Notes du cours de persistance de données III

Nathan Furnal

13 septembre 2021

Table des matières

Liste des définitions

1 Introduction

Dans ce cours nous verrons plusieurs manières de manipuler les données avec SQL, la structure sera la suivante :

DML

Data manipulation language, on se concentre sur la **manipulation** de données notamment via les instructions `INSERT`, `SELECT`, `UPDATE`, `DELETE`,... Aussi appelées "CRUD" (Create, Read, Update, Delete).

DDL

Data definition language, on se concentre sur la création, suppression ou modification d'objets. Notamment, via les instructions `CREATE`, `DROP`, `ALTER`. On peut appliquer ces opération sur des objets comme des `TABLE`, `VIEW`, `INDEX`, `SEQUENCE`, `SYNONYM`, etc.

DCL

Data control language, ensemble d'instructions liées à la sécurité. Notamment, `GRANT`, `REVOKE`, `COMMIT`, `ROLLBACK`.

Index

Structure de données particulière utilisée dans les systèmes de gestion de base données pour accélérer certaines requêtes avec pour contrepartie un coût en mémoire. L'implémentation classique dans les bases de données est le B-tree.

PL/SQL

Langage propriétaire de l'entreprise Oracle qui étend le SQL classique avec des **blocs anonymes**, des **procédures**, des **fonctions** et des **déclencheurs** qui permettent d'exécuter du code de manière événementielle quand une condition est respectée. Ces additions permettent d'optimiser et/ou faciliter le SQL classique.

2 DML (Data manipulation language)

Le SQL (Structured Query Language) est un langage non procédural créé dans les années 1980 dont le développement commence IBM et dont les versions les plus utilisées sont SQL2 (1992) et SQL3 (1999).

De manière générale, un `SELECT` peut s'écrire comme l'exemple suivant, les `[]` dénotent une expression optionnelle.

Les généralités et différences entre dialectes de SQL sont repris dans les slides de la partie DML.

```
SELECT [DISTINCT]{ * ou {liste d'expressions
et/ou aggregate fonction}}
FROM liste de tables et/ou de vues et/ou de joins
[WHERE condition]
[GROUP BY liste d'attributs]
[HAVING condition]
[ORDER BY liste d'attributs [ASC/DESC]
et/ou n°colonne [ASC/DESC] ]
```

SQL permet la définition de synonymes, c'est-à-dire un alias pour un autre objet.

```
CREATE [PUBLIC] SYNONYM [nomSchema1.]nomSynonym
FOR [nomSchema2.]nomObjet
```

```
DROP [PUBLIC] SYNONYM [nomSchema1.]nomSynonym
```

On peut associer des tables, des vues, d'autres synonymes, etc.

2.1 Sous-requêtes

On peut utiliser des sous-requêtes pour comparer des valeurs ou sélectionner un sous-ensemble de valeurs. On peut aussi les inclure dans des `JOIN`.

Par exemple,

```
select * from employe
where empno in (select empno from employe
                join departement on
                empno = dptmgr);
```

ou encore,

```
select min(subquery.empsal)
from (select empsal from employe
      join departement on
      empno = dptmgr) subquery;
```

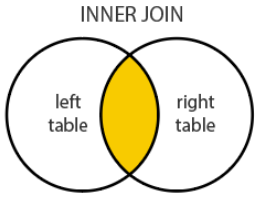
et finalement,

```sql
create or replace view maleManagers
as
select * from employe
      join departement on
      empno = dptmgr
      where empsexe = 'M';

create or replace synonym MM
for maleManagers;

select * from employe e
left join MM
on e.empno = MM.empno
where e.empdpt in (select empdpt from MM);
```

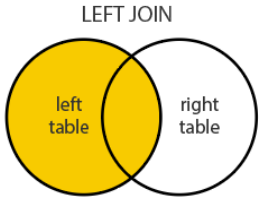
2.2 Jointure externe

Les jointures externes permettent d'intégrer des valeurs additionnelles dans une requête, là où le `JOIN` classique ne permet que d'intégrer les valeurs que l'on retrouve dans les deux tables. S'il y a des doublons lors d'une jointure, elles apparaissent elles aussi plusieurs fois. Par la jointure on va permettre à des tables différentes de joindre des colonnes, si on veut ajouter des lignes et que des requêtes se superposent, il faut utiliser `UNION`.

