SQL
et
Bases de données

Cours 7

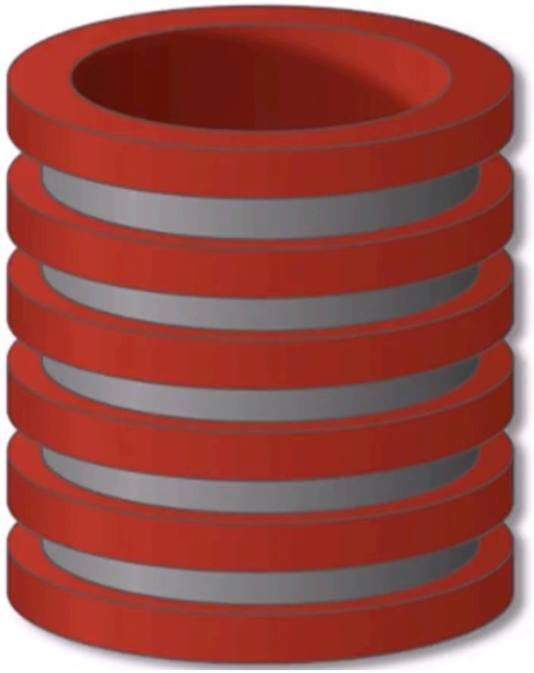
Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/lp-sql

Plan

- exercices (suite)
- niveaux d’isolation
- vues
- nommage des requêtes
- requêtes récursives



- deux bons tutoriels

  http://www.w3schools.com/sql/default.asp

  http://www.programiz.com/sql

client

cID	nom	actif	ville
1	Tom	23000	Bordeaux
2	Jean-Jacques	38000	Paris
3	Martin	51000	Nice
4	Kiki	54000	Pekin
5	Iteki	84000	Tokyo
6	Bob	6100	Nice
7	Albert	12000	Bordeaux
8	Manu	8150	Paris
9	Valou	10300	Bordeaux
10	Joe	32500	Nice
11	Helmut	8150	Paris
12	Martine	11200	Bordeaux
13	Marina	9150	Nice
14	Masha	10290	Nice
15	Julia	32000	Paris
17	Bob	38000	Nice

produit

pID	pNom	pCat	pVille	prix
1	clio	auto	Paris	13000
2	audi	auto	Paris	45000
3	tesla	auto	Pekin	70000
4	tesla	auto	Nice	40000
5	yamaha	moto	Tokyo	8000
6	kawasaki	moto	Tokyo	8000
7	megamo	velo	Paris	3240
8	shimano	velo	Paris	1900
9	btwin	velo	Nice	990
10	triban	velo	Nice	690
11	peugeot	velo	Paris	750
12	bertin	eVelo	Paris	1190
13	trek	eVelo	Bordeaux	1390
14	trek	eVelo	Paris	1350

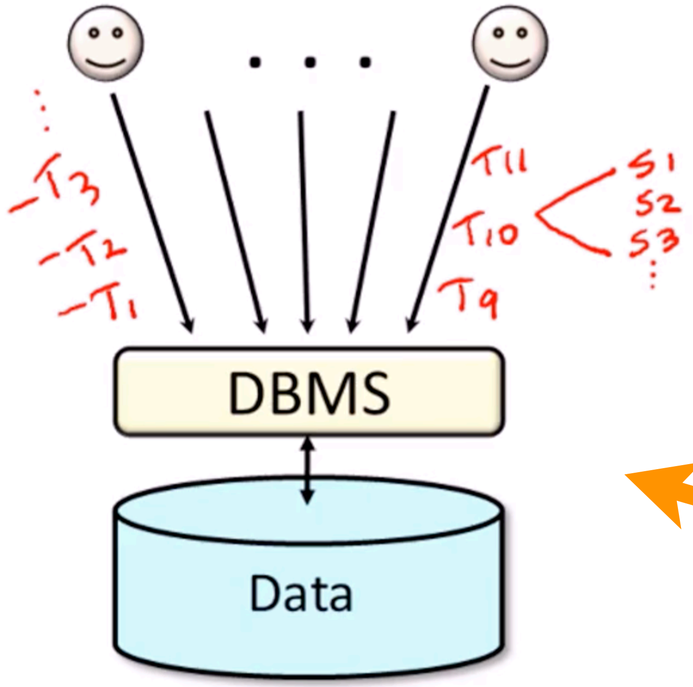
devis

cID	pNom	dCat	commande
2	peugeot	velo	0
7	NULL	velo	0
6	trek	eVelo	0
8	NULL	auto	0
8	NULL	velo	0
8	NULL	eVelo	0
9	honda	auto	0
11	NULL	auto	0
4	honda	moto	0
5	NULL	moto	0
8	triban	velo	0
13	NULL	velo	0
11	yaris	auto	0
12	NULL	velo	0
1	NULL	eVelo	0

