SQL
et
Bases de données

Cours 6

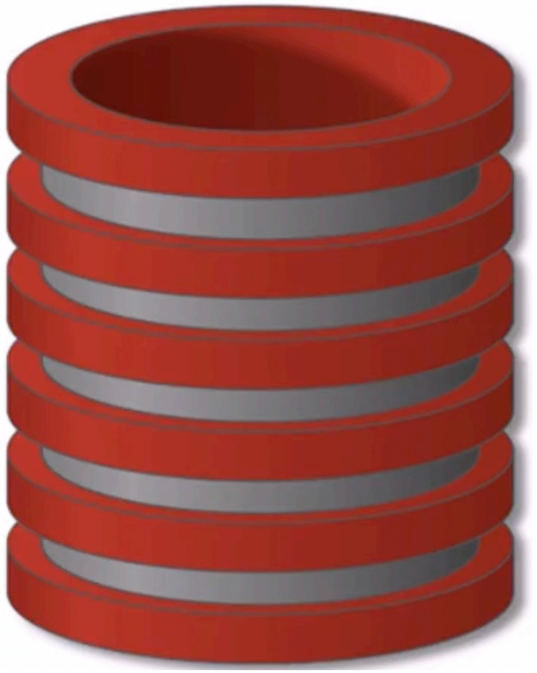
Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/lp-sql

Plan

- exercices (suite)
- concurrence
- tolérance aux pannes
- transactions
- propriétés des transactions
- niveaux d’isolation



- 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

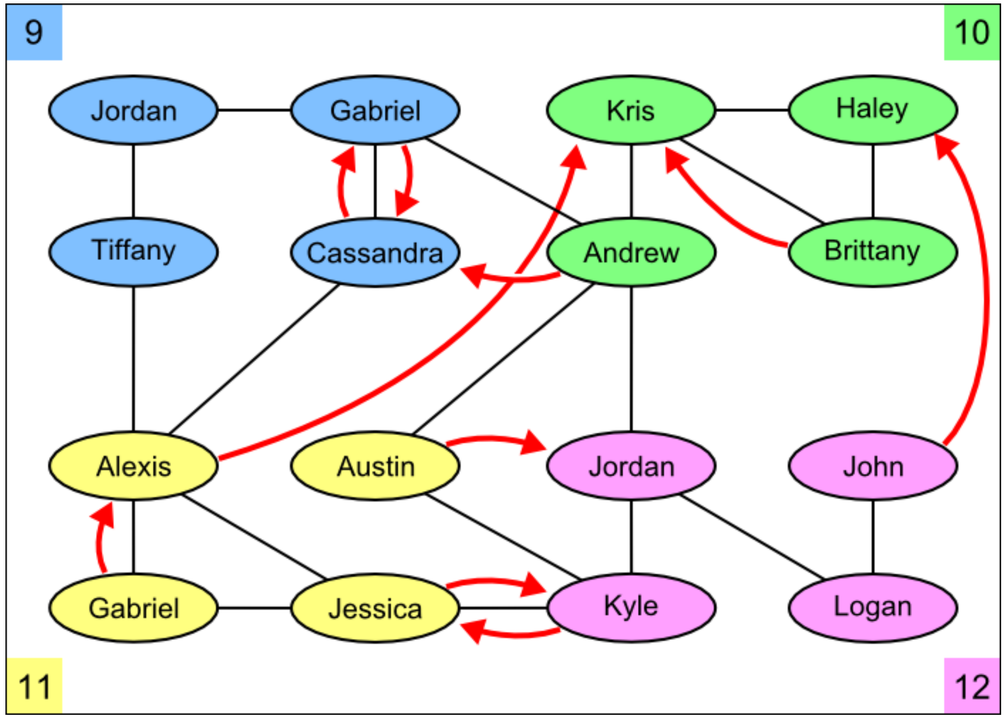
devis

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

Exercice

```
drop table if exists Ecolier;
drop table if exists estAmiDe;
drop table if exists Aime;

create table Ecolier(ID int, nom text, age int);
create table estAmiDe(ID1 int, ID2 int);
create table Aime(ID1 int, ID2 int);
```




```
insert into Ecolier values (1510, 'Jordan', 9);
insert into Ecolier values (1689, 'Gabriel', 9);
insert into Ecolier values (1381, 'Tiffany', 9);
insert into Ecolier values (1709, 'Cassandra', 9);
insert into Ecolier values (1101, 'Haley', 10);
insert into Ecolier values (1782, 'Andrew', 10);
insert into Ecolier values (1468, 'Kris', 10);
insert into Ecolier values (1641, 'Brittany', 10);
insert into Ecolier values (1247, 'Alexis', 11);
insert into Ecolier values (1316, 'Austin', 11);
insert into Ecolier values (1911, 'Gabriel', 11);
insert into Ecolier values (1501, 'Jessica', 11);
insert into Ecolier values (1304, 'Jordan', 12);
insert into Ecolier values (1025, 'John', 12);
insert into Ecolier values (1934, 'Kyle', 12);
insert into Ecolier values (1661, 'Logan', 12);
```