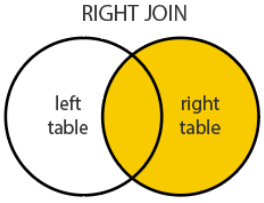
Jointure simple



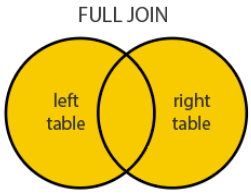
Jointure à gauche



Jointure à droite



Jointure totale



2.3 INSERT - UPDATE - DELETE

Pour modifier une table, on va utiliser les mots clés :

INSERT Permet l'insertion de nouvelles valeurs dans une table tant qu'elles respectent les contraintes d'intégrité imposées par la table.

On peut soit ajouter un tuple

```
INSERT INTO nomRelation [ (liste attributs) ]
       VALUES (liste expressions)
```

Ou bien plusieurs via une sélection

```
INSERT INTO nomRelation [ (liste attributs) ]
       SELECT ...
```

UPDATE Permet de mettre à jour des valeurs existantes d'une table, la mise à jour peut se faire par des fonctions, des SELECT imbriqués ou des attributs de relation. En général on utilisera la clause WHERE pour définir l'endroit de la mise à jour.

Il faut cependant se méfier de ne pas créer des problèmes de concurrence ou de lecture/écriture sale quand on met une table à jour. Ces problèmes sont expliqués dans la section sur les contrôles de données (DCL).

```
UPDATE nomRelation
  SET attribut1 = expression1
  [,attribut2 = expression2] ...
[WHERE condition]
```

DELETE Permet de supprimer des tuples d'une table aussi par SELECT imbriqués ou par sélection via une condition.

```
DELETE FROM nomRelation
[WHERE condition]
```

2.4 SELECT Informations complémentaires

Le complément concerne les slides 46 à 92 de la partie sur les manipulations de données (DML). Ce sont surtout des exemples donc je ne les reprend pas dans les notes entièrement.

2.4.1 Gestion du NULL

- Une expression dont l'évaluation renvoie une valeur numérique, caractères ou temporelle est évaluée à NULL si l'un de ses arguments est NULL.
- Une fonction d'agrégat est calculée en ignorant les NULL (sauf COUNT(*)).
- Lors d'un groupement, les NULL forment un groupe.

De plus, la sélection via un NULL ou l'absence de NULL pose souvent problème.

2.4.2 CASE

Les cas illustrés par CASE WHEN permettent d'obtenir une modification momentanée d'une table ou de sélection via une condition plus complexe. Dans l'expression suivante on créé une colonne temporaire "sexe" qui est une modification de "empsexe".

```
SELECT empno, empnom,
CASE
WHEN empsexe = 'F' THEN 'Féminin'
ELSE 'Masculin'
END AS sexe
FROM employe
```

Un autre exemple plus complexe est celui-ci :

```sql
SELECT d.dptlib
FROM departement d
LEFT JOIN employe e ON d.dptno = e.empdpt
GROUP BY d.dptlib
HAVING sum(CASE WHEN e.empnom LIKE 'M%' THEN 1 ELSE 0 END) >= 1
       AND
       sum(CASE WHEN e.empnom LIKE 'D%' THEN 1 ELSE 0 END) >= 1;
```

2.4.3 Expression de table (Common Table Expression ou CTE)

Une expression de table est une vue locale à la requête, qui s'introduit via le mot-clé `WITH`. Elle permet plus de flexibilité par exemple avec des sous-requêtes utilisées plusieurs fois.

