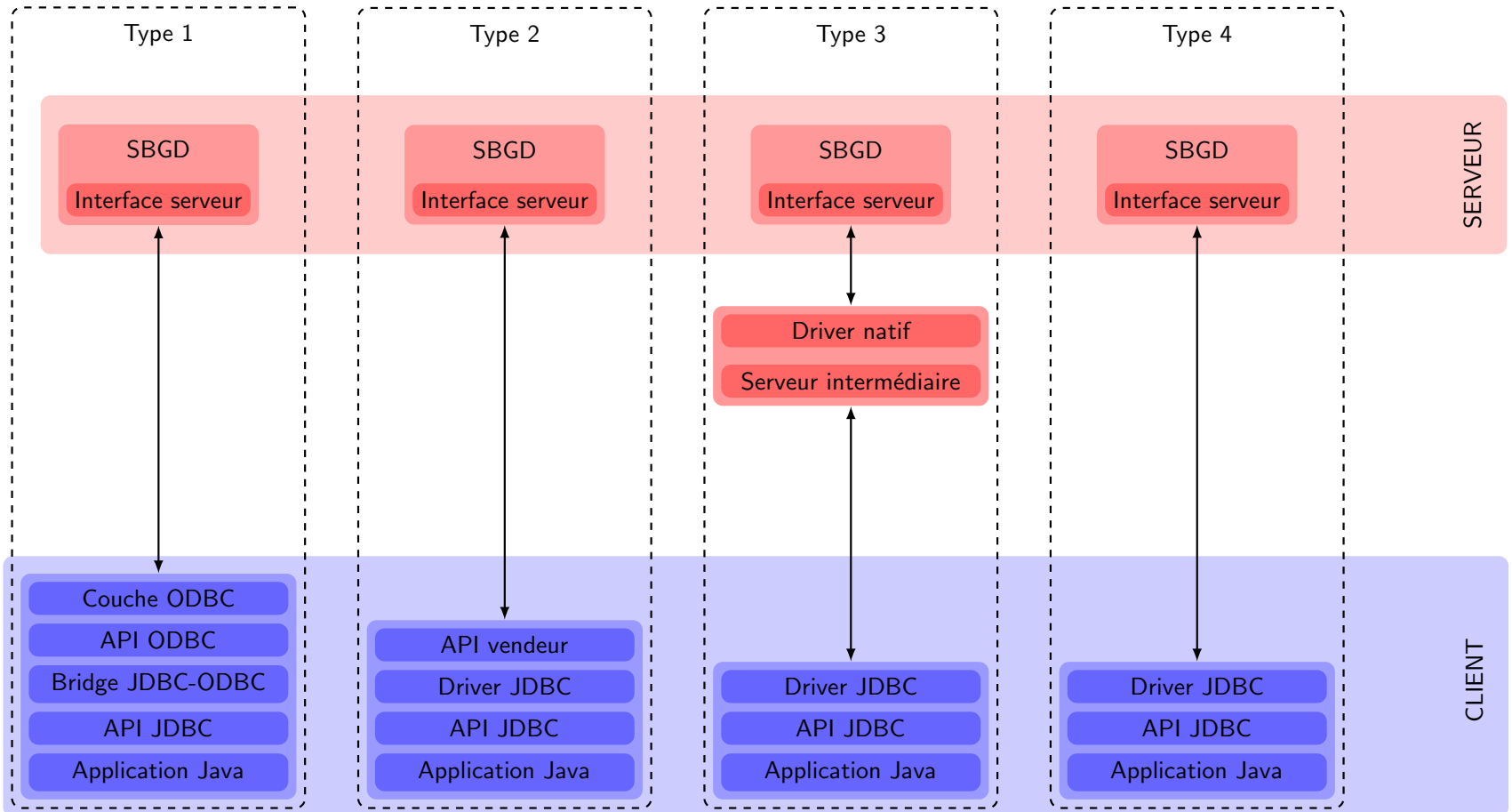
Pourquoi JDBC ?

- ODBC est une couche d'abstraction universelle pour les bases de données, mais pas pour la plateforme (MS-Windows) ;
- JDBC est une API de Java, langage connu ;
- JDBC profite des avantages de Java :
  - indépendance vis-à-vis de la plateforme ;
  - code facile à écrire ;
  - nombreuses bibliothèques disponibles ;
  - interaction avec les *applets*, les *servlets*, une architecture J2EE etc.

Structure d'une application JDBC



Type de drivers JDBC



Huitième partie

Intégration de SQL dans un langage de programmation

Un exemple

```
 1  import java.sql.*;
 2
 3  /**
 4   * Une connexion sur une base PostgreSQL.
 5   *
 6   * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 7   * @version 1.0
 8   */
 9  public class Connexion {
10
11      public static void main(String[] args) {
12          try {
13              System.out.println("Chargement du driver et " +
14                                 "ouverture de la connexion");
15              Class.forName("org.postgresql.Driver");
16
17              String url = "jdbc:postgresql://serv-sun1.isae.fr/bdpolyIN306";
18              String user = "garion";
19              Connection connection = DriverManager.getConnection(url,
20                                                                  user,
21                                                                  "");
```

Un exemple

```
1                // on demande une requete
2                Statement statement = connection.createStatement();
3                ResultSet res = statement.executeQuery("SELECT * " +
4                                                       "FROM Ordinateur");
5
6                // renseignements sur le ResultSet
7                ResultSetMetaData m = res.getMetaData();
8                for (int i = 1; i <= m.getColumnCount(); i++) {
9                    System.out.println(m.getColumnName(i) + " de type " +
10                                      m.getColumnType(i));
11               } // end of for (int i = 1; i <= m.getColumnCount(); i++)
12               System.out.println();
13
14               // on parcourt le ResultSet
15               while (res.next()) {
16                   System.out.println(res.getString("type") + " " +
17                                      res.getFloat("prix"));
18               } // end of while (res.next())
```

Un exemple

```
1               // on ferme proprement
2               res.close();
3               statement.close();
4               connection.close();
5           } catch (ClassNotFoundException e) {
6               e.printStackTrace();
7           } // end of catch
8           catch (SQLException e) {
9               e.printStackTrace();
10          } // end of try-catch
11
12      } // end of main()
13  }
```

Créer et exécuter une requête

Création d'une requête :

- `java.sql.Statement`
- `java.sql.PreparedStatement`
- `java.sql.CallableStatement`

Créer et exécuter une requête

Création d'une requête :

- `java.sql.Statement`
- `java.sql.PreparedStatement`
- `java.sql.CallableStatement`

Exécution d'une requête :

- `public java.sql.ResultSet executeQuery(String query)`
- `public int executeUpdate(String update)`
- `public boolean execute(String query)`

Traitement de la requête

`java.sql.ResultSet` :

- `public String getString(String nomColonne);`
- `public float getFloat(String nomColonne);`
- `public float getFloat(int numColonne).`

Traitement de la requête

`java.sql.ResultSet` :

- `public String getString(String nomColonne);`
- `public float getFloat(String nomColonne);`
- `public float getFloat(int numColonne).`

`java.sql.ResultSetMetaData` :

- `getColumnCount()`
- `getColumnName(int i)`
- `getColumnType(int i)`

Conclusion

Approche statique :

- gestion des variables partagées
- précompilation
- lisibilité
- disponible pour Java dans certains SGBD

Conclusion

Approche statique :

- gestion des variables partagées
- précompilation
- lisibilité
- disponible pour Java dans certains SGBD

Approche dynamique :

- pas de précompilation
- écriture plus lourde
- transformation statique $\rightarrow$ dynamique

Conclusion

Approche statique :

- gestion des variables partagées
- précompilation
- lisibilité
- disponible pour Java dans certains SGBD

Approche dynamique :

- pas de précompilation
- écriture plus lourde
- transformation statique $\rightarrow$ dynamique

Intérêt des requêtes précompilées.

Neuvième partie

Gestion des transactions

28 **Notion de transaction**

29 **Reprise après panne**

30 **Gestion de la concurrence**
- Quelques problèmes
- Séquentialité
- Verrouillage
- Blocage
- Verrouillage intentionnel

31 **Bases de données distribuées**

32 **Fonctionnalités fournies par SQL**

Neuvième partie

Gestion des transactions

Définition

Pour l'instant, les opérations sur la base de données sont considérées comme **atomiques**.

Définition

Pour l'instant, les opérations sur la base de données sont considérées comme **atomiques**.

Que se passe-t-il si on considère deux comptes C1 et C2 et les opérations suivantes :

```
UPDATE Compte SET solde = solde - 100 WHERE no_compte = 'C1';
UPDATE Compte SET solde = solde + 100 WHERE no_compte = 'C2';
```

Définition

Pour l'instant, les opérations sur la base de données sont considérées comme **atomiques**.

Que se passe-t-il si on considère deux comptes C1 et C2 et les opérations suivantes :

```
UPDATE Compte SET solde = solde - 100 WHERE no_compte = 'C1';
UPDATE Compte SET solde = solde + 100 WHERE no_compte = 'C2';
```

Ces deux opérations élémentaires forment un tout : on dit qu'elles forment une **transaction**.

Définition

Pour l'instant, les opérations sur la base de données sont considérées comme **atomiques**.

Que se passe-t-il si on considère deux comptes C1 et C2 et les opérations suivantes :

```
UPDATE Compte SET solde = solde - 100 WHERE no_compte = 'C1';
UPDATE Compte SET solde = solde + 100 WHERE no_compte = 'C2';
```

Ces deux opérations élémentaires forment un tout : on dit qu'elles forment une **transaction**.

Définition (transaction)

Une transaction est une unité logique de travail. C'est une séquence d'opérations élémentaires qui est vue comme atomique d'un point de vue externe au SGBD.

Les propriétés ACID

Proposition (ACID)

Des transactions ayant un « bon » comportement respectent les propriétés suivantes, dites propriétés ACID :

- *« A » pour* **atomicité** *: la transaction doit s'effectuer dans sa totalité ou ne pas s'effectuer du tout ;*
- *« C » pour* **cohérence** *: une transaction préserve la cohérence de la base de données. Elle amène la base d'un état cohérent vers un autre état cohérent.*
- *« I » pour* **isolation** *: une transaction doit s'effectuer comme si aucune autre transaction ne s'effectuait en même temps.*
- *« D » pour* **durabilité** *: les effets d'une transaction sur la base de données ne doivent pas être perdus une fois que la transaction s'est effectuée.*

Les propriétés ACID

Proposition (ACID)

Des transactions ayant un « bon » comportement respectent les propriétés suivantes, dites propriétés ACID :

- *« A » pour* **atomicité** *: la transaction doit s'effectuer dans sa totalité ou ne pas s'effectuer du tout ;*
- *« C » pour* **cohérence** *: une transaction préserve la cohérence de la base de données. Elle amène la base d'un état cohérent vers un autre état cohérent.*
- *« I » pour* **isolation** *: une transaction doit s'effectuer comme si aucune autre transaction ne s'effectuait en même temps.*
- *« D » pour* **durabilité** *: les effets d'une transaction sur la base de données ne doivent pas être perdus une fois que la transaction s'est effectuée.*

Question

Comment garantir ces propriétés ?