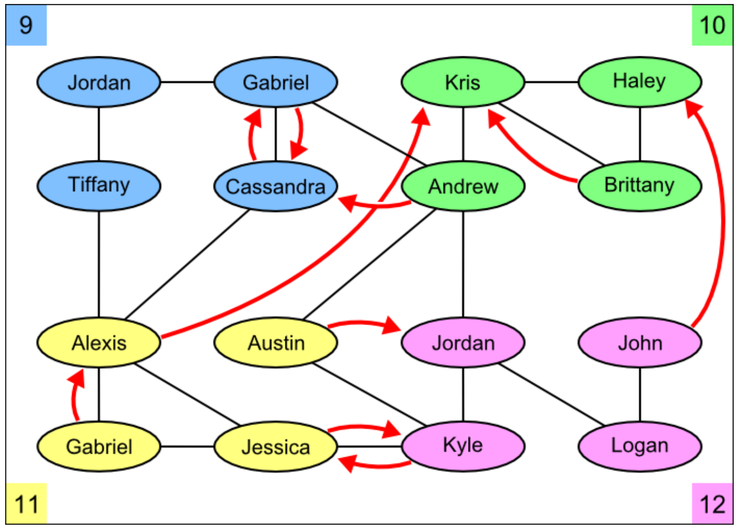
```
insert into estAmiDe values (1510, 1381);
insert into estAmiDe values (1510, 1689);
insert into estAmiDe values (1689, 1709);
insert into estAmiDe values (1381, 1247);
insert into estAmiDe values (1709, 1247);
insert into estAmiDe values (1689, 1782);
insert into estAmiDe values (1782, 1468);
insert into estAmiDe values (1782, 1316);
insert into estAmiDe values (1782, 1304);
insert into estAmiDe values (1468, 1101);
insert into estAmiDe values (1468, 1641);
insert into estAmiDe values (1101, 1641);
insert into estAmiDe values (1247, 1911);
insert into estAmiDe values (1247, 1501);
insert into estAmiDe values (1911, 1501);
insert into estAmiDe values (1501, 1934);
insert into estAmiDe values (1316, 1934);
insert into estAmiDe values (1934, 1304);
insert into estAmiDe values (1304, 1661);
insert into estAmiDe values (1661, 1025);
insert into estAmiDe select ID2, ID1
                        from estAmiDe;
```

```
insert into Aime values(1689, 1709);
insert into Aime values(1709, 1689);
insert into Aime values(1782, 1709);
insert into Aime values(1911, 1247);
insert into Aime values(1247, 1468);
insert into Aime values(1641, 1468);
insert into Aime values(1316, 1304);
insert into Aime values(1501, 1934);
insert into Aime values(1934, 1501);
insert into Aime values(1025, 1101);
```

on peut la charger en http://jeanjacqueslevy.net/lp-sql/ecole.db

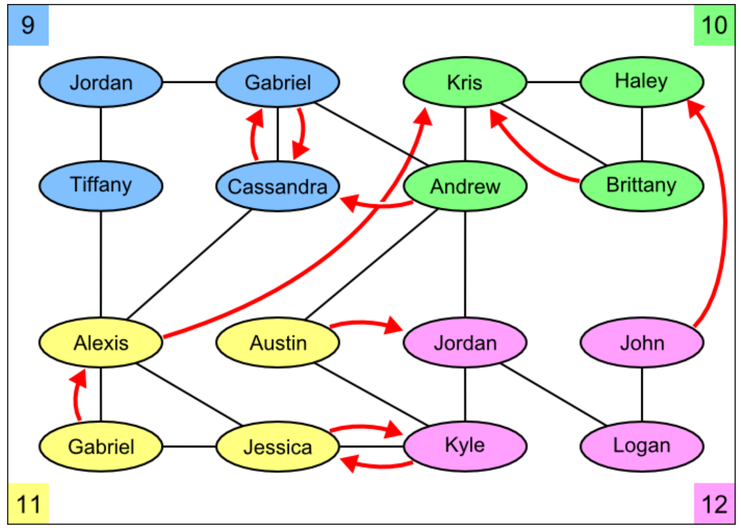
Exercices

- **Q1** Trouver les amis de Gabriel.
- **Q2** Trouver les noms des écoliers qui aiment un écolier de 2 ans ou plus jeune qu'eux.
- **Q3** Imprimer le nom et l'âge de toute paire d'écoliers qui s'aiment mutuellement.
- **Q4** Imprimer (trié par âge, puis par nom) les écoliers qui n'apparaissent pas dans la table Aime.
- **Q5** Imprimer les paires d'écoliers A et B tels que A aime B dont on ne sait rien sur ceux qu'aime B.
- **Q6** Imprimer les noms et âges de ceux qui n'ont pas d'amis du même âge. Imprimer les triés par âge
- **Q7** Pour tout écolier A qui aime un écolier B mais qui ne sont pas amis, trouver un ami commun C (qui pourrait les mettre en relation). Imprimer les noms de A, B, C et leurs âges.
- **Q8** Trouver la différence entre le nombre d'écoliers et le nombre de noms différents.
- **Q9** Trouver le nom et l'âge de tous les écoliers qui sont aimés par un autre écolier.



Exercices

- **Q1** Pour tout écolier A qui aime B, mais B aime un écolier différent C, imprimer les noms et âges de A, B, C
- **Q2** Trouver les écoliers dont les amis ont tous des âges différents d'eux.
- **Q3** Quel est l'âge moyen des amis par écolier ?
- **Q4** Trouver le nombre d'écoliers amis de Cassandra ou amis d'amis de Cassandra.
- **Q5** Trouver le nom et l'âge de l'écolier avec le plus grand nombre d'amis.
- **Q6** Supprimer les plus vieux écoliers de la table des écoliers
- **Q7** Supprimer les paires d'écoliers A et B où A aime B mais B n'aime pas A..
- **Q8** Ajouter les paires A et C où A aime B et B aime C.
  (Les amis de mes amis sont mes amis — Attention aux répétitions !)