Transactions

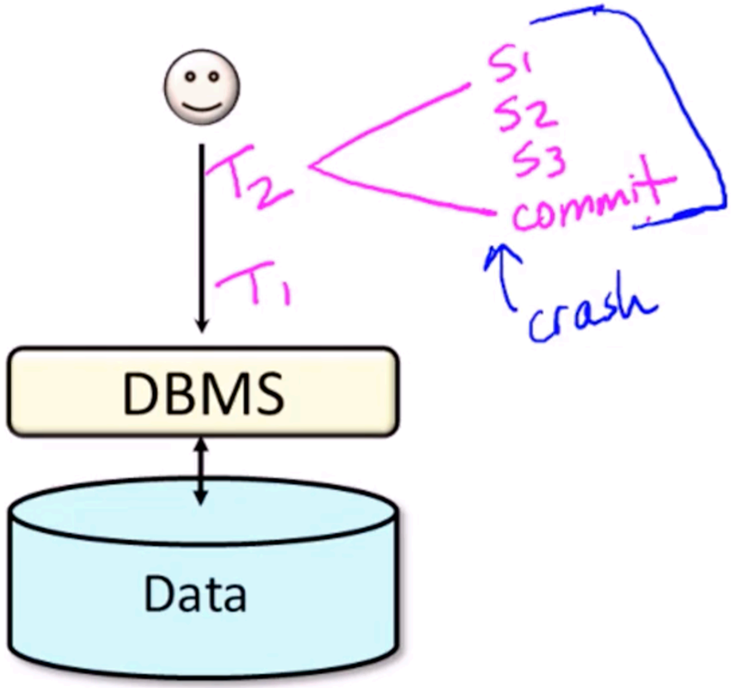
- dans les bases de données, on parle de propriétés `ACID`
    - Atomicité
    - Cohérence
    - Isolation
    - Durabilité

Isolation - sérialisabilité



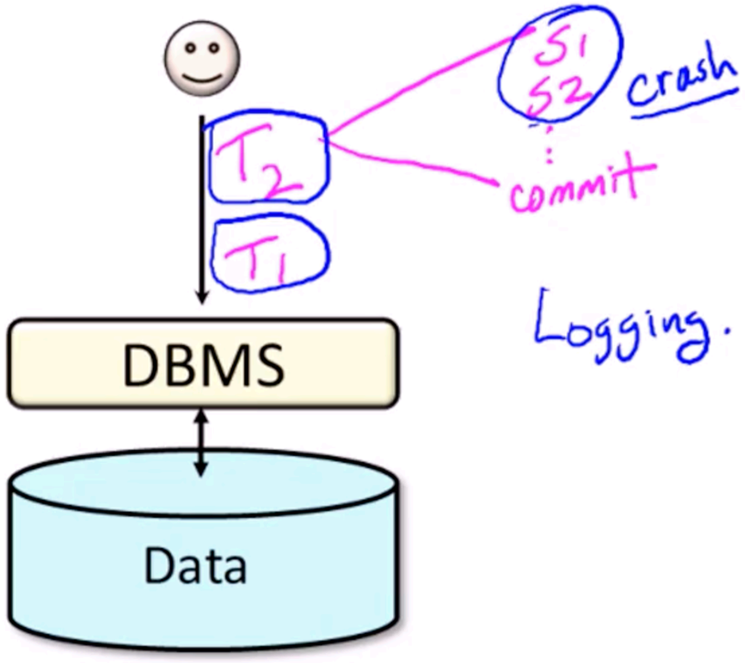
- **isolation**: chacun travaille comme s'il était seul
- **serialisabilité** : les exécutions possibles sont équivalentes à des enchaînements séquentiels de toutes les tâches