Par exemple ou peut simplifier cette expression :

```sql
SELECT empdpt
FROM employe
GROUP BY empdpt
HAVING COUNT(*) >= ALL(SELECT COUNT(*)
                       FROM employe
                       GROUP BY empdpt)
```

```sql
WITH dptNb (dpt, nbEmp) AS
     (SELECT empdpt, COUNT(*)
      FROM employe
      GROUP BY empdpt)
SELECT dpt, nbEmp FROM dptNb
WHERE nbEmp = (SELECT MAX(nbEmp) FROM dptNb)
```

2.4.4 Récursivité

```sql
-- PostgreSQL only accepts recursion with a keyword

WITH RECURSIVE Q2 (dno, dlib, dpere, niv, nompere) AS

(SELECT dptno, dptlib, dptadm, 0, ''::varchar /*Type casting in PostgreSQL*/
        FROM departement
        WHERE dptadm IS NULL
UNION ALL
SELECT dptno, dptlib, dptadm, Q2.niv+1, Q2.dlib
       FROM departement JOIN Q2
       ON Q2.dno = dptadm )
SELECT * FROM Q2;
```

dno	dlib	dpere	niv	nompere
D21	DIRECTION		0	
A00	DEVELOPPEMENT	D21	1	DIRECTION
E11	VENTES	D21	1	DIRECTION
B01	PRODUCTION	A00	2	DEVELOPPEMENT
C01	MAINTENANCE	A00	2	DEVELOPPEMENT
D11	SUPPORT	E11	2	VENTES
E01	MARKETING	E11	2	VENTES
E21	FORMATION	E11	2	VENTES

2.4.5 `FETCH`

`FECTH` Permet de sélectionner un nombre limité de lignes, ce mot-clé est aussi utilisé dans le cadre du langage procédural PL/SQL pour interagir avec les curseurs et les valeurs des curseurs.

```sql
SELECT dptno
FROM departement
FETCH FIRST 3 ROWS ONLY;
```

dptno
A00
B01
C01

2.4.6 Fonctions analytiques

Moins souvent utilisées, je les laisse en lecture de slides.

2.4.7 Bonnes pratiques

- Pas de préfixe inutile
- Alias court et significatif
- Pas de jointure inutile
- Éviter les négations (`NOT`) et différences (=!)
- Utiliser l'`ORDER BY` que si nécessaire

En général, il vaut mieux utiliser une jointure efficace que des conditions compliquées/

3 DDL (Data definition language)

On peut définir et modifier des tables ou d'autres objets via trois mots-clés :

`CREATE` Permet la création de table, vues, fonctions, etc. C'est une commande indispensable pour indiquer qu'on amène un nouvel objet, on peut l'accompagner de `OR REPLACE` pour écraser un objet déjà existant.

`ALTER` Permet la modification d'un objet existant, par exemple en ajoutant des colonnes ou des contraintes à une table existante.

`DROP` Permet de supprimer un objet et de l'abandonner définitivement.

3.1 Schéma conceptuel

C'est à la création d'une table qu'on va définir ses attributs, les types et possiblement certaines contraintes. On parle de schéma conceptuel car même si la table n'est pas encore peuplée, elle a déjà sa structure.

Voici un exemple minimal de schéma conceptuel, pour chaque nouvelle colonne on doit définir son type à l'avance. Ce type sera appliqué à toute la colonne et empêche l'insertion de types différents.

```sql
CREATE TABLE Departement (
dptNo CHAR(3),
dptLib VARCHAR(20)
dptMgr CHAR(3),
dptAdm CHAR(3)
)
```

En général on utilise les types suivants :

Chaînes de caractères

- Longueur fixe : `CHAR(n)`
- Longueur variable : `VARCHAR(n)`

Chaînes de bits

- Longueur fixe : `BIT(n)` ou `CHAR(n)`
- Longueur variable : `VARCHAR(n)`

Nombres

- Virgule fixe : `INTEGER` / `INT`, `SMALLINT`, `BIGINT`, `NUMERIC(precision, scale)` ou `DECIMAL(precision, scale)`
- Virgule flottante : `REAL`, `FLOAT(n)`

Heures et dates

— `DATE`, `TIME`, `TIMESTAMP`,...