Concurrence

- problèmes avec les variables partagées

au début → `x = 0`

plusieurs processus en concurrence

processus 1

```
x = x + 1
```

↓

```
lire x; modifier x; écrire x
```

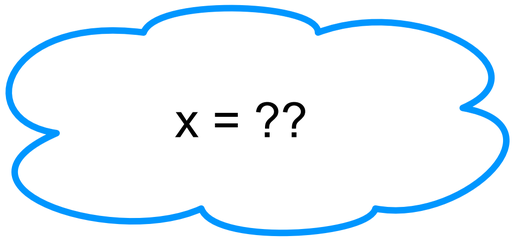
processus 2

```
x = x + 1
```

↓

```
lire x; modifier x; écrire x
```

résultats possibles →



Concurrence

- problèmes avec les variables partagées (niveau machine)

au début → $x = 0$

plusieurs processus en concurrence

processus 1

$R_1 \leftarrow x$

$R_1 \leftarrow R_1 + 1$

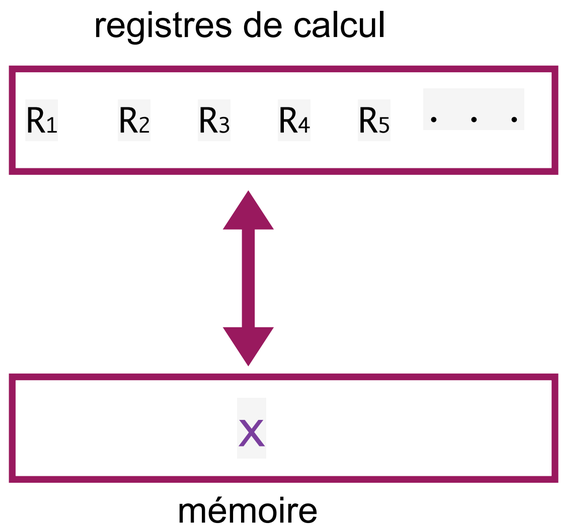
$x \leftarrow R_1$

processus 2

$R_2 \leftarrow x$

$R_2 \leftarrow R_2 + 1$

$x \leftarrow R_2$

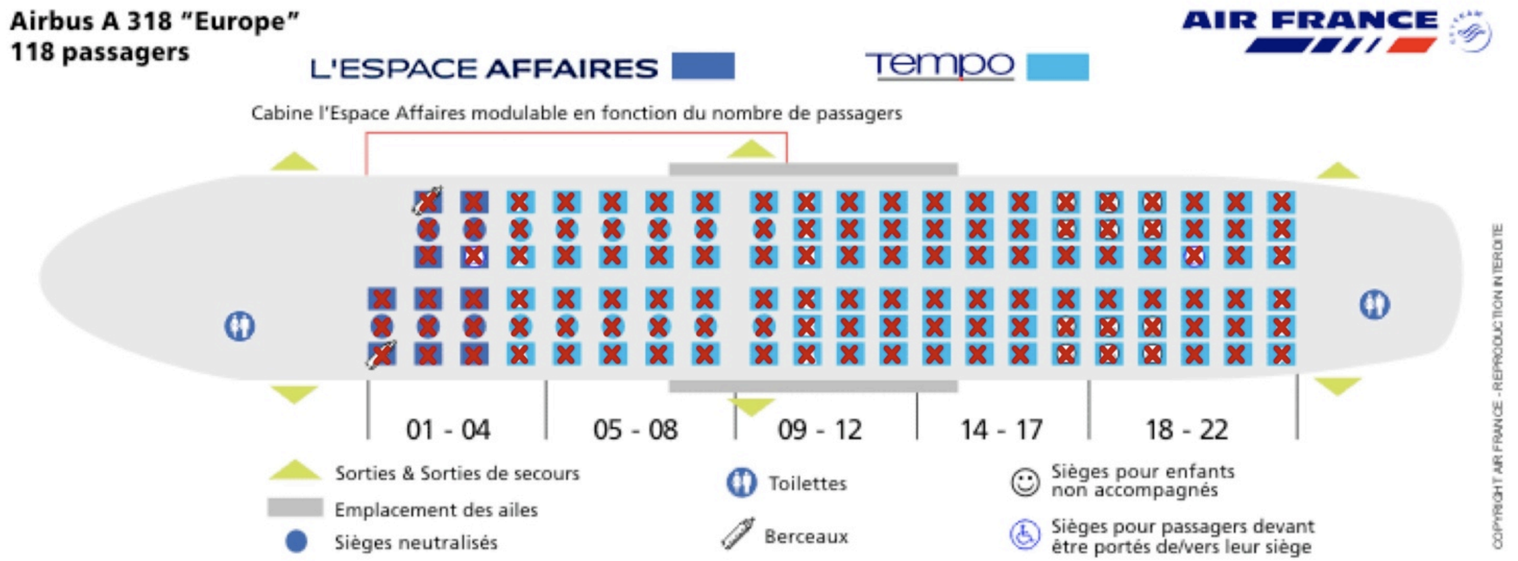


résultats possibles →

x = 2

x = 1

Ressources partagées



Bob

y-a-t'il une place libre?

"oui"

je la prends

y-a-t'il une place libre?