Durabilité



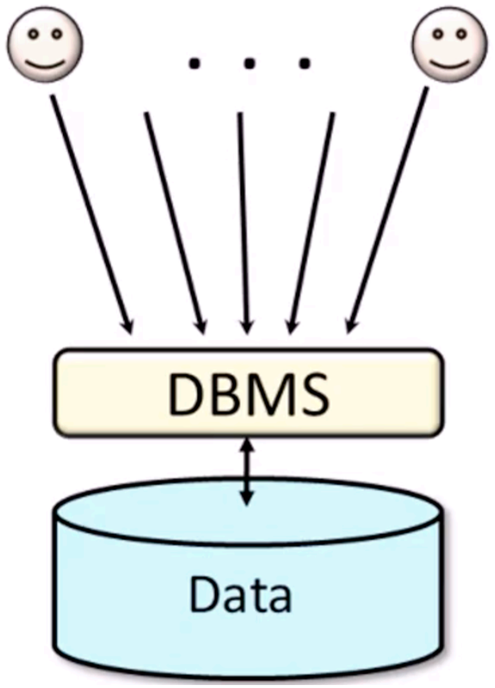
- si crash après `COMMIT`, toutes les modifications persistent
- cette tolérance aux pannes est difficile à implémenter (journalisation, *logging*)

Atomicité - *rollback - abort*



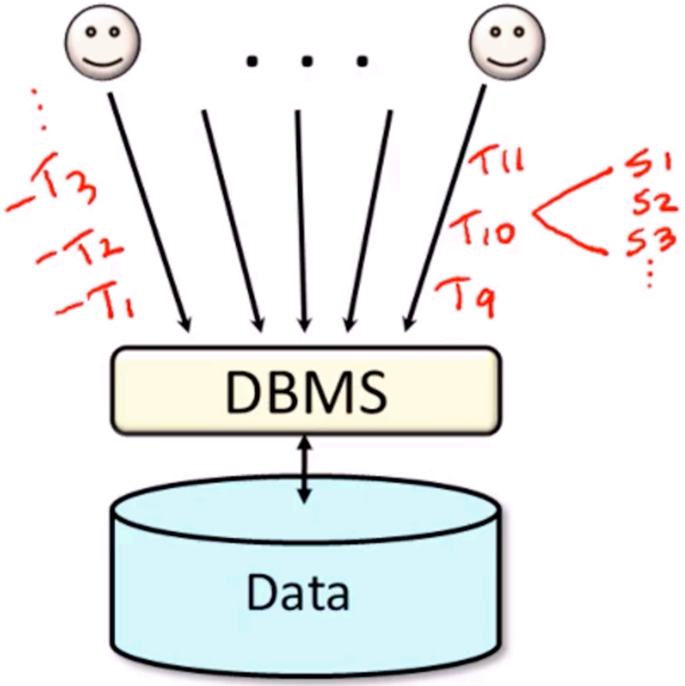
- si crash avant `COMMIT`, toutes les modifications partielles ne sont pas effectuées
- toute transaction signale donc si elle a été complètement effectuer

Cohérence



- supposonsi une contrainte garantie avant et après toute transaction
- la sériabilisité garantit qu'après une séquence de transactions, cette contrainte est maintenue
- donc une contrainte respectée par un utilisateur restera vraie pour plusieurs utilisateurs

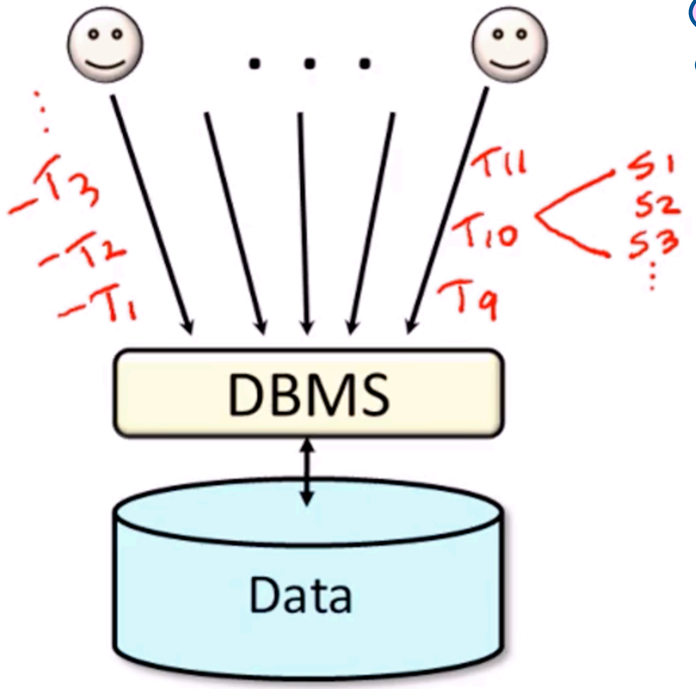
Niveaux d’isolation



- niveaux d’isolation
  - `read uncommitted`
  - `read committed`
  - `repeatable read`
  - `serialisable` (defaut)

