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

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 = 1 073 741 824 B
1 TB	1024^4B = 1 099 511 627 776 B
1 PB	1024^5B = 1 125 899 906 842 624 B

Si un grain de riz ≡ 1 octet...

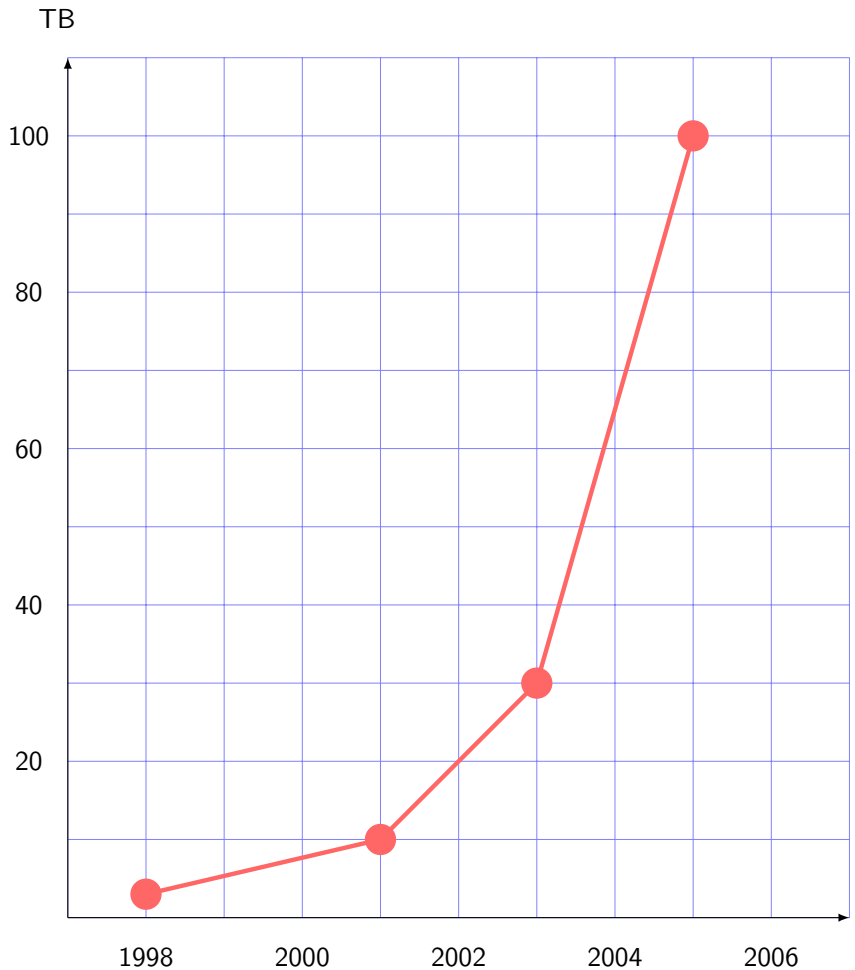
- 1 GB ≡ un sac de 25 kg de riz
- 1 PB permettent de recouvrir le centre de Londres sous 1m de riz

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/`)



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.

Septième partie

SQL (*Structured Query Language*)

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.