Comme dit précédemment, on peut imposer des contraintes sur le schéma conceptuel, ainsi que des valeurs par défaut, il ne faut pas oublier que la contrainte de non nullité apparaît **avant** la valeur par défaut. Par exemple :

```
CREATE TABLE Departement (
dptNo CHAR(3) NOT NULL,
dptLib VARCHAR(20) DEFAULT 'valDef' NOT NULL,
dptMgr CHAR(3) NOT NULL,
dptAdm CHAR(3) DEFAULT 'D21')
```

3.1.1 Contraintes d'intégrités et clés

Les contraintes d'intégrité sont des assertions que les tables et relations devront respecter comme l'unicité des valeurs d'une colonne, une référence à une autre table, une validation de condition. On trouve quatre mots-clés.

— Clé primaire : `PRIMARY KEY`
— Unicité : `UNIQUE`
— Validation de condition : `CHECK`
— Clé étrangère : `FOREIGN KEY`

Les contraintes se déclarent à la création de la table ou lors d'une modification. Il vaut mieux donner un nom explicite à la contrainte pour pouvoir s'y référer par après.

```
[CONSTRAINT nomContrainte] expression de la contrainte
```

Voici un autre exemple :

```
CREATE TABLE Table (
attribut1 TYPE CONSTRAINT ContSurUnAttr expr1,
attribut2 ... ,
CONSTRAINT ContSurTable expr2
)
```

3.1.2 Identifiant primaire

On peut déclarer une clé primaire (identifiant unique et non nul) d'une table de cette manière :

```
CREATE TABLE Departement (
dptNo CHAR(3) NOT NULL CONSTRAINT dptPK PRIMARY KEY,
dptLib VARCHAR(20) NOT NULL,
dptMgr CHAR(3) NOT NULL,
dptAdm CHAR(3) )
```

La création au niveau de la colonne est limitée par exemple dans le cas où deux colonnes sont nécessaires pour créer une clé primaire. ou bien comme ceci :

```
CREATE TABLE Departement (
dptNo CHAR(3) NOT NULL,
dptLib VARCHAR(20) NOT NULL,
dptMgr CHAR(3) NOT NULL,
dptAdm CHAR(3),
CONSTRAINT dptPK PRIMARY KEY(dptNo) )
```

3.1.3 Unicité

On peut réclamer d'une ou plusieurs colonnes que leurs valeurs soient unique. Que ce soit pour éviter des doublons ou que deux, trois,... valeurs exactement similaires ne soient entrées.

```
[CONSTRAINT nomContrainte] UNIQUE [(attributs de la contrainte)]
```

On aura

```
CREATE TABLE Departement (
dptNo CHAR(3) NOT NULL CONSTRAINT dptPK PRIMARY KEY,
dptLib VARCHAR(20) NOT NULL CONSTRAINT dptLibUK UNIQUE,
dptMgr CHAR(3) NOT NULL,
dptAdm CHAR(3)
)
```

Ou bien

```
CREATE TABLE Departement (
dptNo CHAR(3) NOT NULL CONSTRAINT dptPK PRIMARY KEY,
dptLib VARCHAR(20) NOT NULL,
dptMgr CHAR(3) NOT NULL,
dptAdm CHAR(3),
CONSTRAINT dptLibUK UNIQUE(dptLib)
)
```

On peut évidemment avoir des contraintes successives

```
CREATE TABLE Participation (
courseId INT NOT NULL,
coureurId INT NOT NULL,
dossard VARCHAR(20) NOT NULL,
dptMgr DECIMAL(3) NOT NULL,
dptAdm CHAR(3),
CONSTRAINT participationPK
PRIMARY KEY(courseId, coureurId),
CONSTRAINT dossardParCourseUK
UNIQUE(courseId, dossard)
)
```

3.1.4 Validation de condition

Il est possible d'imposer des conditions aux valeurs d'une table, que les valeurs d'une colonne soient positives, qu'une colonne soit supérieure à une autre ou bien encore qu'une colonne respecte un certain format.

```
[CONSTRAINT nomContrainte] CHECK ( condition )
CREATE TABLE Employe (
empNo CHAR(3) NOT NULL,
empNom VARCHAR(50) NOT NULL,
empDpt CHAR(3) NOT NULL,
empSal NUMERIC(10,2) NOT NULL,
empSexe CHAR NOT NULL
CONSTRAINT empSexeCK CHECK (empSexe IN ('M', 'F')),
empDateEngage DATE NOT NULL,
empDateNaiss DATE NOT NULL,
CONSTRAINT empDateNaissCK
CHECK (empDateEngage > empDateNaiss))
```