"oui"

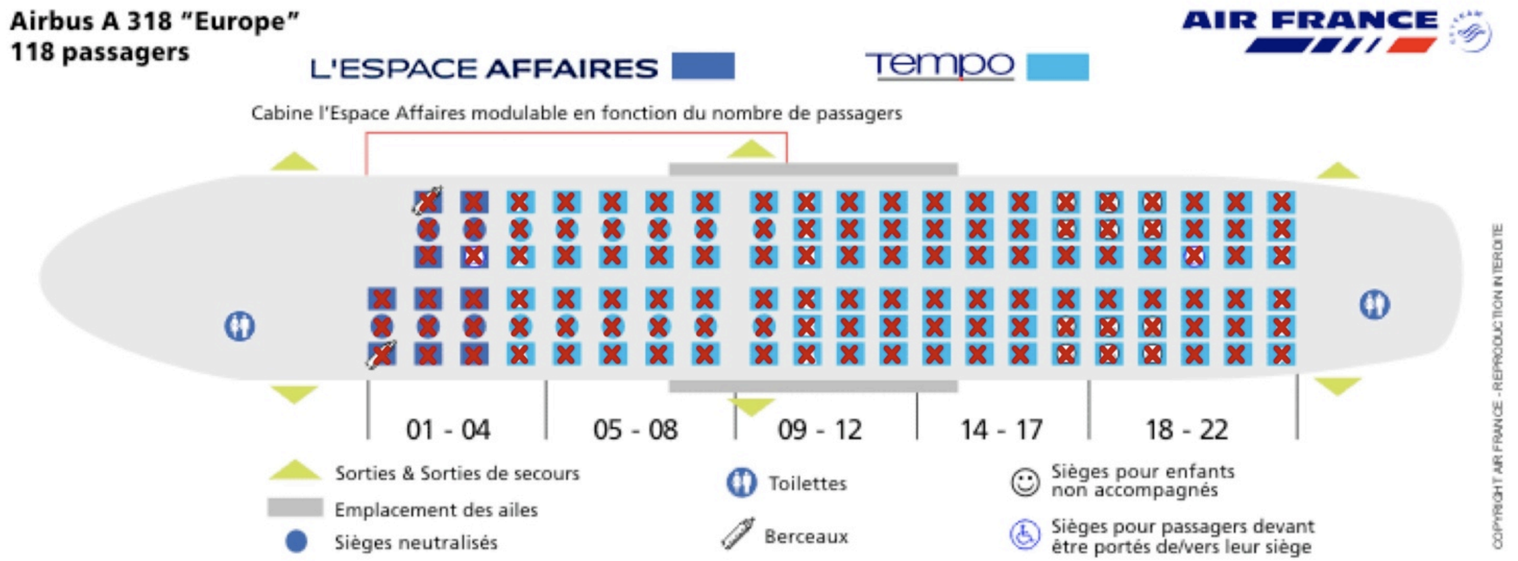
je la prends

Alice

HERCHE OMMUN

INRIA MICROSOFT RESEARCH

Ressources partagées



Bob

y-a-t'il une place libre?

"oui"