Opérations de base du gestionnaire de transactions

Deux opérations de base :

- **COMMIT** qui signale la fin réussie d'une transaction ;
- **ROLLBACK** qui signale l'abandon d'une transaction.

```
BEGIN TRANSACTION
  UPDATE Compte SET solde = solde - 100 WHERE no_compte = 'C1';
  IF  erreur_quelconque THEN GOTO UNDO;
  UPDATE Compte SET solde = solde + 100 WHERE no_compte = 'C2';
  IF  erreur_quelconque THEN GOTO UNDO;
  COMMIT;
  GOTO FINISH;

UNDO:
  ROLLBACK;

FINISH:
END TRANSACTION
```

Neuvième partie

Gestion des transactions

Durabilité des transactions

Comment garantir la **durabilité** des transactions ?

Durabilité des transactions

Comment garantir la **durabilité** des transactions ?

Principe

Il faut s'assurer que les changements effectués par une transaction sont écrits sur le disque.

Durabilité des transactions

Comment garantir la **durabilité** des transactions ?

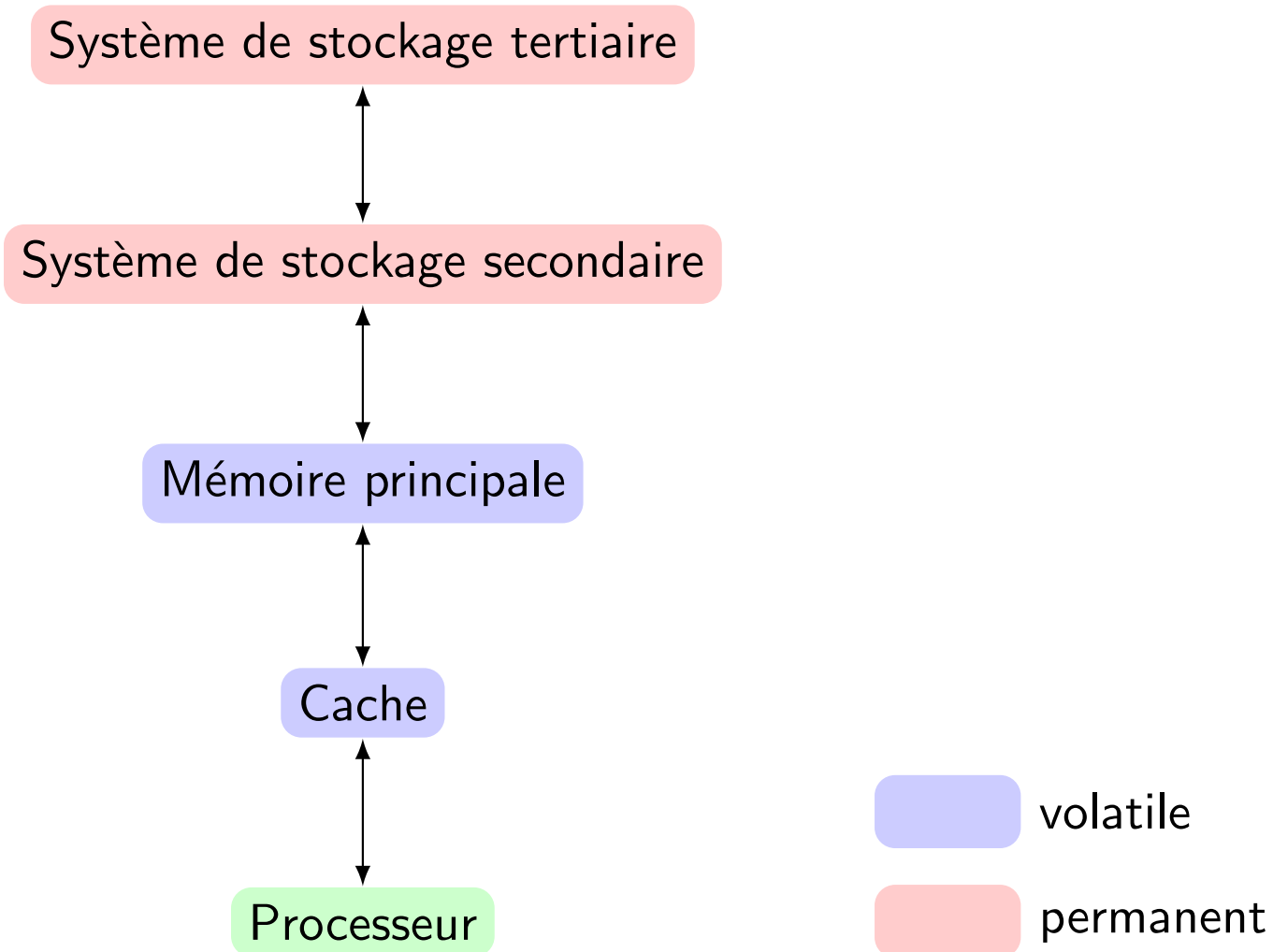
Principe

Il faut s'assurer que les changements effectués par une transaction sont écrits sur le disque.

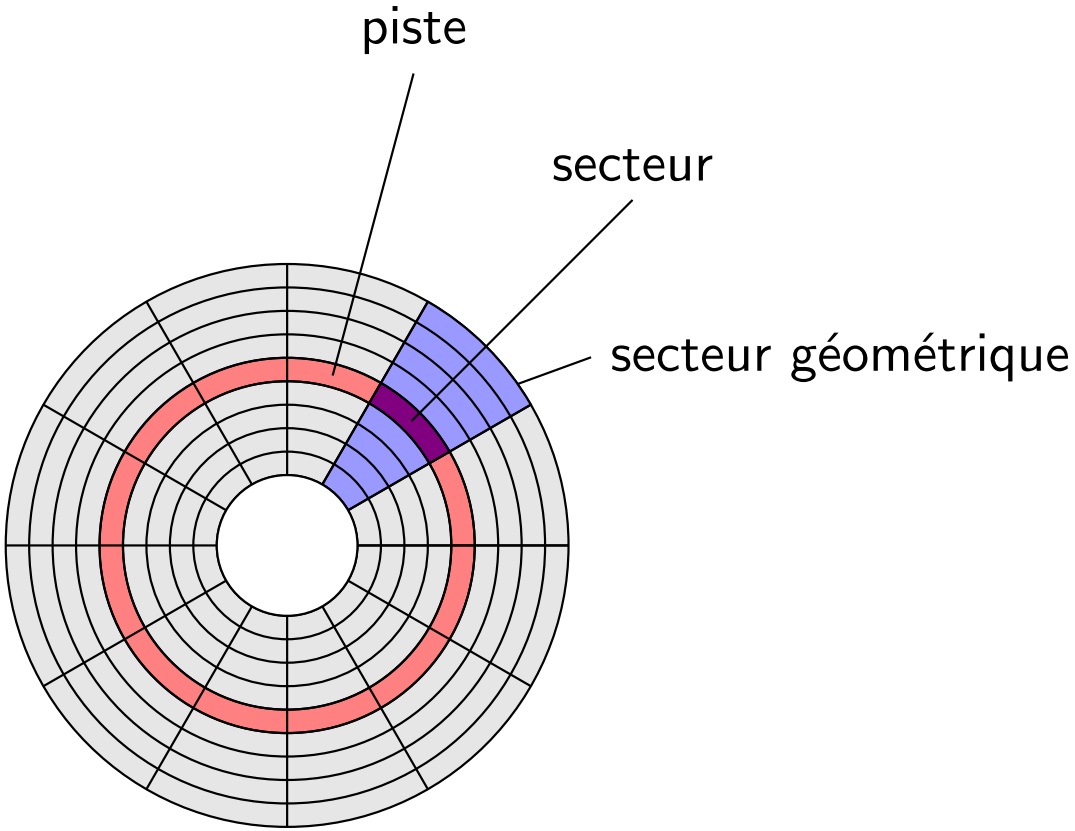
Problème

Écrit-on chaque changement sur le disque dur ?

Retour sur la structure mémoire d'une machine

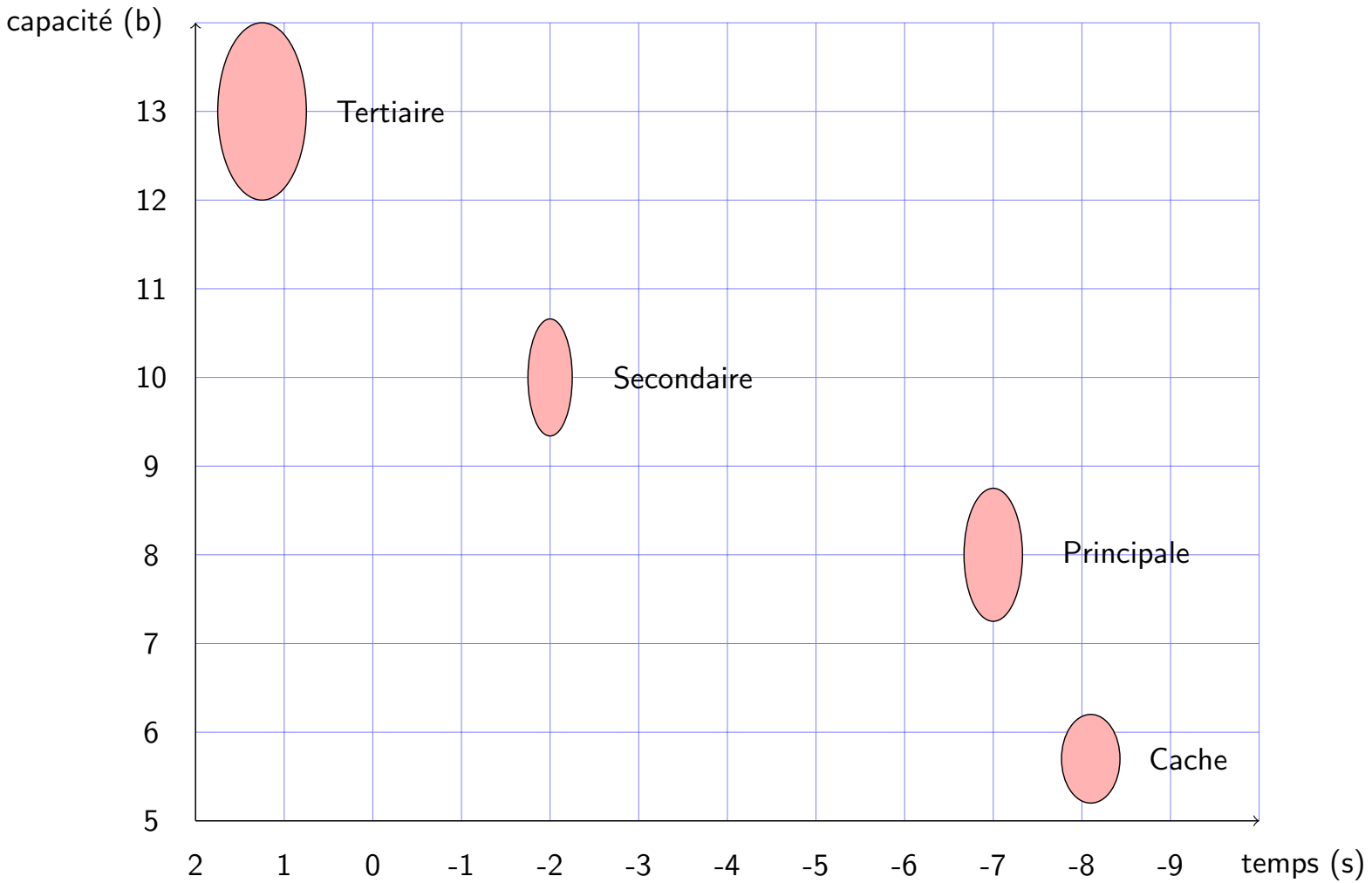


Structure d'un disque dur



- pour la plupart des disques, un secteur = 512 octets
- pour la plupart des OS, taille du bloc de lecture/écriture = 4 Ko

Temps d'accès des différents types de mémoires



Présentation de la reprise après panne

Problème : on travaille habituellement en mémoire centrale.