3.1.5 Clé étrangère

```
[CONSTRAINT nomContrainte]
[FOREIGN KEY (les att. de la FK)]
REFERENCES nomTableCible [(les att. Clé référencée)]
[ ON DELETE {RESTRICT | SET NULL | CASCADE} ]
```

`RESTRICT` est l'option par défaut qui correspond au refus de suppression.

`SET NULL` demande, en cas de suppression du tuple référencé, de remplacer la valeur de la FK par `NULL`.

`CASCADE` demande en cas de suppression du tuple référencé de supprimer les tuples qui y font référence.

```
1. CREATE TABLE Departement (
..., dptMgr CHAR(3) NOT NULL CONSTRAINT ManagerFk
REFERENCES Employe (empNo) ON DELETE RESTRICT,
... )
DELETE FROM Employe -- refus de suppression

2. CREATE TABLE Departement (
..., dptMgr CHAR(3) NOT NULL CONSTRAINT ManagerFk
REFERENCES Employe (empNo) ON DELETE SET NULL,
... )
DELETE FROM Employe -- tous les dptMgr sont à NULL

3.CREATE TABLE Departement (
..., dptMgr CHAR(3) NOT NULL CONSTRAINT ManagerFk
REFERENCES Employe (empNo) ON DELETE CASCADE,
... )
DELETE FROM Employe -- tous les départements qui font
-- référence aux employés sont supprimés
```

Ce n'est pas tout, certaines contraintes on auront un effet "bloquant" dès qu'un changement est fait sur une table et parfois ce n'est pas le comportement qu'on attend. Dans ce cas, on peut attendre la fin d'une transaction pour que la contrainte soit vérifiée. En utilisant les mots-clés :

```
DEFERRABLE [INITIALLY { DEFERRED | IMMEDIATE }]
```

Le choix par défaut pourra être modifié dynamiquement par les programmes au travers de l'exécution de la commande suivante en début de transaction.

```
SET CONSTRAINT[S] { liste de contraintes | ALL }
{ IMMEDIATE | DEFERRED }
```

3.1.6 Suppression de table

On supprime une table avec

```
DROP TABLE nomTable [ CASCADE CONSTRAINTS ]
```

Une suppression d'une table référencée par une autre table n'est pas permise : il faut d'abord supprimer les clés étrangères qui la prennent pour cible.

`CASCADE CONSTRAINTS` permet d'éluder cette obligation avec les dangers que cela représente, à éviter.

3.1.7 Modification de table

On modifie une table à l'aide de `ALTER TABLE` en gardant en tête que :

`ADD` Permet l'ajout d'un attribut, d'une contrainte.

`MODIFY` Permet la modification d'un attribut.

`DROP` Permet la suppression d'un attribution, d'une contrainte.

Par exemple :

```
ALTER TABLE Employe ADD empDateNaissance DATE;
ALTER TABLE Employe MODIFY empDateNaissance NOT NULL;
ALTER TABLE Employe DROP COLUMN empDateNaissance;
ALTER TABLE Employe ADD CONSTRAINT salPositifCheck
CHECK (empSal > 0) ;
ALTER TABLE Employe DROP CONSTRAINT salPositifCheck ;
```

3.1.8 Sémantique

On peut ajouter des commentaires aux tables et aux colonnes de cette manière.

```
COMMENT ON TABLE nomTable IS 'description de la table'

COMMENT ON COLUMN nomTable.nomColonne
IS 'description de la colonne'
```

3.2 Schéma externe

3.2.1 Vues

La vue est une table virtuelle qui contient le résultat d'une opération `SELECT`, elle ne stocke pas les données. Elle fait simplement références aux tables de base à travers le `SELECT` et la requête qui la génère est exécuté à **chaque référence** à la vue.