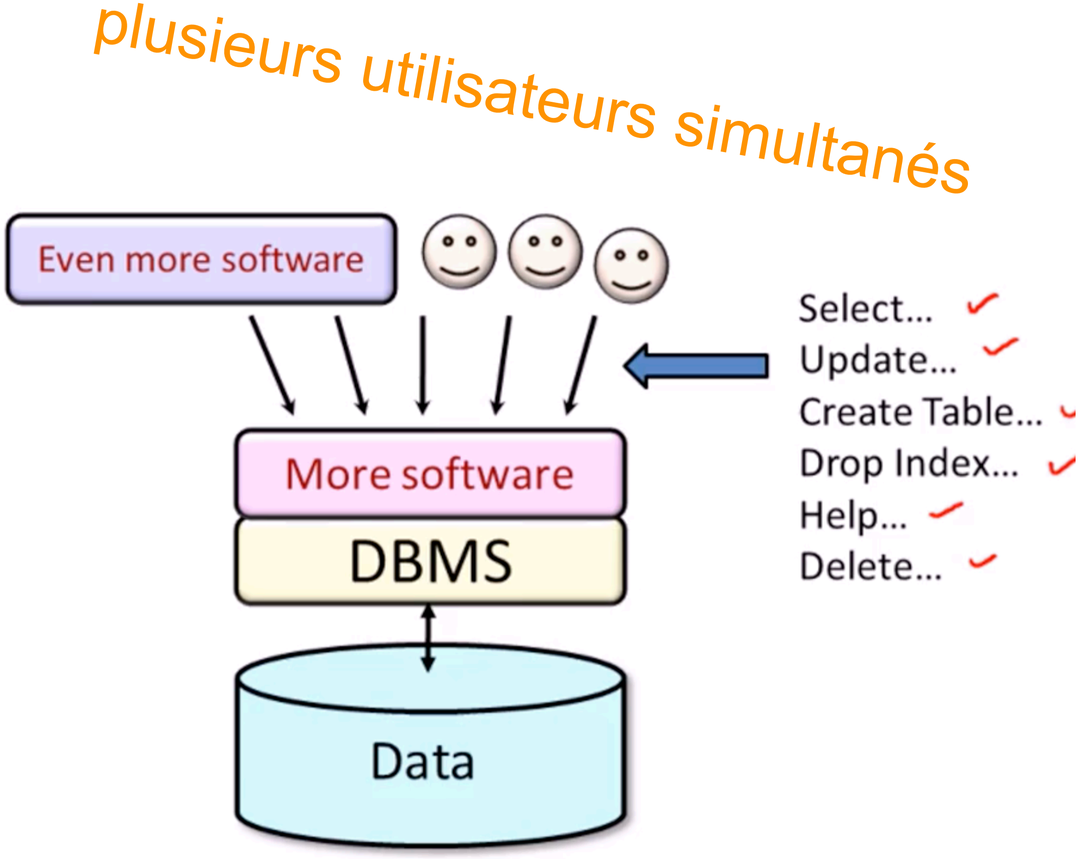
je la prends

y-a-t'il une place libre?
"oui"
je la prends

Alice

Concurrence et SQL

- accès concurrents et pannes (logicielles / matérielles)
- transaction est une ou plusieurs requêtes SQL
- donner l'impression d'être seul utilisateur (isolation)
- une transaction va jusqu'au bout ou ne fait rien du tout



Requêtes concurrentes

utilisateur 1

```
update client set actif = actif + 1000
where client = 'Iteki';
```

en parallèle avec

utilisateur 2

```
update client set actif = actif + 1500
where client = 'Iteki';
```

client

cID	nom	actif	ville
5	Iteki	84000	Tokyo

client.actif [5]

lire; modifier, écrire

84000 + 2500 = 86500

84000 + 1000 = 85000

84000 + 1500 = 85500

résultats possibles

client.actif [5]

lire; modifier, écrire

Requêtes concurrentes

utilisateur 1

```
update devis set dCat = ‘auto’
where cID = 5;
```

client.actif [5]

lire; modifier, écrire

en parallèle avec

devis

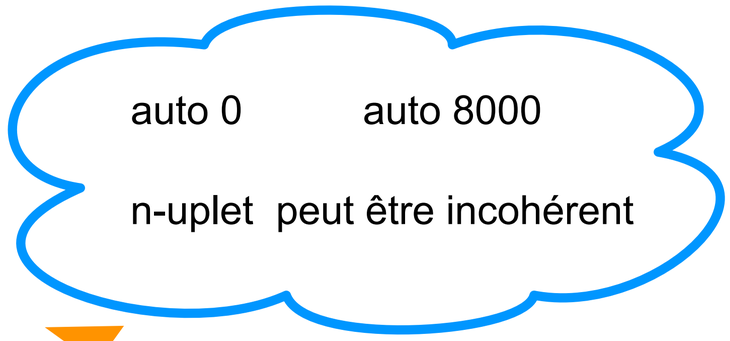
cID	pNom	dCat	commande
5	NULL	moto	0

utilisateur 2

```
update client set commande = 8000
where dCat = ‘moto’;
```

client.actif [5]

lire; modifier, écrire



résultats possibles

Requêtes concurrentes

utilisateur 1

```
update devis set commande = 18000
where dCat = 'auto' and
cID in (select cID from client
  where actif < 20000);
```

devis.commande

lire; modifier, écrire

en parallèle avec

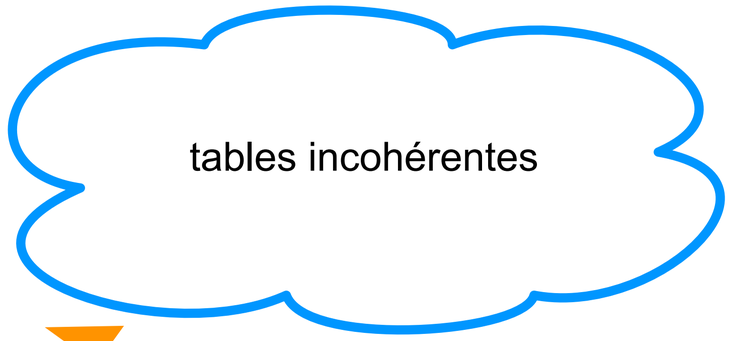
utilisateur 2

```
update client set actif = actif + 10000
where ville = 'Paris';
```

client.actif

lire; modifier, écrire

devis **client**



résultats possibles

Requêtes concurrentes

utilisateur 1

```
insert into archive
  select * from devis where commande = 0;
delete from devis where commande = 0;
```

en parallèle avec

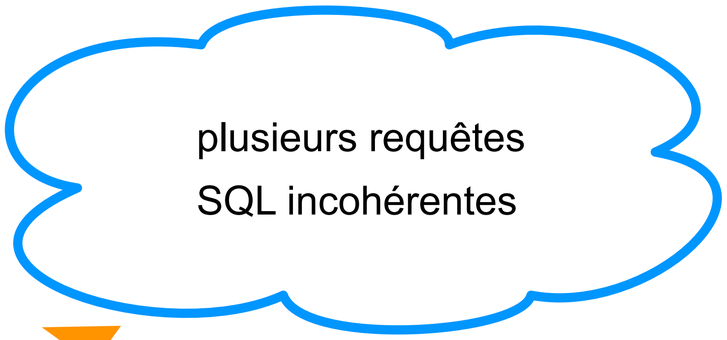
utilisateur 2

```
select count(*) from devis;
select count(*) from archive;
```

devis **archive**

client.actif [5]

`lire; modifier, écrire`



client.actif [5]

`lire; modifier, écrire`

résultats possibles

Requêtes concurrentes

• on veut exécuter les requêtes SQL comme si elles s'exécutaient en isolation complète

• mais on veut les exécuter concurremment autant que possible

- multiprocesseurs
- parallélisme (multi threads)
- I/O asynchrones
- multi-utilisateurs