Niveaux d’isolation

ma transaction est ‘repeatable read’



ma transaction est ‘read uncommitted’

- chaque transaction a son niveau d’isolation
- 2 utilisateurs d’une même BD peuvent avoir des niveaux différents

Dirty Reads

- lecture d'une donnée modifiée mais non encore validée par un *commit*

T1

```
begin transaction;
. . .
update client set actif = actif + 10000
where nom = 'Jean-Jacques';
. . .
commit;
```

en concurrence avec

T2

```
begin transaction;
. . .
select avg(actif) from client;
. . .
commit;
```

Niveau d'isolation: *Read Uncommitted*

• lecture d'une donnée modifiée mais non encore validée est autorisée

T1

```
begin transaction;
. . .
update client set actif = actif + 10000  where nom = 'Jean-Jacques';
. . .
commit;
```

• exécution non sérialisable.
• le résultat n'est ni équivalent à ***T1 ; T2*** , ni à ***T2 ; T1***

en concurrence avec

T2

```
begin transaction;
set transaction isolation level read uncommitted;
. . .
select avg(actif) from client;
. . .
commit;
```

• sérialisabilité pas importante dans certains cas (par exemple, si on accepte une imprécision sur la moyenne)

• alors implémentation efficace des transactions

Niveau d'isolation: *Read Committed*

• lecture d'une donnée modifiée mais non encore validée n'est pas autorisée

T1

```
begin transaction;
. . .
update client set actif = actif + 10000  where nom = 'Jean-Jacques';
. . .
commit;
```

• exécution non sérialisable.
• le résultat n'est ni équivalent à ***T1 ; T2*** , ni à ***T2 ; T1***

en concurrence avec

T2

```
begin transaction;
set transaction isolation level read committed;
. . .
select avg(actif) from client;
select max(actif) from client;

. . .
commit;
```

• alors implémentation plus efficace qu'avec des transactions sérialisables

• un peu plus de cohérence, car pas de lectures sales

Niveau d'isolation: *Repeatable Read*

• lecture d'une donnée est autorisée de multiples fois si elle ne change pas de valeur

T1

```
begin transaction;
. . .
update client set actif = actif + 10000  where nom = 'Jean-Jacques';
update client set ville = 'Nice'  where nom = 'Jean-Jacques';
. . .
commit;
```

• exécution non sérialisable.
• le résultat n'est ni équivalent à ***T1 ; T2*** , ni à ***T2 ; T1***

en concurrence avec

T2

```
begin transaction;
set transaction isolation level repeatable read;
. . .
select avg(actif) from client;
select nom, ville from client;
. . .
commit;
```

• alors implémentation plus efficace qu'avec des transactions sérialisables

• encore plus de cohérence, mais pas sur des attributs séparés

Niveau d'isolation: *Repeatable Read*

• problème avec les valeurs fantômes

T1

```
begin transaction;
. . .
insert into client [ 100 nouveaux n-uplets ];
. . .
commit;
```

en concurrence avec

T2

```
begin transaction;
set transaction isolation level repeatable read;
. . .
select avg(actif) from client;
select nom, ville from client;
. . .
commit;
```

• exécution non sérialisable.
• le résultat n'est ni équivalent à ***T1 ; T2*** , ni à ***T2 ; T1***

• les valeurs ne changent pas, mais la relation change !

• les nouveaux n-uplets ne seront pas pris en compte

Niveau d'isolation: *Read Only*

- une transaction peut être déclarée en lecture seule

```
begin transaction;
set transaction read only;
set transaction isolation level repeatable read;
. . .
select avg(actif) from client;
select nom, ville from client;
. . .
commit;
```

- alors implémentation plus efficace

Niveaux d’isolation

	dirty reads	nonrepeatable reads	fantômes
`Read Uncommitted`	☹	☹	☹
`Read Committed`	☺	☹	☹
`Repeatable Read`	☺	☺	☹
`Seriazable`	☺	☺	☺

- par défaut: seriazable (sqlite)
- par défaut: repeatable read (Oracle, MySql)