C'est un outil utile car elles permettent l'indépendance logique des données, elles cachent les tables originales aux utilisateurs tout en simplifiant les noms ou les sélections complexes. Finalement, c'est une manière compacte de conserver des opérations nommées et/ou de masquer des jointures fréquemment utilisées.

Elle se créé, sans surprise, avec :

```
CREATE [OR REPLACE] VIEW nom-de-vue [(liste d'attributs)] AS
SELECT liste d'attributs ...
[WITH CHECK OPTION]
```

`WITH CHECK OPTION` permet d'assurer que la vue ne soit pas modifiée. Expliqué ici.

En pratique, c'est pour interdire aux utilisateurs qui ont le droit `UPDATE` ou `INSERT` sur la vue d'effectuer des manipulations qui génèrent des tuples qui ne sont pas visualisables au travers de la vue.

Par exemple :

```
CREATE VIEW DepCoord (no, lib, managerNom, masseSal) AS
SELECT dptNo, dptLib, mgr.empNom, SUM(emp.empSal)
FROM Departement
JOIN Employe mgr ON dptMgr=mgr.empNo
JOIN Employe emp ON dptNo=emp.empDpt
GROUP BY dptNo, dptLib, mgr.empnom;
```

et elle se supprime simplement avec :

```
DROP VIEW nom_vue;
```

3.2.2 Manipulation des vues

Il existe évidemment des restrictions sur les vues qui peuvent varier d'un SGBD à l'autre. En général, la vue est le produit d'une sélection et donc est plus difficile à modifier.

Exemple : pas de modification, suppression ou ajout possible sur cette vue.

```
CREATE VIEW masseSal(masse) AS
SELECT SUM(empSal) FROM Employe
```

1. Restriction en sélection

   On ne peut pas réaliser de groupements sur un attribut défini par le biais d'une fonction synthétique (`SUM`, `AVG`,...) et un tel attribut ne peut pas non plus être l'argument d'une fonction synthétique.

   ```
   CREATE VIEW masseSal(dpt, masse, nbEmp) AS
   SELECT empdpt, SUM(empSal), COUNT(*)
   FROM Employe
   GROUP BY empdpt
   SELECT nbEmp, COUNT(*)
   ```

```
6
7  FROM masseSal GROUP BY nbEmp -- interdit
8
9  SELECT AVG(masse) FROM masseSal -- interdit
```

2. Restriction en mise à jour
   Une mise à jour doit être faite sur une seule ligne des tables de base. Donc la vue ne peut pas avoir :
   — d'instructions ensemblistes (`UNION`, `INTERSECT`, `EXCEPT`)
   — de fonctions synthétiques
   — de clause `GROUP BY`
   Pour la mise à jour d'une vue avec jointure, la vue doit préserver l'unicité de la clé des tables de base.

```
1  CREATE VIEW chef AS
2  SELECT dptno, dptlib, dptmgr,
3  empno, empnom, empsal, empsexe, empdpt
4  FROM employe e
5  JOIN departement d ON d.dptno = e.empdpt;
```

   La table de base departement n'est pas clé-préservée. La table de base employe est clé-préservée.
3. Autres restrictions
   — En ajout :
      — Les attributs de la table non présents dans la vue seront affectés de valeur `NULL`.
   — En suppression et modification :
      — La vue ne doit pas contenir de clause `DISTINCT`.
      — La clause `WHERE` ne peut contenir un select imbriqué corrélé.
   — En modification :
      — Les expressions ne peuvent être modifiées (par ex. salaire *12).

4 DCL (Data control language)

Le contrôle des données portent sur les privilèges et contrôle d'accès des données ainsi que sur la gestion entourant les données comme la gestion des transaction et des concurrences. De ce fait, en DCL on s'intéresse aussi à la gestion de la sécurité qui entoure les bases de données.

4.1 Privilèges

Le langage SQL fournit deux mots clés permettant d'octroyer ou de supprimer des droits :

`GRANT` Permet de donner des droits de (`SELECT`, `UPDATE DELETE`, `INSERT`,...) à un/des utilisateurs sur un objet donné.