- on ne peut pas faire une écriture à chaque opération, cela pénaliserait les performances.
- lorsque le système est redémarré, les données en mémoire centrale disparaissent.

Présentation de la reprise après panne

Problème : on travaille habituellement en mémoire centrale.

- on ne peut pas faire une écriture à chaque opération, cela pénaliserait les performances.
- lorsque le système est redémarré, les données en mémoire centrale disparaissent.

Il faut savoir quand les modifications sont réellement inscrites.

Présentation de la reprise après panne

Problème : on travaille habituellement en mémoire centrale.

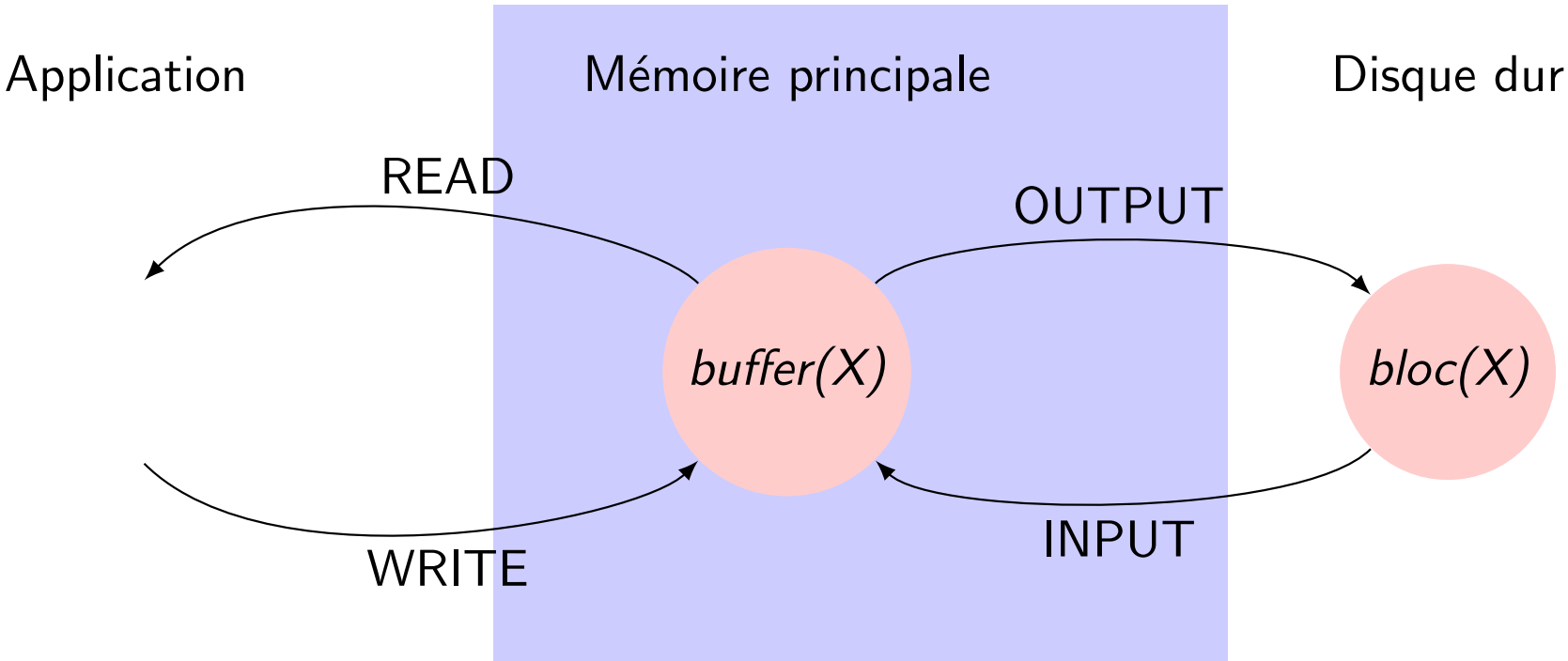
- on ne peut pas faire une écriture à chaque opération, cela pénaliserait les performances.
- lorsque le système est redémarré, les données en mémoire centrale disparaissent.

Il faut savoir quand les modifications sont réellement inscrites.

Conclusion

Le problème de la durabilité est donc lié à celui de la reprise après panne.

Opérations de base du gestionnaire de transactions



Opérations de base : exemple

Etape	Action	t	MemC1	MemC2	DC1	DC2
0		0	1000	500	1000	500
1	`t := READ(C1)`	1000	1000	500	1000	500
2	`t := t - 100`	900	1000	500	1000	500
3	`WRITE(C1,t)`	900	900	500	1000	500
4	`t := READ(C2)`	500	900	500	1000	500
5	`t := t + 100`	600	900	500	1000	500
6	`WRITE(C2,t)`	600	900	600	1000	500
7	`OUTPUT(C1)`	600	900	600	900	500
8	`OUTPUT(C2)`	600	900	600	900	600

Journal

On utilise un journal qui a différentes entrées :

- <START T> : la transaction T a démarré ;
- <**COMMIT** T> : la transaction T a réussi. Tous les changements effectués par T **doivent être inscrits sur le disque**.
- <ABORT T> : la transaction n'a pas réussi. Dans ce cas, aucun de ses changements ne doit apparaître sur le disque.
- <T, X, v> : la transaction T a changé la valeur de l'élément X et l'ancienne valeur ou la nouvelle valeur de X était ou est v. Ce changement concerne le changement en mémoire centrale et non pas sur le disque.

Undo-logging

Idée intuitive : effacer les changements qui ont été faits par une transaction qui ne s'est pas déroulée correctement (interrompue par une panne ou par un **ROLLBACK**).

Undo-logging

Idée intuitive : effacer les changements qui ont été faits par une transaction qui ne s'est pas déroulée correctement (interrompue par une panne ou par un **ROLLBACK**).

1. si une transaction `T` modifie un élément `X`, alors on doit écrire `<T, X, v>` où **v est l'ancienne valeur de X** dans le journal (donc sur le disque) **avant** que la nouvelle valeur de `X` ne soit écrite sur le disque.
2. si une transaction `T` est réussie, alors on écrit `<COMMIT T>` dès que tous les changements effectués par la transaction ont été écrits sur le disque.

Undo-logging : exemple

Etape	Action	t	MemC1	MemC2	DC1	DC2	Journal
0		0	1000	500	1000	500	<START T>
1	t := **READ**(C1)	1000	1000	500	1000	500	
2	t := t - 100	900	1000	500	1000	500	
3	**WRITE**(C1,t)	900	900	500	1000	500	<T,C1,1000>
4	t := **READ**(C2)	500	900	500	1000	500	
5	t := t + 100	600	900	500	1000	500	
6	**WRITE**(C2,t)	600	900	600	1000	500	<T,C2,500>
7	FLUSH LOG	600	900	600	1000	500	
8	**OUTPUT**(C1)	600	900	600	900	500	
9	**OUTPUT**(C2)	600	900	600	900	600	
10		600	900	600	900	600	<**COMMIT** T>
11	FLUSH LOG	600	900	600	900	600	

Reprise après panne avec undo-logging

Comment redémarrer un système proprement avec l'*undo-logging* ?

Reprise après panne avec undo-logging

Comment redémarrer un système proprement avec l'*undo-logging* ?

- on examine le journal en entier en partant de la fin du journal ;
- on trouve une entrée du type <**COMMIT** T>, alors on est sûr que les données modifiées par la transaction T ont été inscrites sur le disque ;
- on rencontre une entrée <T, X, v> et que nous n'avons pas rencontré d'entrée <**COMMIT** T>.
  - ➥ T est **incomplète** et on doit la **déjouer**
  - ➥ on utilise les entrées <T, X, v>
- le même principe s'applique pour une entrée <ABORT T>.

Notion de point de contrôle

On essaye d'éviter de parcourir tout le journal (trop coûteux).

Notion de point de contrôle

On essaye d'éviter de parcourir tout le journal (trop coûteux).

Principe

On force l'écriture de temps en temps.

Notion de point de contrôle

On essaye d'éviter de parcourir tout le journal (trop coûteux).

Principe

On force l'écriture de temps en temps.

1. écrire une entrée <START CHKPT(T1, ..., Tn)> dans le journal. T1, ..., Tn sont toutes les transactions actives au moment de l'écriture.
2. attendre que T1, ..., Tn se finissent. Continuer à accepter des transactions.
3. quand T1, ..., Tn ont fini, écrire <**END** CHKPT> dans le journal.

Notion de point de contrôle

On essaye d'éviter de parcourir tout le journal (trop coûteux).

Principe

On force l'écriture de temps en temps.

1. écrire une entrée <START CHKPT(T1, ..., Tn)> dans le journal. T1, ..., Tn sont toutes les transactions actives au moment de l'écriture.
2. attendre que T1, ..., Tn se finissent. Continuer à accepter des transactions.
3. quand T1, ..., Tn ont fini, écrire <**END** CHKPT> dans le journal.

Pour la reprise :

- soit on rencontre d'abord une entrée <**END** CHKPT>
  ➥les entrées précédant le début du checkpoint sont inutiles.
- soit on rencontre d'abord une entrée <START CHKPT(T1, ..., Tn)
  ➥on remonte jusqu'à la transaction Ti la plus ancienne.

Journalisation par redo

Principe

1. *si une transaction T modifie un élément X, alors on doit écrire <T, X, v> dans le journal sur le disque* **avant** *que la nouvelle valeur de X ne soit écrite sur le disque. Cette entrée signifie : X a été modifié en mémoire principale et la nouvelle valeur de X est v.*
2. *si une transaction T est réussie, alors on écrit <**COMMIT** T>* **avant** *de commencer à écrire les changements effectués par la transaction sur le disque.*

Journalisation par redo : exemple

Etape	Action	t	MemC1	MemC2	DC1	DC2	Journal
0		0	1000	500	1000	500	<START T>
1	t := **READ**(C1)	1000	1000	500	1000	500	
2	t := t - 100	900	1000	500	1000	500	
3	**WRITE**(C1,t)	900	900	500	1000	500	<T,C1,900>
4	t := **READ**(C2)	500	900	500	1000	500	
5	t := t + 100	600	900	500	1000	500	
6	**WRITE**(C2,t)	600	900	600	1000	500	<T,C2,600>
7		600	900	600	1000	500	<**COMMIT** T>
8	FLUSH LOG	600	900	600	1000	500	
9	**OUTPUT**(C1)	600	900	600	900	500	
10	**OUTPUT**(C2)	600	900	600	900	600	

Journalisation par undo/redo

Ces deux systèmes ont toutefois des défauts :

- *undo* : augmentation du nombre d'entrées/sorties ;
- *redo* : beaucoup d'informations en mémoire centrale.

Journalisation par undo/redo

Ces deux systèmes ont toutefois des défauts :

- *undo* : augmentation du nombre d'entrées/sorties ;
- *redo* : beaucoup d'informations en mémoire centrale.