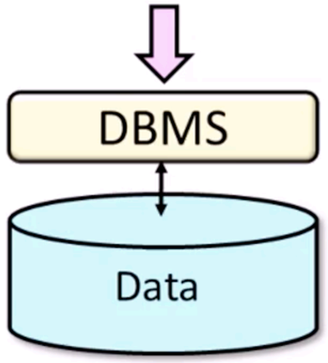
• par exemple on veut les exécuter concurremment si les utilisateurs opèrent sur des parties différentes de la base de données

Tolérance aux pannes

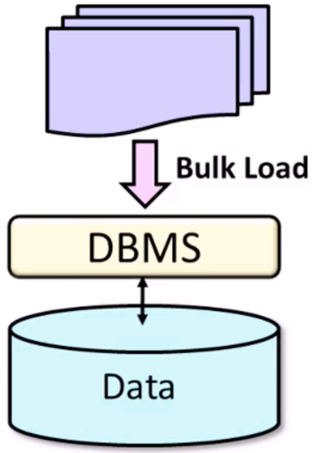
3 exemples de crash

crash possible pendant la modification de données

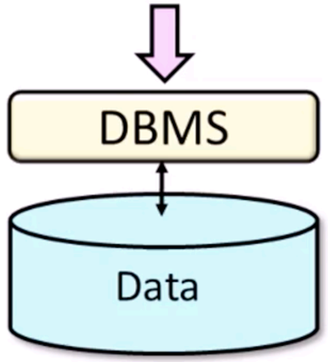
```
insert into archive
  select * from devis where commande = 0;
delete from devis where commande = 0;
```



crash possible pendant le chargement de données externes



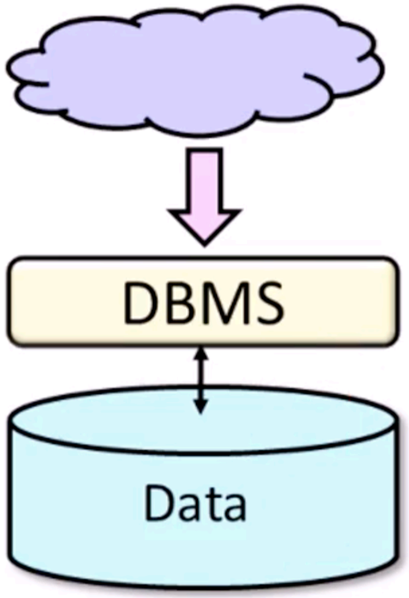
modifications bufferisées en mémoire



SOLUTION ??

Tolérance aux pannes

- le but est d’éviter les requêtes exécutées partiellement
- et donc finir complètement les requêtes ou ne pas les faire du tout.

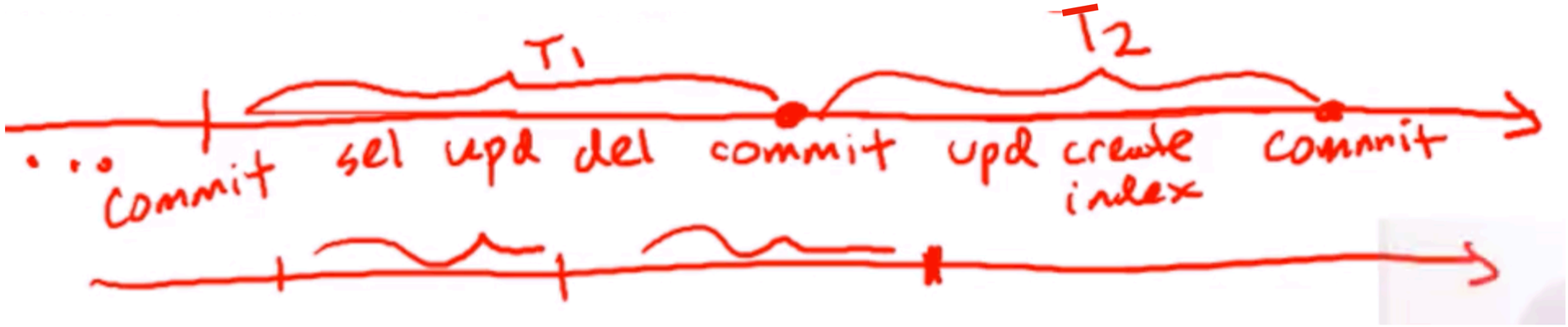


Transactions

- les requêtes concurrentes et les pannes sont traitées de la même manière par des '**transactions**'

- une transaction est une suite de requêtes SQL traitées comme une seule unité SQL

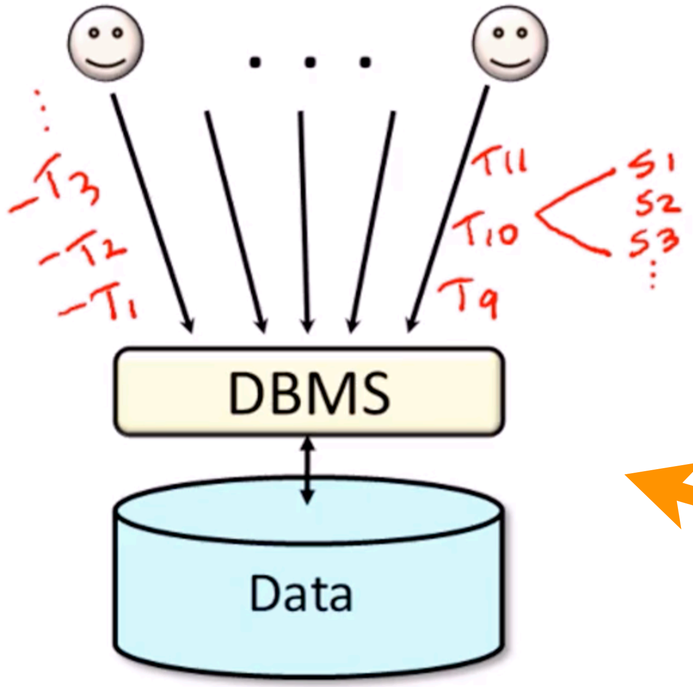
  - les transactions commencent avec `BEGIN TRANSACTION`
  - et finissent avec le mot clé `COMMIT`
  - à la fin d'une session il y a un `COMMIT` implicite
  - un mode `AUTOCOMMIT` transforme toute requête SQL en transaction