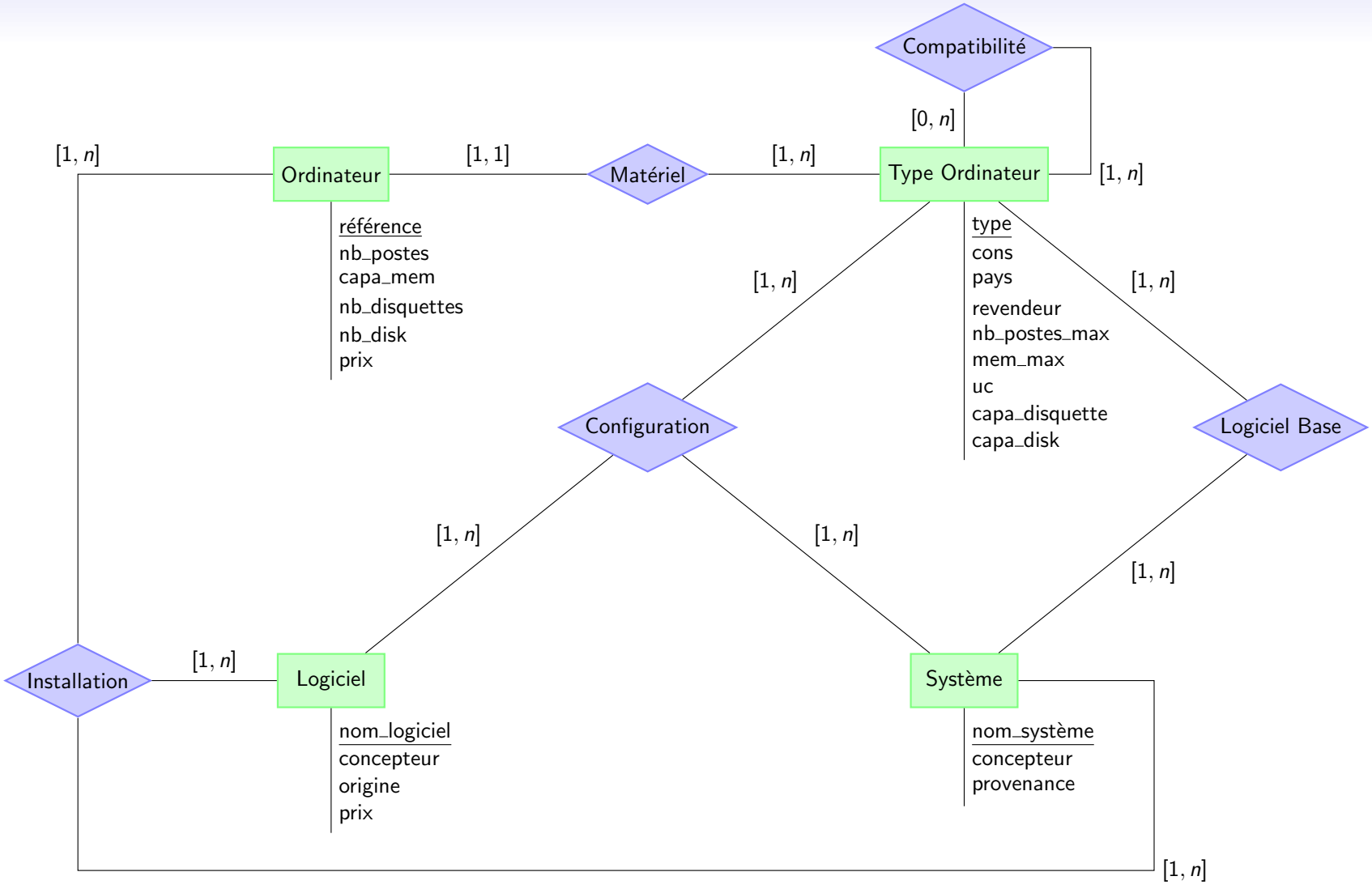
SQL est le langage déclaratif le plus utilisé actuellement.

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

Plan

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';
```

Plan

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.

Plan

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.

Plan

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;
```

Plan

PostgreSQL, un SGBD libre

- SGBD relationnel
- libre
- développé à Berkeley

PostgreSQL, un SGBD libre

- SGBD relationnel
- libre
- développé à Berkeley

Respect du standard SQL avec des limitations :

- des problèmes dans la gestion des droits ;
- pas d'assertions ;
- pas de requêtes dans les clauses **`CHECK`**
- pas de type de données complexe : BLOB. . .
- restrictions dans les *triggers* et syntaxe différente

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();
```

Dixième partie

Gestion des transactions

Plan

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
```

Plan

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 ?