On peut combiner les deux techniques :

Principe

Dans une procédure de journalisation par undo/redo, **avant** *de modifier un élément X de la base de données sur le disque, il est nécessaire d'écrire une entrée <T, X, v, w> sur le disque.*

Journalisation par undo/redo : exemple

Etape	Action	t	MemC1	MemC2	DC1	DC2	Journal
0		0	1000	500	1000	500	<START T>
1	t := **READ**(C1)	1000	1000	500	1000	500	
2	t := t - 100	900	1000	500	1000	500	
3	**WRITE**(C1,t)	900	900	500	1000	500	<T,C1,1000,900>
4	t := **READ**(C2)	500	900	500	1000	500	
5	t := t + 100	600	900	500	1000	500	
6	**WRITE**(C2,t)	600	900	600	1000	500	<T,C2,500,600>
8	FLUSH LOG	600	900	600	1000	500	
9	**OUTPUT**(C1)	600	900	600	900	500	
10		600	900	600	900	500	<**COMMIT** T>
11	**OUTPUT**(C2)	600	900	600	900	600	

Reprise avec undo/redo

La politique est la suivante :

1. commencer avec deux listes de transactions, la liste UNDO et la liste REDO. Initialiser la liste UNDO avec la liste de toutes les transactions enregistrées dans le compte rendu du point de contrôle le plus récent. Initialiser la liste REDO avec l'ensemble vide.
2. faire une recherche en avant dans le journal, en partant du point de contrôle.
3. si une entrée dans le journal correspondant à un `BEGIN` **`TRANSACTION`** est rencontrée pour la transaction `T`, ajouter `T` à la liste UNDO.
4. si une entrée dans le journal correspondant à un **`COMMIT`** est rencontrée pour la transaction `T`, déplacer `T` de la liste UNDO vers la liste REDO.

Reprise avec undo/redo

La politique est la suivante :

1. commencer avec deux listes de transactions, la liste UNDO et la liste REDO. Initialiser la liste UNDO avec la liste de toutes les transactions enregistrées dans le compte rendu du point de contrôle le plus récent. Initialiser la liste REDO avec l'ensemble vide.
2. faire une recherche en avant dans le journal, en partant du point de contrôle.
3. si une entrée dans le journal correspondant à un `BEGIN` **`TRANSACTION`** est rencontrée pour la transaction `T`, ajouter `T` à la liste UNDO.
4. si une entrée dans le journal correspondant à un **`COMMIT`** est rencontrée pour la transaction `T`, déplacer `T` de la liste UNDO vers la liste REDO.

Reprise en arrière : on annule les transactions de la liste UNDO.
Reprise en avant : on rejoue les transactions de la liste REDO.

undo/redo : exemple

<START T1>
<T1, A, 4, 5>
<START T2>
<**COMMIT** T1>
<T2, B, 9, 10>
<START CHKPT(T2)>
<T2, C, 14, 15>
<START T3>
<T3, D, 19, 20>
<**END** CHKPT>
<**COMMIT** T2>
<**COMMIT** T3>

$\text{UNDO} = \{\}$
$\text{REDO} = \{\}$

undo/redo : exemple

<START T1>
<T1, A, 4, 5>
<START T2>
<**COMMIT** T1>
<T2, B, 9, 10>
<START CHKPT(T2)>
<T2, C, 14, 15>
<START T3>
<T3, D, 19, 20>
<**END** CHKPT>
<**COMMIT** T2>
<**COMMIT** T3>

$\text{UNDO} = \{\text{T2}\}$
$\text{REDO} = \{\}$

undo/redo : exemple

<START T1>
<T1, A, 4, 5>
<START T2>
<**COMMIT** T1>
<T2, B, 9, 10>
<START CHKPT(T2)>
<T2, C, 14, 15>
<START T3>
<T3, D, 19, 20>
<**END** CHKPT>
<**COMMIT** T2>
<**COMMIT** T3>

UNDO = {T2,T3}
REDO = {}

undo/redo : exemple

<START T1>
<T1, A, 4, 5>
<START T2>
<**COMMIT** T1>
<T2, B, 9, 10>
<START CHKPT(T2)>
<T2, C, 14, 15>
<START T3>
<T3, D, 19, 20>
<**END** CHKPT>
<**COMMIT** T2>
<**COMMIT** T3>

UNDO = {T3}
REDO = {T2}

undo/redo : exemple

<START T1>
<T1, A, 4, 5>
<START T2>
<**COMMIT** T1>
<T2, B, 9, 10>
<START CHKPT(T2)>
<T2, C, 14, 15>
<START T3>
<T3, D, 19, 20>
<**END** CHKPT>
<**COMMIT** T2>
<**COMMIT** T3>

UNDO = {}
REDO = {T2, T3}

Neuvième partie

Gestion des transactions

Perte de mise à jour

Transaction T1	Temps	Transaction T2
-	-	-
RETRIEVE(A)	t_1	-
-	t_2	RETRIEVE(A)
UPDATE(A)	t_3	-
-	t_4	**UPDATE**(A)

Perte de mise à jour

Transaction T1	Temps	Transaction T2
-	-	-
RETRIEVE(A)	t_1	-
-	t_2	RETRIEVE(A)
UPDATE(A)	t_3	-
-	t_4	**UPDATE**(A)

Problème

La mise à jour effectuée par T1 est perdue à la date t_4.

Dépendances non validées

Transaction T1	Temps	Transaction T2
-	-	-
-	t_1	**UPDATE**(A)
RETRIEVE(A) ou **UPDATE**(A)	t_2	-
-	t_3	**ROLLBACK**

Dépendances non validées

Transaction T1	Temps	Transaction T2
-	-	-
-	t_1	**UPDATE**(A)
RETRIEVE(A) ou **UPDATE**(A)	t_2	-
-	t_3	**ROLLBACK**

Problème

T_1 travaille sur une valeur de A qui n'est pas valide.

Analyse incohérente

Trois comptes C1, C2 et C3 qui présentent respectivement des soldes de 40, 50 et 20.

Transaction T1	Temps	Transaction T2
-	-	-
RETRIEVE(C1)	t_1	-
sum = 40	-	-
RETRIEVE(C2)	t_2	-
sum = 90	-	-
-	t_3	**UPDATE**(C3,30)
-	t_4	**UPDATE**(C1,30)
RETRIEVE(C3)	t_5	-
sum = 120	-	-

Problèmes posés par la concurrence

Les problèmes viennent du fait que l'on cherche à entrelacer des transactions.

Problèmes posés par la concurrence

Les problèmes viennent du fait que l'on cherche à entrelacer des transactions.

Comment résoudre ces problèmes ?

1. en proposant un modèle formel de l'entrelacement des transactions ;
2. en caractérisant les « bons » entrelacements ;
3. en cherchant des conditions «pratiques » suffisantes.

Ordonnancement : définitions et hypothèses

Hypothèse

Toute transaction exécutée isolément des autres transactions amène la base de données d'un état cohérent vers un état cohérent.

Ordonnancement : définitions et hypothèses

Hypothèse

Toute transaction exécutée isolément des autres transactions amène la base de données d'un état cohérent vers un état cohérent.

Définition (ordonnancement)

Soient $T_1, \ldots, T_n$ des transactions. On appelle **ordonnancement** *une séquence d'actions élémentaires de lecture et d'écriture effectuées par les transactions $T_1, \ldots, T_n$. Cette séquence est complète : pour $i \in \{1, \ldots, n\}$ toutes les opérations effectuées par T_i se retrouvent dans la séquence. Soit T_i une transaction. On représentera par $r_i(A)$ une lecture de l'élément A de la base de données par la transaction T_i. On représentera par $w_i(A)$ une écriture de l'élément A de la base de données par la transaction T_i.*

Ordonnancement : exemple

Soit une transaction T_1 effectuant les opérations suivantes sur la base de données :

- lecture d'un élément A
- modification de A
- lecture d'un élément B
- modification de B

Soit une deuxième transaction T_2 effectuant les mêmes opérations. Un ordonnancement pour T_1 et T_2 est :

$$r_1(A); w_1(A); r_2(A); w_2(A); r_2(B); r_1(B); w_2(B); w_1(B)$$

Séquentialité

Définition (séquentialité)

Soient $T_1, \ldots, T_n$ des transactions. Un ordonnancement σ sur $T_1, \ldots, T_n$ est séquentiel ssi $\forall i \in \{1, \ldots, n\}$ $\forall i \in \{1, \ldots, n\}$ $i \neq j$, si une action de T_i précède une action de T_j dans σ, alors toutes les actions de T_i précédent toutes les actions de T_j dans σ.

Séquentialité

Définition (séquentialité)

Soient $T_1, \ldots, T_n$ des transactions. Un ordonnancement σ sur $T_1, \ldots, T_n$ est séquentiel ssi $\forall i \in \{1, \ldots, n\}$ $\forall i \in \{1, \ldots, n\}$ $i \neq j$, si une action de T_i précède une action de T_j dans σ, alors toutes les actions de T_i précédent toutes les actions de T_j dans σ.

Si on reprend l'exemple précédent :

$$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$$

est un ordonnancement séquentiel.

Sérialisabilité

Considérons maintenant l'ordonnancement suivant :

$$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$$

Sérialisabilité

Considérons maintenant l'ordonnancement suivant :

$$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$$

Définition (sérialisabilité)

Un ordonnancement est sérialisable si son exécution produit le même résultat qu'un ordonnancement séquentiel.

Sérialisabilité par conflit

Condition **suffisante** pour garantir la sérialisabilité.

Définition (conflit)

Soient deux actions consécutives dans un ordonnancement, alors ces deux actions sont en **conflit** *si lorsque l'on les permute dans l'ordonnancement, l'effet d'au moins une de ces actions est changé.*

Sérialisabilité par conflit

Condition **suffisante** pour garantir la sérialisabilité.

Définition (conflit)

Soient deux actions consécutives dans un ordonnancement, alors ces deux actions sont en **conflit** *si lorsque l'on les permute dans l'ordonnancement, l'effet d'au moins une de ces actions est changé.*

Soient deux transactions différentes T_i et T_j et deux éléments d'une base de données X et Y :

- $r_i(X); w_i(Y)$ sont en conflit ;
- $w_i(X); w_j(X)$ sont en conflit ;
- $r_i(X); w_j(X)$ et $w_i(X); r_j(X)$ sont en conflit.

Sérialisabilité par conflit

Définition (équivalence par conflit)

On dit que deux ordonnancements sont **équivalents par conflit** *si on peut transformer l'un en l'autre par une série d'échanges d'actions non conflictuels. Un ordonnancement est* **sérialisable par conflit** *s'il est équivalent par conflit à un ordonnancement séquentiel.*

Graphe de précédence

Définition

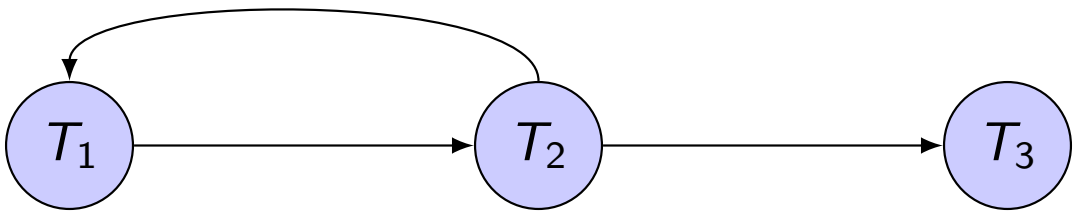
Soit σ un ordonnancement comportant les transactions T_1 et T_2. On dit que T_1 précède T_2, noté $T_1 <_\sigma T_2$ s'il existe une action A_1 de T_1 et une action A_2 de T_2 telles que :

- *A_1 précède A_2 dans σ ;*
- *A_1 et A_2 concernent le même élément de la base ;*
- *au moins une des deux actions est une action d'écriture.*

On construit un graphe de précédence de la façon suivante : chaque nœud du graphe est une transaction et il existe un arc du nœud T_i au nœud T_j si $T_i <_\sigma T_j$.