Vues

- une vue crée une entité logique (pas une nouvelle table)

```
create view clientParis as
select cID, nom, actif
from client
where ville = 'Paris';

select * from clientParis;
```

- on se sert des vues comme des tables
- requêtes plus naturelles et plus modulaires
- quelques données peuvent être cachées pour certains utilisateurs

les vues sont très utilisées dans les vraies applications

Vues

- les requêtes sur des vues sont réécrites par le système de BD en requêtes vers les vraies tables

```
create view clientParis as
select cID, nom, actif
from client
where ville = 'Paris';

select * from clientParis;
```

```
select * from clientParis;
```


```
select cID, nom, actif
from client
where ville = 'Paris';
```

- les vues ne sont pas des tables

- mais problèmes pour les requêtes de modification

  [on verra plus tard]

les vues sont très utilisées dans les vraies applications

Nommer une requête avec WITH

- on peut déclarer localement une requête

```
with clientParis
 as (select CID, nom, actif
 from client
 where ville = 'Paris')
select * from clientParis;
```

- ce nom est local à la requête en cours
- c'est donc différent d'une vue dont le nom est permanent

```
with clientParisDevis
 as (select client.cID, nom, actif, dCat
 from client, devis
 where ville = 'Paris'
 and client.cID = devis.cID)
select * from clientParisDevis;
```

3 manières de nommer

- localement avec `WITH`

```
with clientParis
 as (select CID, nom, actif
 from client
 where ville = 'Paris')
select * from clientParis;
```

- localement avec `AS`

```
select * from
 (select CID, nom, actif
 from client
 where ville = 'Paris')
as clientParis;
```

- globalement avec `VUE`

```
create view clientParis as
select cID, nom, actif
from client
where ville = 'Paris';

select * from clientParis;
```

pour plus de sécurité

```
create view if not exists clientParis as
select cID, nom, actif
. . .
```

Requête récursive

- localement avec `WITH`, on peut nommer des requêtes

```
WITH R1 AS (requête-1),
     R2 AS (requête-2),
     …
     Rn AS (requête-n)
requête avec R1, R2, …, Rn (et autres tables);
```

- et aussi faire des requêtes récursives

```
WITH RECURSIVE
     R1 AS (requête-1),
     R2 AS (requête-2),
     …
     Rn AS (requête-n)
requête avec R1, R2, …, Rn (et autres tables);
```

R1, R2, .. Rn aussi possibles dans leurs définitions

Requête récursive

- exemple 1: relation « ancêtre de » dans une famille
- exemple 2: relation « supérieur de » dans une entreprise
- exemple 3: relation « meilleur chemin » dans un déplacement
- ces exemples font intervenir la notion de fermeture transitive inexprimable sans récursivité

Requête récursive

- exemple 1: relation « ancêtre de » dans une famille

```
create table "parentDe" (
[parent] text,
[enfant] text
);
```

- relation « grand-père »

```
select P1.parent, P2.enfant
from parentDe as P1, parentDe as P2
where P1.enfant = P2.parent;
```

- relation « arrière-grand-père »

```
select P1.parent, P3.enfant
from parentDe as P1, parentDe as P2, parentDe as P3
where P1.enfant = P2.parent
and P2.enfant = P3.parent;
```

Requête récursive

- relation « arrière-grand-père »

```
with grandParentDe (a, d)
 as (select P1.parent as a, P2.enfant as d
     from parentDe as P1, parentDe as P2
     where P1.enfant = P2.parent)
select a, enfant from grandParentDe, parentDe
where d = parentDe.parent;
```

- relation « ancètre »

```
with recursive ancetreDe (a, d)
 as (select parent as a, enfant as d from parentDe
     union
     select ancetreDe.a, parentDe.enfant
     from ancetreDe, parentDe
     where ancetreDe.d = parentDe.parent)
select a from ancetreDe
where d = 'Jean-Jacques';
```

Requête récursive

Exercice 2

- dans l'exemple 2: on crée 3 relations « employe », « chefDe », « projet »
  et on chercher à calculer la somme des salaires dans un projet

Exercice 3

- dans exemple 3: relation « vol » avec 4 attributs « départ », « destination », « compagnie », « coût »
  et on cherche le trajet le moins cher entre 2 villes

Prochains cours

- vues (suite)
- interface avec Python, HTML et Javascript
- droits d’accès
- correspondance avec la logique du 1er ordre
- dépendances et formes normales