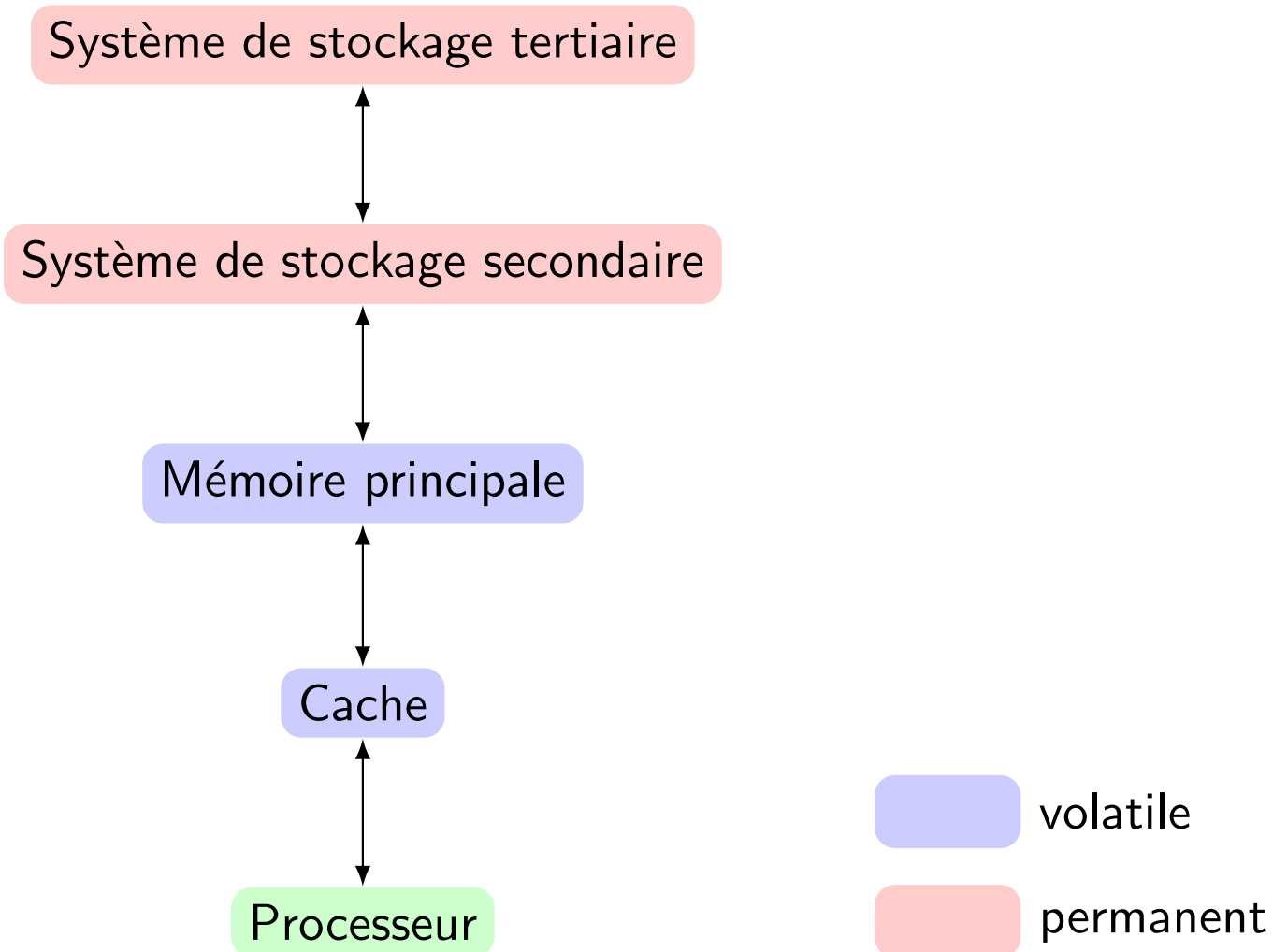
Principe

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

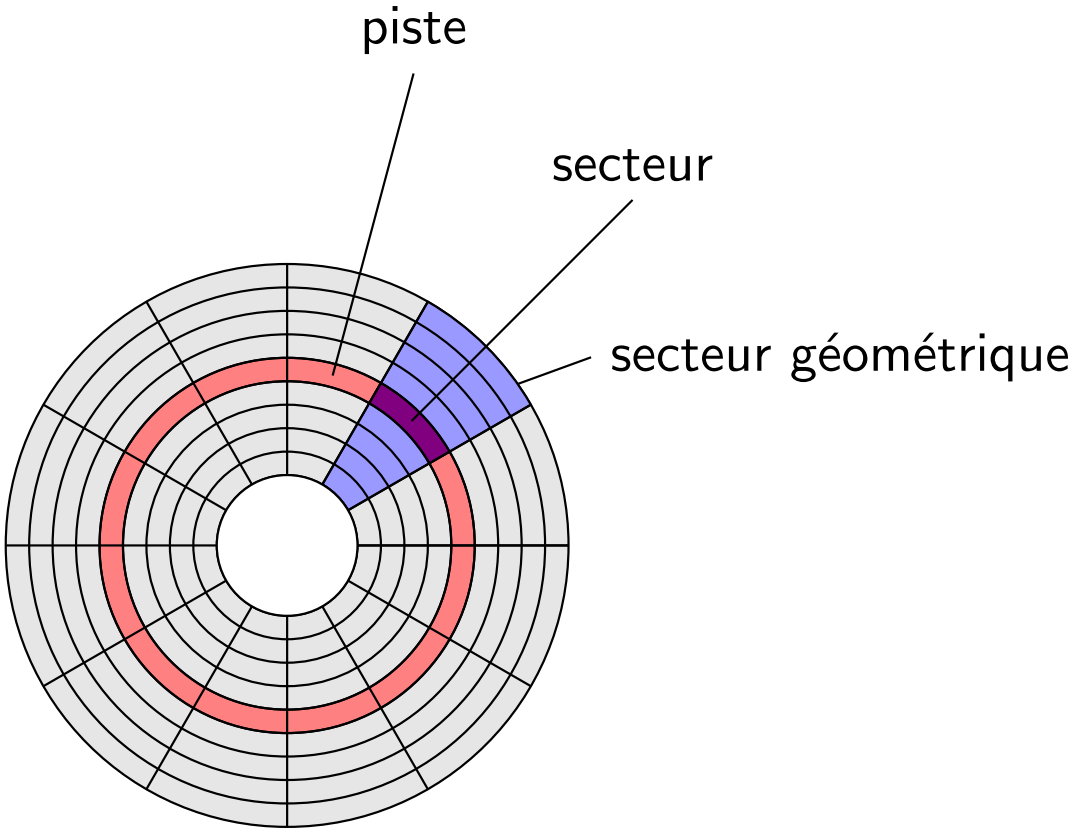
Problème

Écrit-on chaque changement sur le disque dur ?