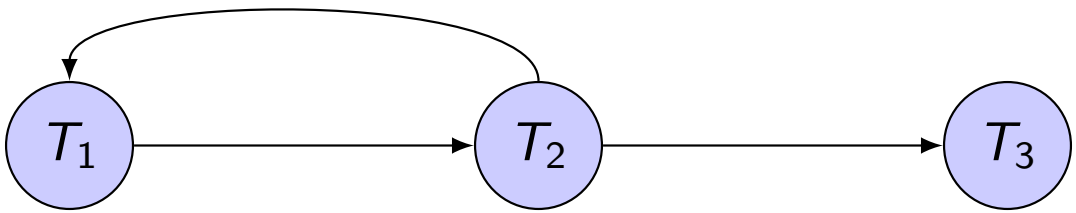
Graphe de précédence

$$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$$



Graphe de précédence

$$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$$



Théorème

Un ordonnancement ayant un graphe de précédence acyclique est sérialisable par conflit. Un ordonnancement sérialisable par conflit a un graphe de précédence acyclique.

Principes du verrouillage

Principe (cohérence des transactions)

Une transaction ne peut accéder ou modifier un élément que si elle a un verrou sur cet élément et qu'elle ne l'a pas relâché. Si une transaction a un verrou sur un élément, elle devra relâcher ce verrou. C'est le principe de **cohérence des transactions**.

Principes du verrouillage

Principe (cohérence des transactions)

Une transaction ne peut accéder ou modifier un élément que si elle a un verrou sur cet élément et qu'elle ne l'a pas relâché. Si une transaction a un verrou sur un élément, elle devra relâcher ce verrou. C'est le principe de **cohérence des transactions**.

Principe (légalité des ordonnancements)

Deux transactions ne peuvent pas posséder en même temps un verrou sur un même élément. C'est le principe de **légalité des ordonnancements**.

Verrouillage à deux phases

Définition

Pour toute transaction, toutes les demandes de verrous précédent les demandes de déverrouillage.

Verrouillage à deux phases

Définition

Pour toute transaction, toutes les demandes de verrous précédent les demandes de déverrouillage.

Cela signifie deux choses :

- avant d'agir sur un élément (par exemple, un n-uplet d'une base de données), une transaction doit obtenir un verrou sur cet élément.
- après l'abandon d'un verrou, une transaction ne doit plus jamais pouvoir obtenir de verrous.

Verrouillage à deux phases

Définition

Pour toute transaction, toutes les demandes de verrous précédent les demandes de déverrouillage.

Cela signifie deux choses :

- avant d'agir sur un élément (par exemple, un n-uplet d'une base de données), une transaction doit obtenir un verrou sur cet élément.
- après l'abandon d'un verrou, une transaction ne doit plus jamais pouvoir obtenir de verrous.

Théorème

Le protocole du verrouillage à deux phases garantit la sérialisabilité par conflit.

Verrous exclusifs et partagés

2PL trop restrictif : introduction de verrous avec modes.

Verrous exclusifs et partagés

2PL trop restrictif : introduction de verrous avec modes.

		Lecture	Écriture
Verrous	X	Oui	Oui
	S	Oui	Non

		Demande de verrou	
		X	S
Verrou posé	X	Non	Non
	S	Non	Oui

Verrous exclusifs et partagés

Principe

Une transaction qui souhaite accéder à la valeur d'un élément doit d'abord obtenir un verrou S sur cet élément.
Une transaction qui souhaite modifier un élément doit d'abord obtenir un verrou X sur cet élément.
Si une demande de verrou émise par la transaction T_2 est refusée car elle entre en conflit avec un verrou déjà détenu par la transaction T_1, la transaction T_2 est mise en attente. T_2 devra attendre jusqu'à ce que T_1 abandonne le verrou.
Une transaction possédant un verrou de type S sur un élément peut demander un verrou de type X sans relâcher son verrou.

Problème de perte de mise à jour

Transaction T1	Temps	Transaction T2
-	-	-
RETRIEVE(A)	t_1	-
$sl_1(A)$	-	-
-	t_2	RETRIEVE(A)
-	-	$sl_2(A)$
UPDATE(A)	t_3	-
$xl_1(A)$	-	-
attente	-	-
attente	t_4	**UPDATE**(A)
attente	-	$xl_2(A)$
attente	-	attente

Problème de dépendance non validée

Transaction T1	Temps	Transaction T2
-	-	-
-	t_1	**UPDATE**(A)
-	-	$xl_2(A)$
RETRIEVE(A) ou **UPDATE**(A)	t_2	-
$sl_1(A)$ ou $xl_1(A)$	-	-
attente	-	-
attente	t_3	**ROLLBACK**
attente	-	$u_2(A)$
obtention du verrou	-	-

Problème d'analyse incohérente

Transaction T1	Temps	Transaction T2
-	-	-
RETRIEVE(C1)	t_1	-
$sl_1(C1)$	-	-
sum = 40	-	-
RETRIEVE(C2)	t_2	-
$sl_1(C2)$	-	-
sum = 90	-	-
-	t_3	**UPDATE**(C3,30)
-	-	$xl_2(C3)$
-	t_4	**UPDATE**(C1,30)
-	-	$xl_2(C1)$
-	-	attente
RETRIEVE(C3)	t_5	attente
$sl_1(C3)$	-	attente
attente	-	attente

Graphe d'attente

Problème du blocage à résoudre : on utilise un graphe d'attente.

Graphe d'attente

Problème du blocage à résoudre : on utilise un graphe d'attente.

Définition (graphe d'attente)

Soit $T_1, \ldots, T_n$ des transactions. Un graphe d'attente est un graphe où :

- *les nœuds du graphe sont les transactions ;*
- *il existe un arc du noeud T_i vers le nœud T_j ssi il existe un élément de la base A tel que T_j possède un verrou sur A, T_i attende un verrou sur A et que T_1 ne peut obtenir son verrou que lorsque T_j aura relâché le sien.*

Graphe d'attente

Problème du blocage à résoudre : on utilise un graphe d'attente.

Définition (graphe d'attente)

Soit $T_1, \ldots, T_n$ des transactions. Un graphe d'attente est un graphe où :

- *les nœuds du graphe sont les transactions ;*
- *il existe un arc du noeud T_i vers le nœud T_j ssi il existe un élément de la base A tel que T_j possède un verrou sur A, T_i attende un verrou sur A et que T_1 ne peut obtenir son verrou que lorsque T_j aura relâché le sien.*

Il faut choisir une **victime**.

Granularité

On peut poser des verrous sur différentes parties de la base.

Les niveaux de granularité considérés sont les suivants :

1. la relation est l'élément de plus haut niveau que l'on peut verrouiller ;
2. chaque relation est composée de **blocs** qui contiennent eux-même des tuples ;
3. enfin chaque tuple est verrouillable.

Granularité

On peut poser des verrous sur différentes parties de la base.

Les niveaux de granularité considérés sont les suivants :

1. la relation est l'élément de plus haut niveau que l'on peut verrouiller ;
2. chaque relation est composée de **blocs** qui contiennent eux-même des tuples ;
3. enfin chaque tuple est verrouillable.

Problème

Si on veut poser un verrou X sur la base, faut-il vérifier tous les tuples de la base ?

Principe du verrouillage intentionnel

Principe

Le principe du verrouillage intentionnel est le suivant :

1. *si l'on veut placer un verrou S ou X sur un élément, on commence en haut de la hiérarchie ;*
2. *si on se situe sur l'élément que l'on veut verrouiller, on demande alors le verrou correspondant ;*
3. *si l'élément est « plus bas » dans la hiérarchie, alors on pose un verrou intentionnel correspondant sur ce nœud.*

Acquisition de verrous

		Verrou demandé			
		IS	*IX*	*S*	*X*
Verrou Posé	*IS*	Oui	Oui	Oui	Non
	IX	Oui	Oui	Non	Non
	S	Oui	Non	Oui	Non
	X	Non	Non	Non	Non

Attention, ce système de verrouillage intentionnel peut conduire à l'apparition de données **fantômes**.

Neuvième partie

Gestion des transactions

Principe du commit à deux phases

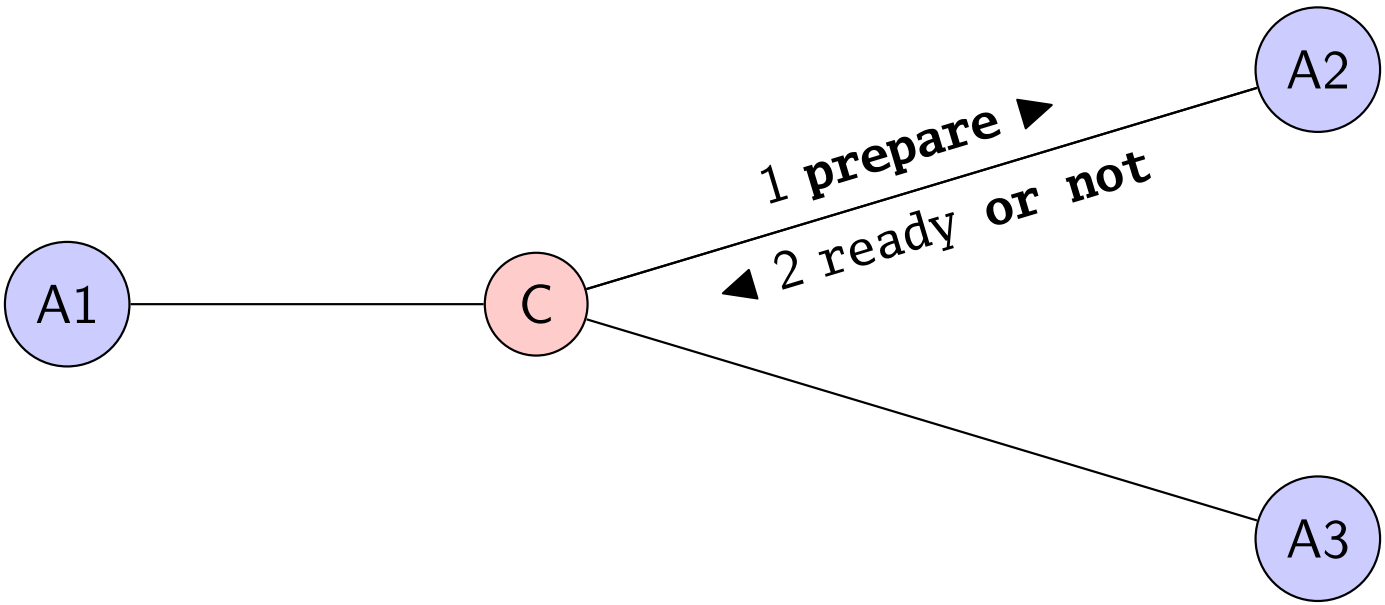
Problème des transactions concernant des bases de données distribuées : comment être sûr que toutes les bases ont effectué les modifications ?

- chaque base de données a son propre journal et il n'y a pas de journal global ;
- il existe un site, appelé **coordinateur**, qui joue un rôle spécial (il peut s'agir par exemple du site qui a initié la transaction). Son rôle est de garantir que les différents SGBD valident où annulent les mises à jour dont ils sont responsables à l'unisson.