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

Isolation - sérialisabilité

T1

```
update client set actif = actif + 1000
where client = 'Iteki';
```

T2

```
update client set actif = actif + 1500
where client = 'Iteki';
```

client

cID	nom	actif	ville
5	Iteki	84000	Tokyo

serialisabilité → seules exécutions possibles

$T_1 ; T_2$ ou $T_2 ; T_1$

résultats possibles →

84000 + 1000 + 1500 = 86500

ou

84000 + 1500 + 1000 = 86500

Isolation - sérialisabilité

T_1

```
update devis set commande = 18000
where dCat = 'auto' and
cID in (select cID from client
  where actif < 20000);
```

T_2

```
update client set actif = actif + 10000
where ville = 'Paris';
```

serialisabilité →

seules exécutions possibles

$T_1 ; T_2$ ou $T_2 ; T_1$

ne garantit pas l'ordre de l'exécution

devis **client**

résultats possibles →

plusieurs résultats possibles

Isolation - sérialisabilité

T_1

```
insert into archive
  select * from devis where commande = 0;
delete from devis where commande = 0;
```

T_2

```
select count(*) from devis;
select count(*) from archive;
```

serialisabilité →

seules exécutions possibles

$T_1 ; T_2$ ou $T_2 ; T_1$

on détruit et on compte

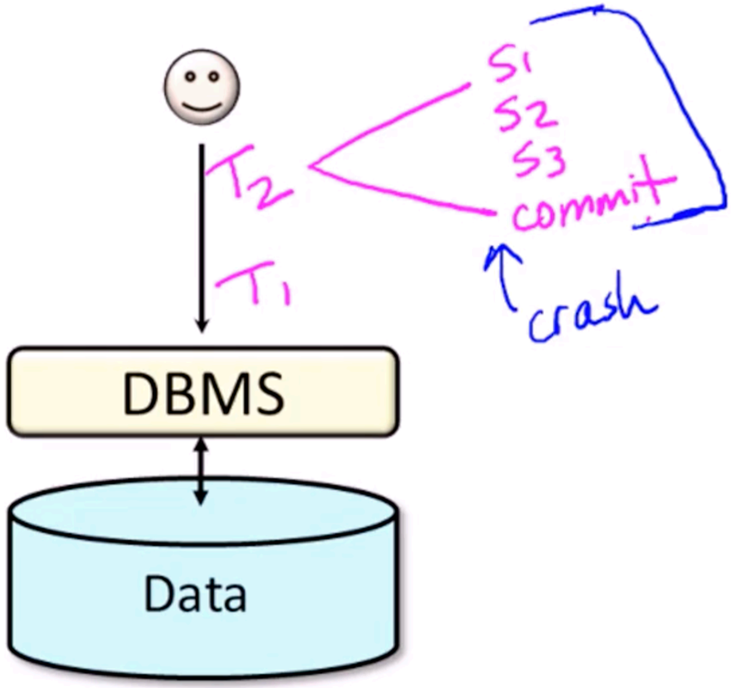
on compte et on détruit

devis **archive**

résultats possibles →

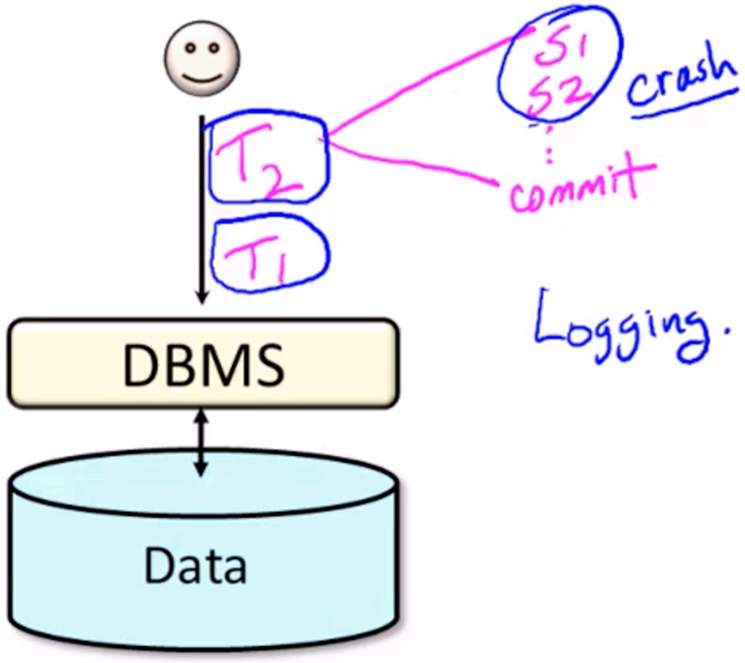