Retour sur la structure mémoire d'une machine

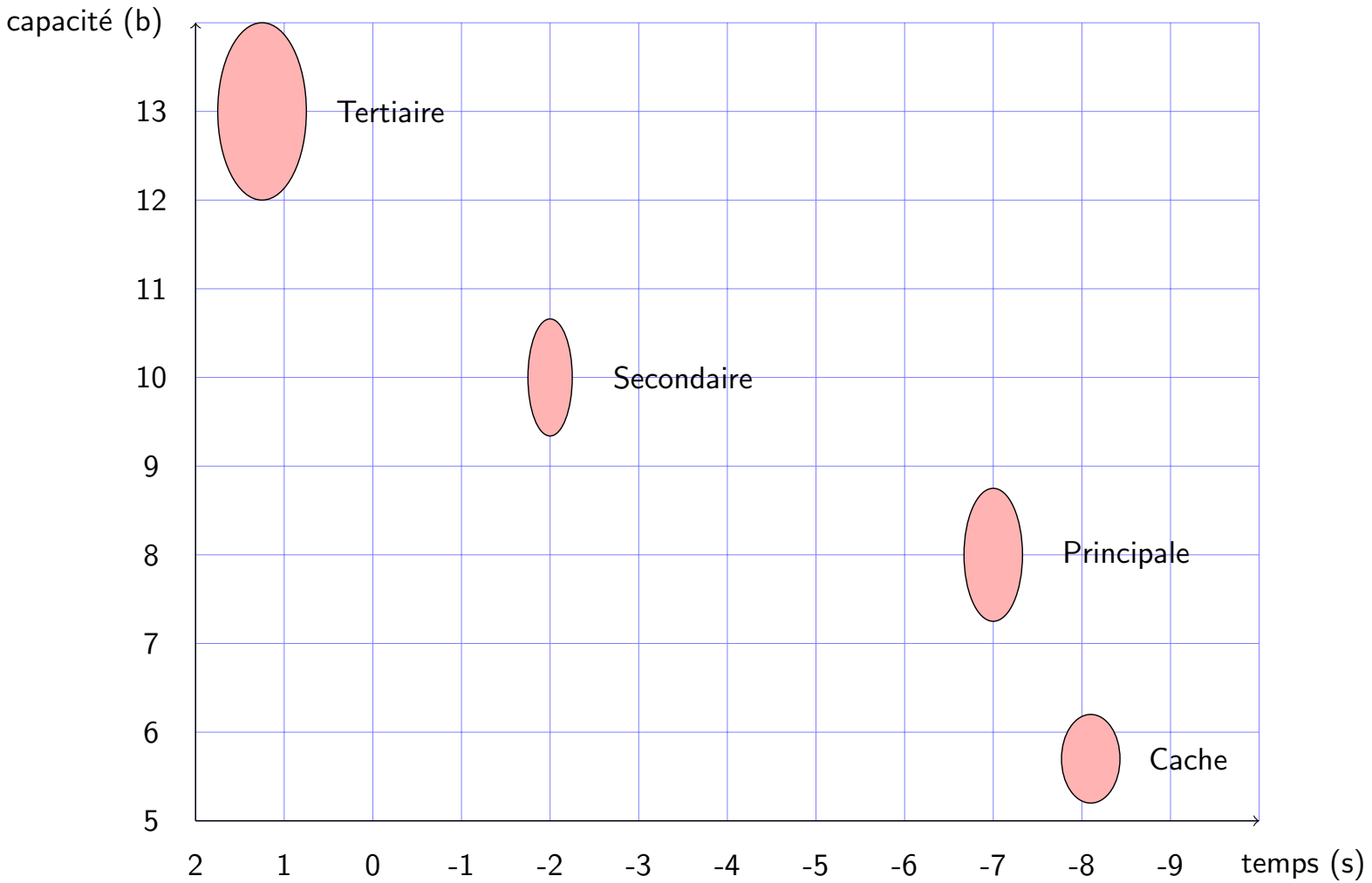


Structure d'un disque dur



- 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.