Phase I du commit

1. le coordinateur place une entrée <**Prepare** T> dans son journal ;
2. le coordinateur envoie le message **prepare** T aux différents sites ;
3. chaque site recevant le message **prepare** T décide de valider (*commit*) ou d'annuler (*rollback*) les composants de T le concernant ;
4. si un site veut valider les composants de T le concernant, il entre dans un état dit *precommitted*. Dans cet état, le site ne peut plus annuler les composants de T sans un ordre du coordinateur. Le site s'assure localement que les composants de T ne devront pas être annulés en cas de panne du système, écrit l'entrée <Ready T> dans son journal et envoie le message ready T au coordinateur ;
5. si le site veut annuler les composants de T, il écrit l'entrée <Don't commit T> dans son journal et envoie le message don't commit T au coordinateur. Il peut alors annuler les composants de T le concernant.

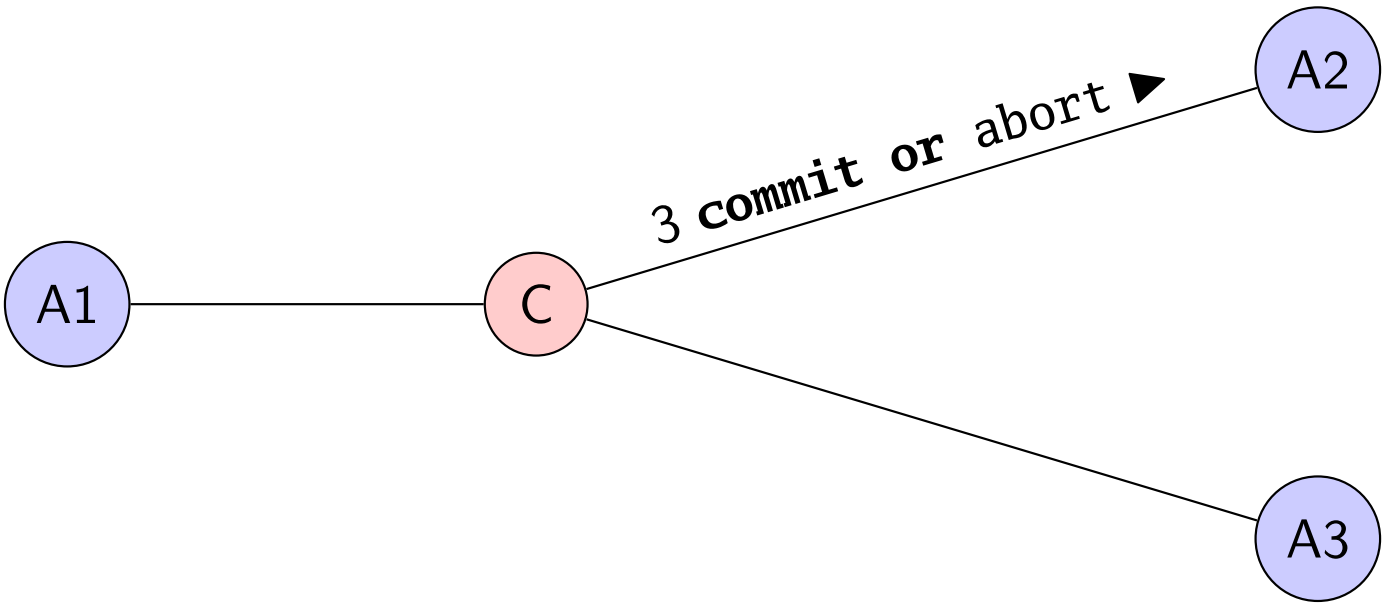
Phase I du commit



Phase II du commit

1. si le coordinateur a reçu le message `ready T` de la part de tous les sites, alors il écrit l'entrée <**`Commit T`**> dans son journal et envoie le message **`commit T`** à tous les sites ;
2. si le coordinateur a reçu le message `don't commit T` de la part d'un des sites, alors il écrit l'entrée <`Abort T`> dans son journal et envoie le message `abort T` à tous les sites ;
3. si un site reçoit un message **`commit T`**, il valide les composants de `T` le concernant et écrit <**`Commit T`**> dans son journal ;
4. si un site reçoit un message `abort T`, il annule les composants de `T` le concernant et écrit <`Abort T`> dans son journal ;

Phase II du commit



Neuvième partie

Gestion des transactions

COMMIT, ROLLBACK et degrés d'isolation

Par défaut, en utilisant l'interpréteur, toute commande est une transaction.

```
START TRANSACTION
...
[COMMIT|ROLLBACK]
```

COMMIT, ROLLBACK et degrés d'isolation

Par défaut, en utilisant l'interpréteur, toute commande est une transaction.

```
START TRANSACTION
...
[COMMIT|ROLLBACK]
```

On peut spécifier le fait qu'une transaction ne peut que lire des données.

```
SET TRANSACTION READ ONLY;

SET TRANSACTION READ WRITE;
```

COMMIT, ROLLBACK et degrés d'isolation

Par défaut, en utilisant l'interpréteur, toute commande est une transaction.

```
START TRANSACTION
...
[COMMIT|ROLLBACK]
```

On peut spécifier le fait qu'une transaction ne peut que lire des données.

```
SET TRANSACTION READ ONLY;

SET TRANSACTION READ WRITE;
```

On peut spécifier le niveau d'isolation d'une transaction.

```
SET TRANSACTION ISOLATION LEVEL [SERIALIZABLE|REPEATABLE READ|
                                 READ COMMITTED|READ UNCOMMITTED]
```

Comportements anormaux

- la lecture salissante (*dirty read*) : supposons que la transaction T_1 effectue une mise à jour sur un certain tuple, que la transaction T_2 récupère ensuite ce tuple et que la transaction T_1 soit annulée par un **ROLLBACK**. Les valeurs des attributs du tuple observé par T_2 sont alors fausses.
- la lecture non renouvelable : supposons que la transaction T_1 récupère un tuple, que la transaction T_2 effectue ensuite une mise à jour de ce tuple et que la transaction T_1 récupère de nouveau le « même » tuple. La transaction T_1 a en fait récupéré le même tuple deux fois mais a observé des valeurs différentes.
- le fantôme : supposons que la transaction T_1 récupère un ensemble de tuples qui satisfont une certaine condition. Supposons que la transaction T_2 insère ensuite une ligne qui satisfait la même condition. Si la transaction T_1 répète maintenant la même demande, elle observera une ligne qui n'existait pas précédemment (que l'on appelle un fantôme).

Lien entre comportements anormaux et niveaux d'isolation

	lecture salissante	lecture non renouvelable	fantôme
READ UNCOMMITTED	Oui	Oui	Oui
READ COMMITTED	Non	Oui	Oui
REPEATABLE **READ**	Non	Non	Oui
SERIALIZABLE	Non	Non	Non

Conclusion

Gestion des transactions : garantir les propriétés ACID

- reprise après panne
- gestion de la concurrence par verrous

Gestion des transactions assez transparente pour l'utilisateur.

Dixième partie

Bases de données objets

Introduction

Le modèle relationnel est un modèle simple : tables plates, types simples.
Comment avoir des attributs plus complexes ?

➥ avantages des langages orientés objets (collections. . .)

Introduction

Le modèle relationnel est un modèle simple : tables plates, types simples.
Comment avoir des attributs plus complexes ?

➥ avantages des langages orientés objets (collections. . .)

Problème

Comment gérer la persistance des objets ?

Une solution : sauvegarder l'état des objets dans un fichier.

Introduction

Le modèle relationnel est un modèle simple : tables plates, types simples.
Comment avoir des attributs plus complexes ?

➥ avantages des langages orientés objets (collections. . .)

Problème

Comment gérer la persistance des objets ?

Une solution : sauvegarder l'état des objets dans un fichier.

Mais :

- la persistance n'est pas transparente ;
- la programmation de ces opérations de sauvegarde est délicate ;
- l'accès aux objets en mémoire ;
- problème de la concurrence sur accès à des fichiers.

Introduction

Le modèle relationnel est un modèle simple : tables plates, types simples.
Comment avoir des attributs plus complexes ?

➥ avantages des langages orientés objets (collections. . .)

Problème

Comment gérer la persistance des objets ?

Une solution : sauvegarder l'état des objets dans un fichier.

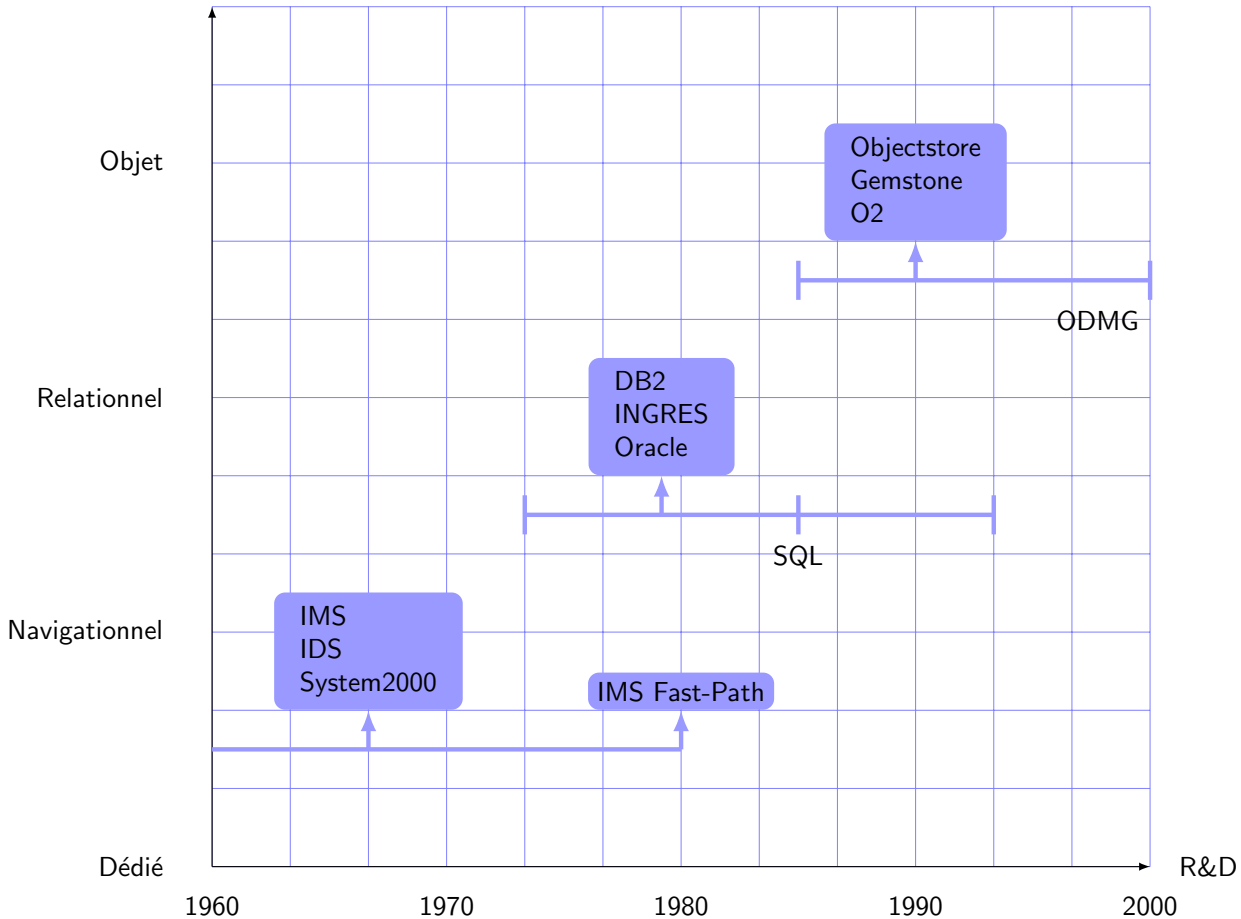
Mais :

- la persistance n'est pas transparente ;
- la programmation de ces opérations de sauvegarde est délicate ;
- l'accès aux objets en mémoire ;
- problème de la concurrence sur accès à des fichiers.