`REVOKE` Permet de retirer des droits de (`SELECT`, `UPDATE DELETE`, `INSERT`,...) à un/des utilisateurs sur un objet donné.

Voici la syntaxe :

```
1  GRANT {liste de privilèges | ALL}
2  ON nomObjet
3  TO {user | role | PUBLIC }
4  [WITH GRANT OPTION]
```

Privilège	Table	Vue	Séquence	Programme
SELECT	✓	✓	✓	
INSERT	✓	✓		
UPDATE	✓	✓		
DELETE	✓	✓		
EXECUTE				✓

Lorsqu'on utilise les mots-clés `WITH GRANT OPTION`, cela permet à l'utilisateur qui a obtenu des droits d'aussi les transmettre. Donc, pour pouvoir attribuer un privilège il faut soit être propriétaire de l'objet soit avoir reçu des droits `WITH GRANT OPTION`.

Pour expliquer le fonctionnement de l'obtention de privilège, voici un liste de commandes accompagnées d'un tableau. `U0` désigne le détenteur de la table et les indices donnent le numéro de la commande qui ont attribué tel droit à tel utilisateur.

L'utilisateur `U0` propriétaire de `Table1` exécute :

```
GRANT SELECT ON Table1 TO Public; --1
GRANT INSERT, UPDATE(a1,a2) ON Table1 TO Ua; --2
GRANT DELETE ON Table1 TO Ua, Ub; --3
GRANT ALL ON Table1 TO Uc; --4
GRANT INSERT Table1 TO Uz WITH GRANT OPTION; --5
```

Après quoi, l'utilisateur `Uz` exécute :

```
GRANT INSERT ON U0.Table1 TO Ub -- 6
GRANT ALL ON U0.Table1 TO Ud -- 7
-- La requête 7 ci-dessus ne donne que le privilège INSERT !
GRANT INSERT ON U0.Table1 TO Ue, Ua WITH GRANT OPTION -- 8
```

Récapitulatif :

Table1	U_0	U_a	U_b	U_c	U_d	U_e	U_z
SELECT	✓	✓$_1$	✓$_1$	✓$_{1,4}$	✓$_1$	✓$_1$	✓$_1$
INSERT	✓	✓$_{2,8}$	✓$_6$	✓$_4$	✓$_7$	✓$_7$	✓$_5$
UPDATE	✓	✓$_{2*}$		✓$_4$			
DELETE	✓	✓$_3$	✓$_3$	✓$_4$			

On enlève des privilèges de manière similaire :

```
REVOKE { liste de privilèges | ALL }
ON nomObjet
FROM { user | role | PUBLIC }
```

- Le privilège est enlevé immédiatement.
- Pour pouvoir enlever un privilège à l'utilisateur, il faut soit être propriétaire de l'objet reçu soit avoir reçu un privilège avec `WITH GRANT OPTION`.
- On ne peut pas supprimer un privilège à un utilisateur si celui ci l'a reçu via `PUBLIC`. On ne peut pas supprimer via `PUBLIC` un privilège à un utilisateur l'ayant reçu personnellement.
- Si plusieurs utilisateurs ont donné un droit sur un objet, il faut que chacun retire ses droites pour que le bénéficiaire n'en dispose plus.
- Il y a cascade dans la révocation d'un privilège transmis avec `WITH GRANT OPTION`.

Si, par exemple, L'utilisateur `U0` propriétaire de `Table1` exécute :

```
REVOKE SELECT ON Table1 FROM Ua; --1 (impossible !)
REVOKE DELETE ON Table1 FROM Public; --2 (impossible !)
REVOKE UPDATE ON Table1 FROM Ua --3
REVOKE DELETE Table1 FROM Uc; --4
REVOKE INSERT Table1 FROM Uz --5
REVOKE ALL ON Table1 FROM Ub --6
-- Ub possède toujours le privilège par PUBLIC.
REVOKE SELECT Table1 FROM Uc --7
-- Uc possède toujours le privilège par PUBLIC.
```