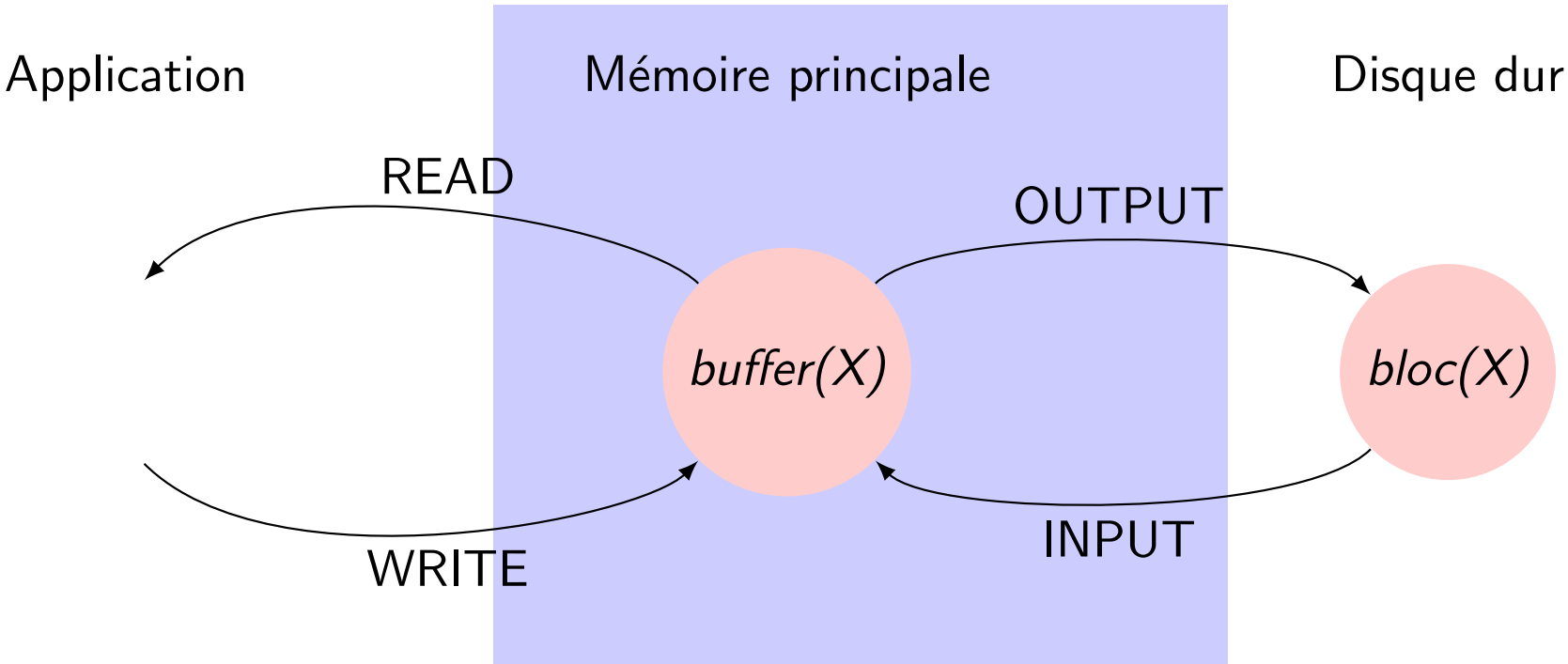
- on ne peut pas faire une écriture à chaque opération, cela pénaliserait les performances.
- lorsque le système est redémarré, les données en mémoire centrale disparaissent.