Deux approches :

- approche relationnel-objet ;
- approche orientée objet.

Historique des bases de données orientées objets



Pourquoi des bases de données orientées objets ?

Nature des applications :

- classiquement : types de données simples ;
- actuellement : graphiques, texte, données géométriques, structures complexes. . .

Pourquoi des bases de données orientées objets ?

Nature des applications :

- classiquement : types de données simples ;
- actuellement : graphiques, texte, données géométriques, structures complexes. . .

Gestion des données fortement structurées :

- relations entre ensemble de n-uplets
  - ➥ jointures coûteuses

Pourquoi des bases de données orientées objets ?

Nature des applications :

- classiquement : types de données simples ;
- actuellement : graphiques, texte, données géométriques, structures complexes. . .

Gestion des données fortement structurées :

- relations entre ensemble de n-uplets
  - ➥ jointures coûteuses

Intégration dans un langage de programmation :

- pas de structures de contrôle avec SQL ;
- intégration à un langage : délicate (styles de programmation différents).

Pourquoi des bases de données orientées objets ?

Nature des applications :

- classiquement : types de données simples ;
- actuellement : graphiques, texte, données géométriques, structures complexes. . .

Gestion des données fortement structurées :

- relations entre ensemble de n-uplets
  - ➥ jointures coûteuses

Intégration dans un langage de programmation :

- pas de structures de contrôle avec SQL ;
- intégration à un langage : délicate (styles de programmation différents).

Conversion de types (*impedance mismatch*).

Pourquoi des bases de données orientées objets ?

Nature des applications :

- classiquement : types de données simples ;
- actuellement : graphiques, texte, données géométriques, structures complexes. . .

Gestion des données fortement structurées :

- relations entre ensemble de n-uplets
  - ➥ jointures coûteuses

Intégration dans un langage de programmation :

- pas de structures de contrôle avec SQL ;
- intégration à un langage : délicate (styles de programmation différents).

Conversion de types (*impedance mismatch*).

Interfaces de manipulation.

Dixième partie

Bases de données objets

Objectifs

Objectifs :

- réduire, voire éliminer le dysfonctionnement (*impedance mismatch*) entre langage de programmation et langage de bases de données ;
- supporter directement les objets arbitrairement complexes grâce à un modèle à objets ;
- partager le code réutilisable des applications au sein du SGBD.

Sémantique	**Objets**
entité	objet
identité d'entité	identité d'objet
type	classe
attribut	attribut
agrégation	classe dépendante
généralisation	héritage
associations	
	opération
	encapsulation

Définitions

Définition (base de données objets)

Une base de données objets est une organisation cohérente d'objets persistants et partagés par des utilisateurs concurrents.

SGBDO :

- intégrité des données ;
- permanence des données ;
- gestion des transactions ;
- indépendance physique des données ;
- possibilité de requêtes.

Définition (SGBDO)

La définition minimale d'un SGBDO est donnée par :

$$SGBDO = SGBD + objets + héritage + polymorphisme$$

Schéma conceptuel d'une base de données à objets

Définition (schéma conceptuel)

Modèle conceptuel $= \{Classes + Associations\}$

On peut utiliser des notations semi-formelles comme UML.e

Deux approches pour les bases de données à objets

Définition (relationnel-objet)

L'approche relationnel-objet est fondée sur une extension du modèle relationnel et de son langage de référence SQL avec les concepts de la programmation orientée objet.

Deux approches pour les bases de données à objets

Définition (relationnel-objet)

L'approche relationnel-objet est fondée sur une extension du modèle relationnel et de son langage de référence SQL avec les concepts de la programmation orientée objet.

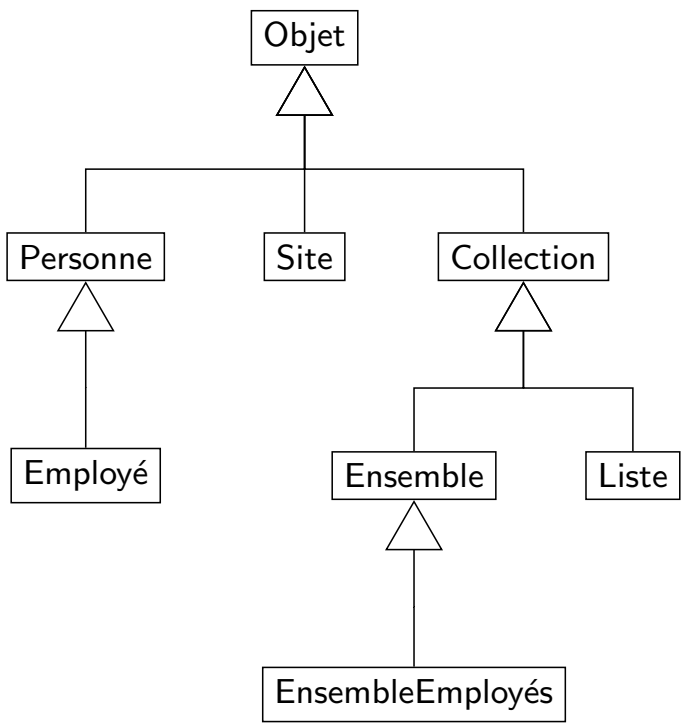
Définition (orienté objet)

L'approche orientée objet repose sur une extension d'un modèle à objets pour la définition et la manipulation d'objets persistants.

Caractéristiques principales

Ce sont des caractéristiques communes aux deux approches :

- gestion de la persistance. Il faut pouvoir manipuler de façon transparente les objets persistants comme des objets temporaires.
  - ➥ primitives de manipulation et de liaison des objets.
    - persistance « manuelle » : on le fait « à la main »
    - persistance par héritage (au niveau de la classe) ;
    - persistance par référence (au niveau de l'objet).
- support des collections : algèbre étendue aux objets complexes



Caractéristiques principales

- évolution des schémas : favoriser la réutilisation de façon transparente.
  - modification d'un attribut de classe : transparente si l'encapsulation est respectée ;
  - modification d'une opération de classe : doit être répercutée sur les opérations des programmes et éventuellement redéfinie dans les sous-classes ;
  - modification des liens d'héritage d'une classe ;
  - modification du graphe de classes.

Caractéristiques principales

- évolution des schémas : favoriser la réutilisation de façon transparente.
  - modification d'un attribut de classe : transparente si l'encapsulation est respectée ;
  - modification d'une opération de classe : doit être répercutée sur les opérations des programmes et éventuellement redéfinie dans les sous-classes ;
  - modification des liens d'héritage d'une classe ;
  - modification du graphe de classes.

Utilisation de la liaison dynamique pour garantir la cohérence des applications.

Caractéristiques principales

- évolution des schémas : favoriser la réutilisation de façon transparente.
  - modification d'un attribut de classe : transparente si l'encapsulation est respectée ;
  - modification d'une opération de classe : doit être répercutée sur les opérations des programmes et éventuellement redéfinie dans les sous-classes ;
  - modification des liens d'héritage d'une classe ;
  - modification du graphe de classes.

Utilisation de la liaison dynamique pour garantir la cohérence des applications.

Comment garantir la cohérence des objets en présence de modification de schéma :

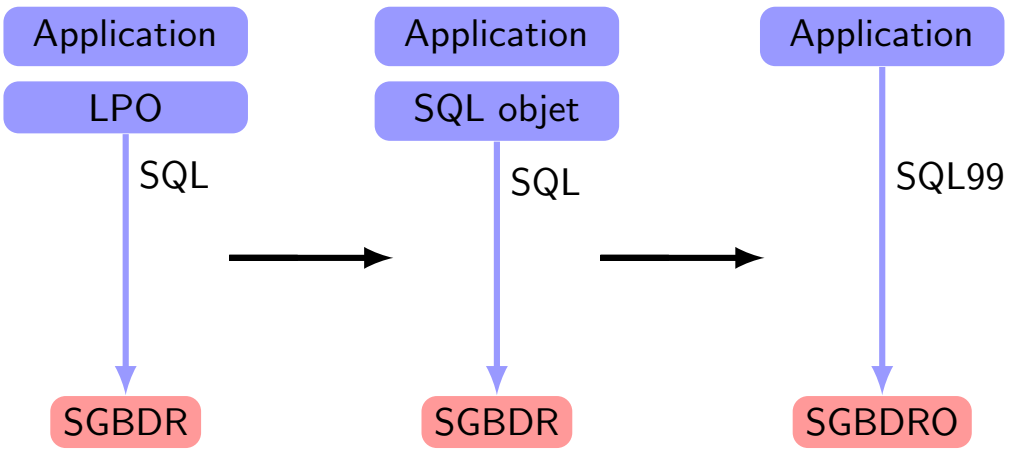
- approche manuelle : interdiction de mise à jour d'un type dès que des instances de ce type existent.
- approche mise à jour immédiate : propage automatiquement et immédiatement les mises à jour du schéma sur les objets concernés.
- approche mise à jour paresseuse : diffère la mise à jour des objets jusqu'au prochain accès en mémoire.

Dixième partie

Bases de données objets

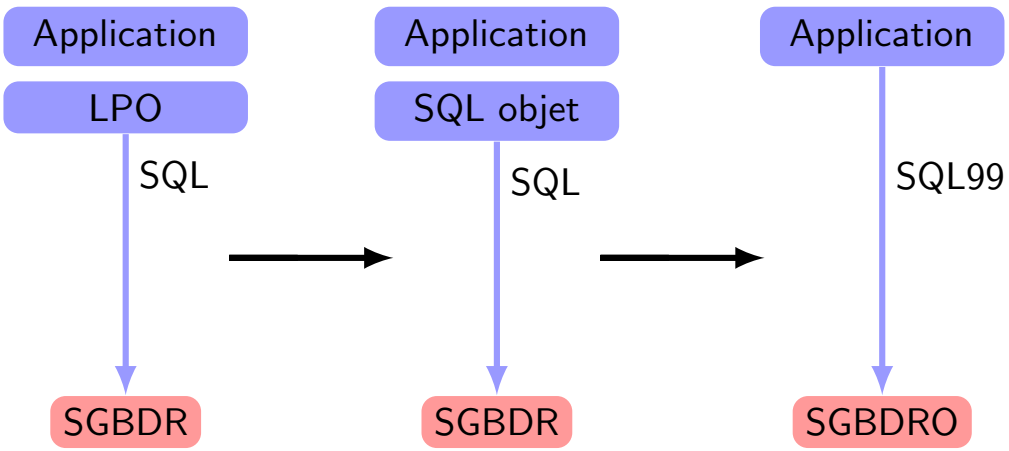
Évolution du relationnel à l'objet

Une norme : SQL :1999 ou SQL3.



Évolution du relationnel à l'objet

Une norme : SQL :1999 ou SQL3.



Deux extensions :

- définition de domaines spécifiée par la définition de types.
- le domaine et les relations entre domaines peuvent incorporer la notion d'identité essentielle à l'approche objets.