plusieurs requêtes SQL incohérentes

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

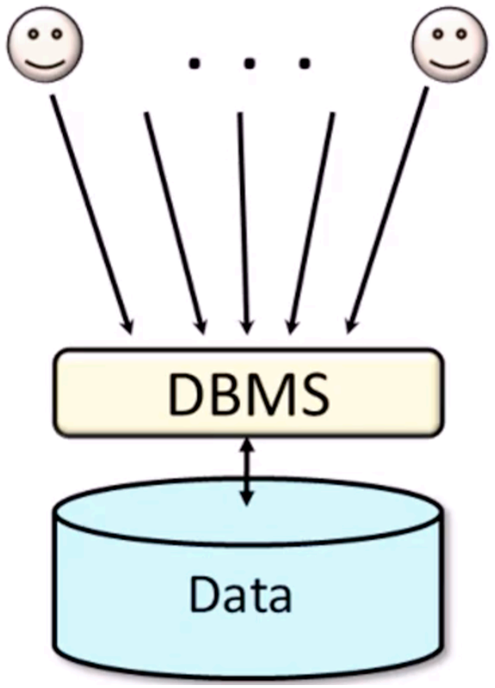
Atomicité - *rollback - abort*

- abandon volontaire d'une séquence partielle de requêtes SQL
  (le système peut aussi annuler cette séquence)

```
Begin Transaction;
<get input from user>
SQL commands based on input
<confirm results with user>
If ans='ok' Then Commit; Else Rollback;
```

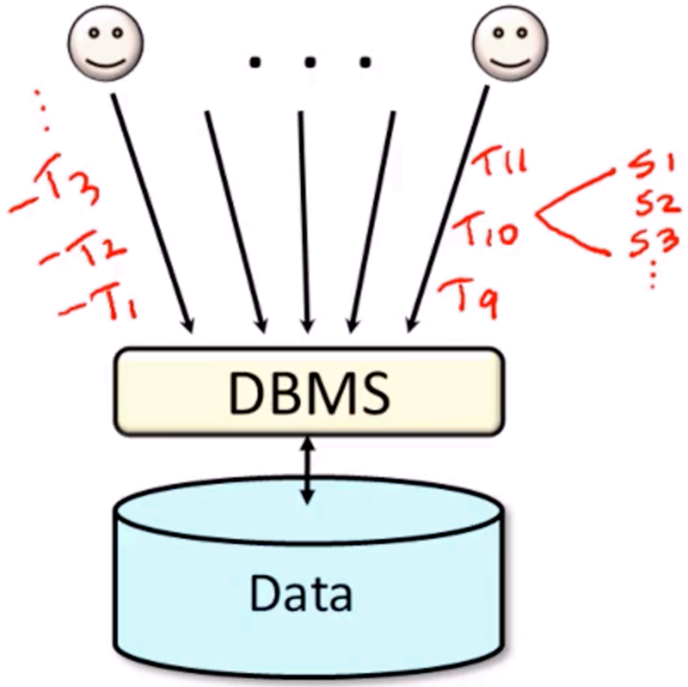
- pour garantir l'atomicité, un système de verrou (locking) est positionné au début de transaction
- selon le résultat de l'entrée (input), on finit ou abandonne la transaction
- alors toutes les variables de la BD doivent reprendre leurs valeurs initiales
- attention: les variables externes ne sont pas réinitialisées (distribution d'argent, impression, …)

Cohérence



- supposons 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



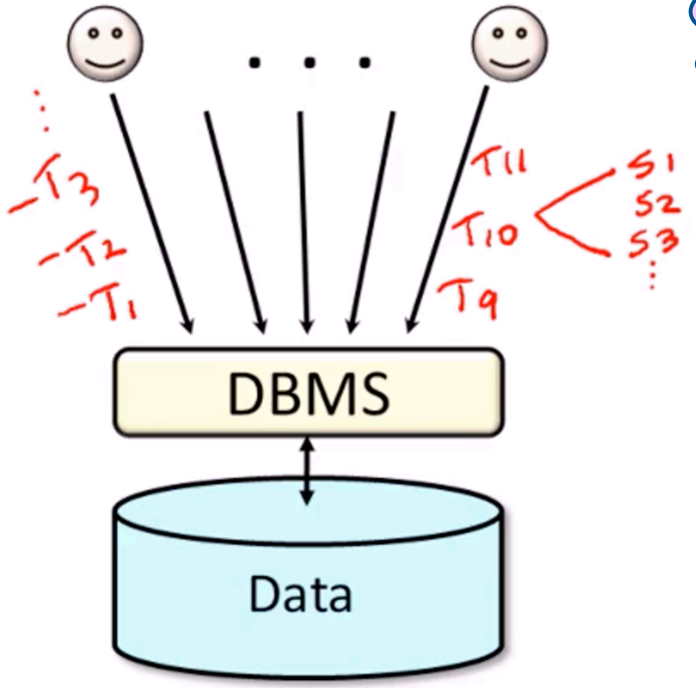
- niveaux d’isolation
  - read uncommitted
  - read committed
  - repeatable read

efficacité + / -

cohérence - / +

Niveaux d’isolation

ma transaction est ‘repeatable read’



ma transaction est ‘read uncommitted’

- chaque transaction a son niveau d’isolation

Prochains cours

- isolation (suite)
- vues
- interface avec Python, HTML et Javascript
- correspondance avec la logique du 1er ordre