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}

Plan

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

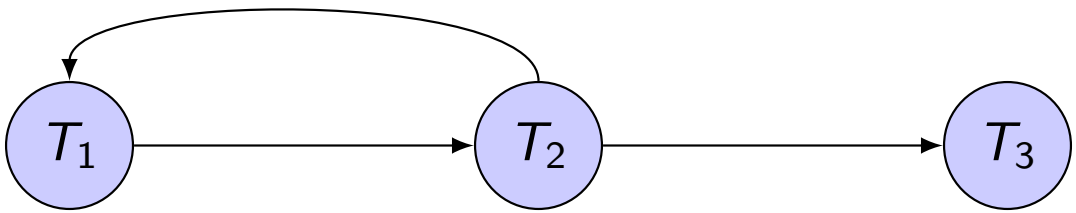
Soit σ un ordonnancement comportant les transactions T_1 et T_2. On dit que T_1 précède T_2, noté $T_1 <_\sigma T_2$ s'il existe une action A_1 de T_1 et une action A_2 de T_2 telles que :

- *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 $T_i <_\sigma T_j$.

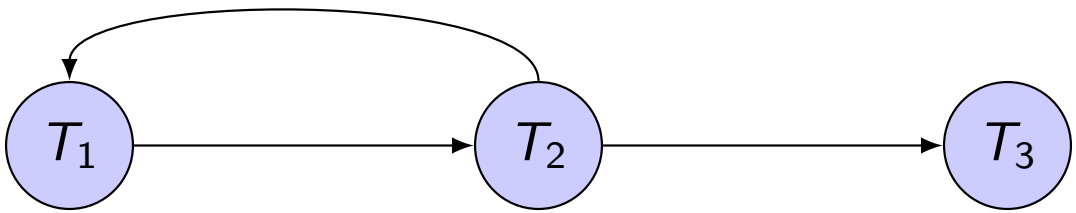
Graphe de précédence

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



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**.