Définition des objets

```
CREATE TYPE T AS (
    ...
    nomChamp TypeChamp,
    ...
);
```

Définition des objets

```
CREATE TYPE T AS (
   ...
   nomChamp TypeChamp,
   ...
);
```

Exemple :

```
CREATE TYPE Employe AS (
   nom             CHAR(10),
   date_naissance DATE
);
```

Création de méthode

Déclaration dans le type :

```
CREATE TYPE T AS (
    ...
    nomChamp TypeChamp,
    ...
)
METHOD nomMethode() RETURNS Type;
```

Création de méthode

Déclaration dans le type :

```
CREATE TYPE T AS (
    ...
    nomChamp TypeChamp,
    ...
)
METHOD nomMethode() RETURNS Type;
```

Création de la méthode :

```
CREATE METHOD nomMethode() RETURNS Type
    Corps de la methode
```

Création de méthode : exemple

```
CREATE TYPE Employe AS (
   nom             CHAR(10),
   date_naissance DATE
)
METHOD age() RETURNS INTEGER(4);

CREATE METHOD age() RETURNS INTEGER
LANGUAGE SQL
DETERMINISTIC
CONTAINS SQL
RETURN NULL ON NULL INPUT
BEGIN
  RETURN CAST(INTERVAL DAY (CURRENT_DATE - ADT_PERSONNE.PSR_DATE_NAISSANCE)
              AS FLOAT) / 365.2422
END;
```

Typage des tuples d'une relation

```
CREATE TABLE nomTable OF Type(
  ...
);
```

Typage des tuples d'une relation

```
CREATE TABLE nomTable OF Type(
  ...
);
```

Exemple :

```
CREATE TABLE listeEmployes OF Employe(
  date_arrivee   DATE,
  PRIMARY KEY(nom)
);

SELECT l.nom()
FROM listeEmployes AS l
WHERE l.age() < 35;
```

Obtention d'une référence vers les tuples

On peut donc avoir soit une clé générée par le système, soit la clé primaire comme référence :

```
CREATE TABLE ... (
  ...
  REF IS attRef [SYSTEM GENERATED|DERIVED]
  ...
);
```

Obtention d'une référence vers les tuples

On peut donc avoir soit une clé générée par le système, soit la clé primaire comme référence :

```
CREATE TABLE ... (
  ...
  REF IS attRef [SYSTEM GENERATED|DERIVED]
  ...
);
```

On peut utiliser un attribut référence vers un tuple d'une autre relation :

```
attribut REF(Type) SCOPE Relation
```

Manipulation des objets

Accès aux attributs d'un tuple d'une relation R de type T défini avec les champs $c_1, \ldots, c_n$:

```
SELECT t.c_i()
FROM R AS t
WHERE t.c_j()...
```

Attention, un tuple t de R n'a qu'un seul composant : l'objet !

Manipulation des objets

Accès aux attributs d'un tuple d'une relation R de type T défini avec les champs $c_1, \ldots, c_n$:

```
SELECT t.c_i()
FROM R AS t
WHERE t.c_j()...
```

Attention, un tuple t de R n'a qu'un seul composant : l'objet !

Accès à un objet en utilisant une référence (relation R dont l'attribut *att* est de type T) :

```
DEREF(att)
```

```
att->c_i
```

Dixième partie

Bases de données objets

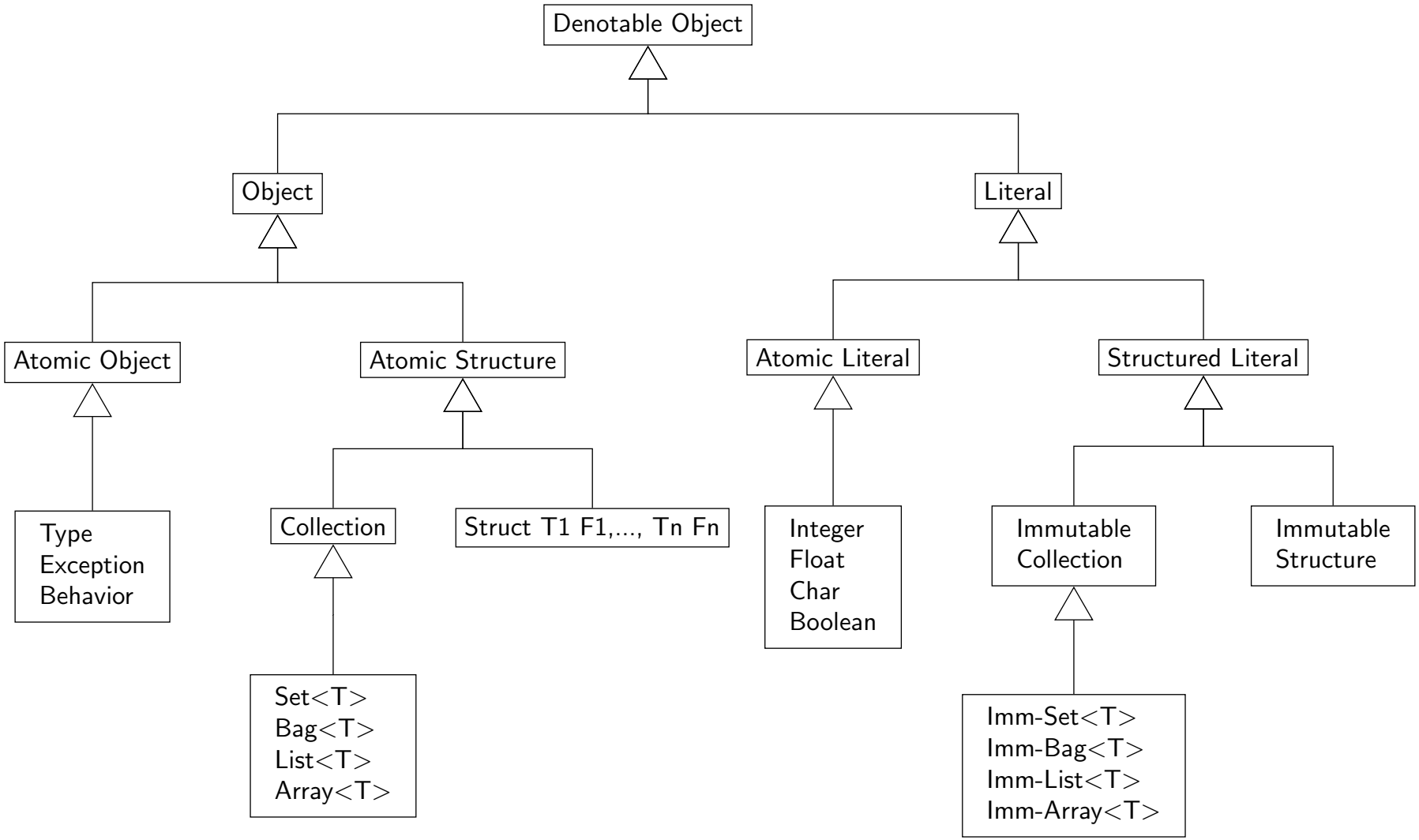
Modèle

Un standard : ODMG-93 (adopté par l'OMG en 1994).

- un modèle de données pour représenter la base de données orientée objets,
- un langage de définition de données, ODL (*Object Definition Language*),
- un langage de manipulation des objets, OML (*Object Manipulation Language*),
- un langage d'interrogation d'objets, OQL (*Object Query Language*).

Intégration à des langages de programmation : C++, Smalltalk, Java.

Modèle



Durée de vie des objets définie à la création de l'objet.

Définition d'objets avec ODL

```
class nomClasse extends C1 : ... : Cn
                (extent nomExtent key nomCle)
{
  attribute Type nomAtt1;
  attribute enum nomEnum {'String1', ..., 'Stringn'} nomAtt2;
  attribute Struct nomStruct {Type1 champ1, ..., Typen champn} nomAtt3;
  ...
  relationship [Set<C>|Bag<C>|List<C>|Array<C,i>|C] nomRelation
                                              inverse C::relation;
  ...
  TypeRetour nomMethode([in|out|inout] TypePar, ...) raises(nomException);
  ...
}
```

Définition d'objets avec ODL : exemple

```
class Site (extent sites, keys nom) {
   attribute string nom;
   attribute Employe chef;
   relationship Set<Employe> emps
                inverse Employe::affectation;
   void ajouter (in Employe)
        raises (deja_affecte);
};

class Employe (extent employes, keys nom) {
   attribute string nom;
   attribute string adresse;
   relationship Site affectation inverse Site::emps;
   void affecter (in Site)
        raises (deja_affecte, affectation_incorrecte);
};

class Ingenieur:Employe (extent ingenieurs) {
   attribute string projet;
};
```

Manipulation des objets avec OQL

Syntaxe proche de SQL, intégrable à un langage de programmation.

Manipulation des objets avec OQL

Syntaxe proche de SQL, intégrable à un langage de programmation.

Accès à un élément d'un objet par notation pointée : `a.p`

- si p est un attribut, alors $a.p$ renvoie la valeur de cet attribut ;
- si p est une relation, alors $a.p$ renvoie l'objet ou la collection d'objets associés à a par la relation p ;
- si p est une méthode, alors $a.p(...)$ est le résultat de l'application de p à a.

Manipulation des objets avec OQL

Syntaxe proche de SQL, intégrable à un langage de programmation.

Accès à un élément d'un objet par notation pointée : `a.p`

- si *p* est un attribut, alors *a.p* renvoie la valeur de cet attribut ;
- si *p* est une relation, alors *a.p* renvoie l'objet ou la collection d'objets associés à *a* par la relation *p* ;
- si *p* est une méthode, alors *a.p*(...) est le résultat de l'application de *p* à *a*.

Clause **SELECT – FROM – WHERE** :

```
SELECT [DISTINCT] a.p
FROM Collection a
WHERE C(a)
ORDER BY ...
```

Expressions quantifiées

Pour tester si les éléments d'une collection C vérifient tous une propriété P :

```
FOR ALL x IN C : P(x)
```

Expressions quantifiées

Pour tester si les éléments d'une collection C vérifient tous une propriété P :

```
FOR ALL x IN C : P(x)
```

Pour tester s'il existe au moins un élément d'une collection C vérifiant une propriété P :

```
EXISTS x IN C : P(x)
```

Conclusion

Avantages des SGBDO :

- offrent un plus grand niveau d'abstraction ;
- peuvent gérer les opérations partagées comme les données et supportent les objets complexes ;
- apportent une solution génériques aux applications complexes qui étaient jusqu'alors supportées par des systèmes dédiés.

Conclusion

Avantages des SGBDO :

- offrent un plus grand niveau d'abstraction ;
- peuvent gérer les opérations partagées comme les données et supportent les objets complexes ;
- apportent une solution génériques aux applications complexes qui étaient jusqu'alors supportées par des systèmes dédiés.

Approche relationnel-objet :

- enrichit le modèle relationnel (langage de types, sous-typage) ;
- compatibilité ascendante ;
- problèmes de compatibilité avec les langages de programmation ;
- nouveau standard : SQL :2003

Conclusion

Avantages des SGBDO :

- offrent un plus grand niveau d'abstraction ;
- peuvent gérer les opérations partagées comme les données et supportent les objets complexes ;
- apportent une solution génériques aux applications complexes qui étaient jusqu'alors supportées par des systèmes dédiés.

Approche relationnel-objet :

- enrichit le modèle relationnel (langage de types, sous-typage) ;
- compatibilité ascendante ;
- problèmes de compatibilité avec les langages de programmation ;
- nouveau standard : SQL :2003

Approche orientée objets :

- standard ODMG compatible avec l'OMG ;
- modèle et interfaces indépendants des langages de programmation ;
- groupe de travail pour intégrer OQL à SQL :1999.