Plan

Principe du commit à deux phases

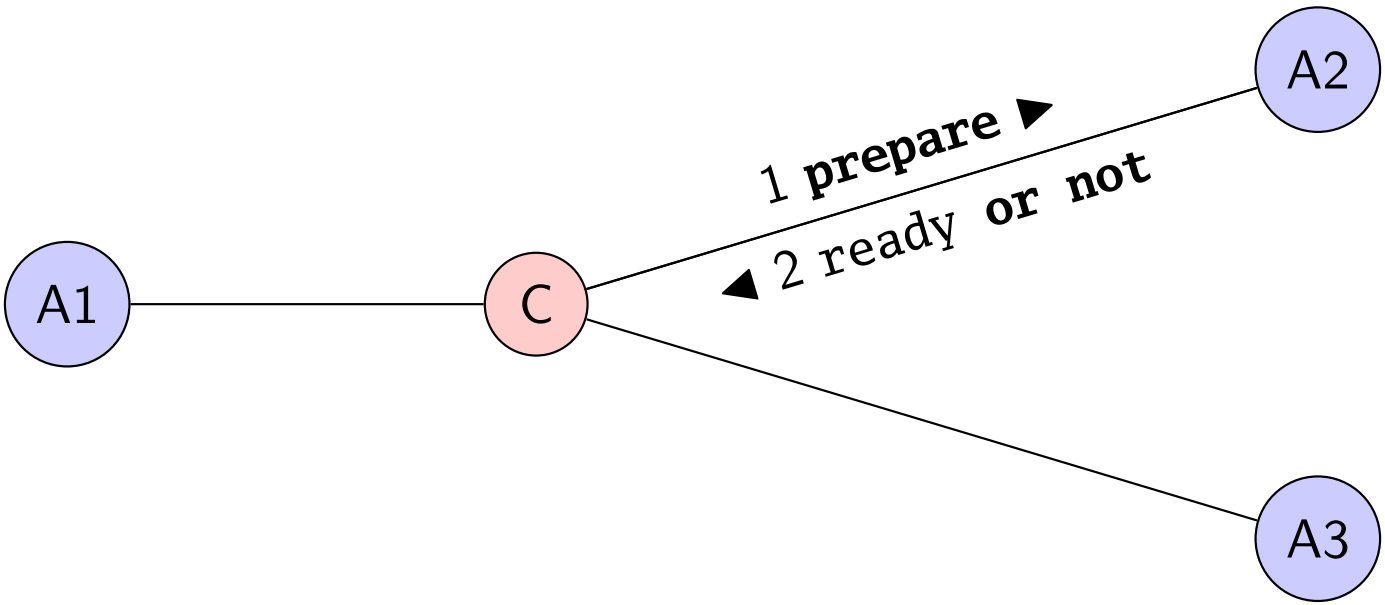
Problème des transactions concernant des bases de données distribuées : comment être sûr que toutes les bases ont effectué les modifications ?

- 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.

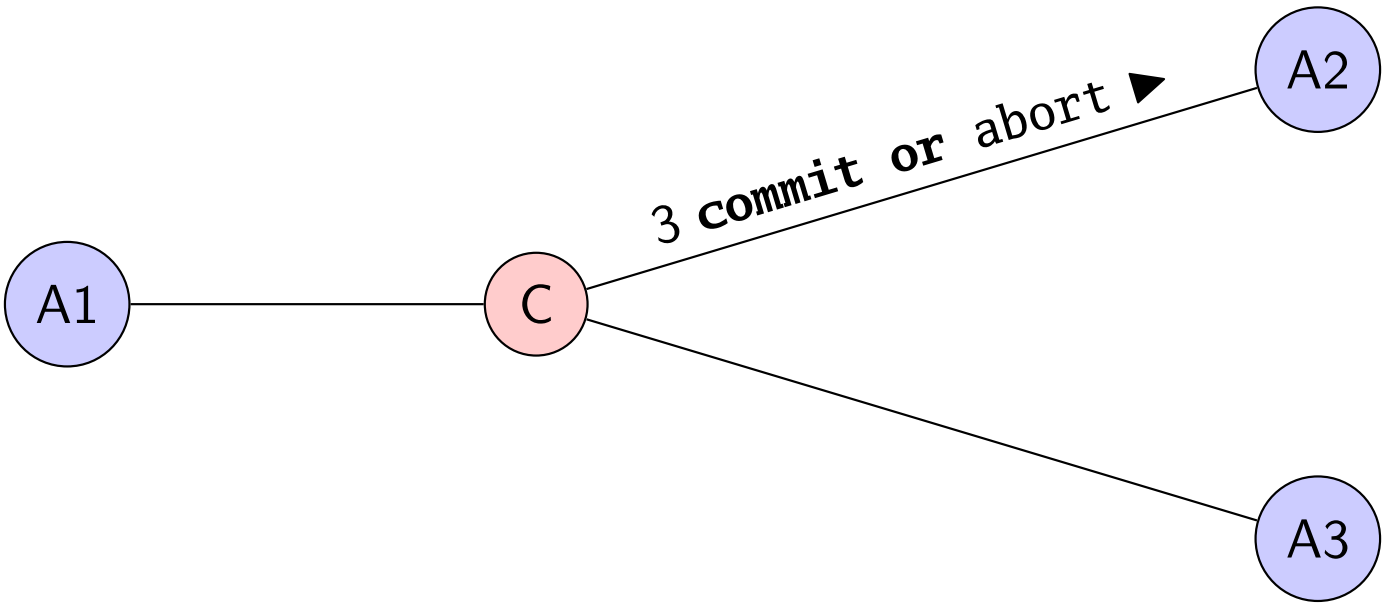
Phase I du commit



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



Plan

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.