Table1	U_0	U_a	U_b	U_c	U_d	U_e	U_z
SELECT	✓	✓$_1$	✓$_1$	✓$_{1,\cancel{4}}^{7}$	✓$_1$	✓$_1$	✓$_1$
INSERT	✓	✓$_{2,\cancel{8}}^{5}$	~~✓$_6$~~^{5}	✓$_4$	~~✓$_7$~~^{5}	~~✓$_8$~~^{5}	~~✓$_5$~~^{5}
UPDATE	✓	~~✓$_{2*}$~~^{3}		✓$_4$			
DELETE	✓	✓$_3$	~~✓$_3$~~^{6}	~~✓$_4$~~^{4}			

Les privilèges sur les vues se comportent de la même façon, par exemple, imaginons que `U0` propriétaire de la table `Client`, exécute la requête.

```
CREATE VIEW clientsPositifs AS
SELECT * FROM Client WHERE compte > 0
WITH CHECK OPTION;
```

GRANT SELECT, UPDATE ON clientsPositifs TO Ua ;

On a donc attribuer des droits à une vue comme on aurait pu le faire sur un table, maintenant il y a un niveau d'abstraction en plus et l'utilisateur `Ua` n'a pas accès aux tables.

En pratique,

- Un utilisateur qui a les privilèges sur une vue ne doit pas avoir ce privilège sur les objets manipulés par cette vue.
- Un utilisateur qui a les privilèges sur un programme ne doit pas avoir ce privilège sur les objets manipulés par cet programme.
- Un programme/une vue s'exécutent par défaut avec les droits du propriétaires de la vue/du programme.

4.2 Transaction

On peut imaginer une base de données comme un ensemble de relations qui représentent des relations réelles du monde pratique, comme une relation entre client et fournisseur ou entre différents départements. Dans ce cas, une unité logique de travail, une **transaction** peut être plus qu'une simple requête. On peut vouloir définir plusieurs requêtes ou actions qui appartiennent à la même logique. Donc en clair, une transaction est :

- Unité de traitement séquentiel, exécutée pour le compte d'un usager qui, appliquée à une base de données cohérente, restitue une base de données cohérente.
- Une transaction est une séquence d'opérations du DML qui est *atomique* vis-à-vis des problèmes de concurrence et reprise après panne.

Par exemple, l'achat d'un journal peut être codé comme :

```sql
-- 1. Retirer le journal du stock

UPDATE stock SET quantite = quantite -1
WHERE idArticle = '123' ;

-- 2. Encaisser le montant

UPDATE caisse SET recette = recette + 2;
```

Ces deux opérations forment une transaction car elles appartiennent à la même logique. Si une des requêtes n'est pas exécutée alors il y a décalage entre la réalité du business et la représentation logique dans la base de données.

- Une transaction est une opération complexe
  - Atomique
  - Cohérente
  - Isolée
  - Durable

Cette logique dans l'utilisation des transactions portent aussi le nom de **ACID**, acronyme correspondant aux caractéristiques ci-dessus.

- **Atomicité**
  - L'exécution d'une transaction est atomique si, quoi qu'il arrive, elle est exécutée complètement ou pas du tout.
- **Cohérence**
  - La transaction doit faire passer la base de données d'un état cohérent à un autre.
- **Isolation**
  - Les transactions, même si elles s'exécutent sur un même intervalle de temps, travaillent sans interférence. Les transactions sont exécutées comme si chaque transaction disposait de la base de données pour elle seule.
- **Durabilité**
  - Dés que la transaction valide ses modifications, le SGBD doit garantir qu'elles sont permanentes même en cas de panne.

Le **début** d'une transaction démarre à la connexion ou à la fin de la transaction précédente. La fin d'une transaction est donnée par un `COMMIT` ou une instruction DDL.

Pour annuler une transaction, on utilisera un `ROLLBACK`.

La validation d'une transaction entraîne la persistance des effets des opérations effectuées tandis que l'annulation d'une transaction annule les effets d'une transaction.

Une transaction non-validée est automatiquement annulée. Aussi, une transaction est toujours exécutée pour le compte d'un utilisateur [une session de connexion à la base de données].

4.3 Problèmes

Les bases de données peuvent rencontrer des problèmes, typiquement quand des transactions essaient d'accéder aux **mêmes données** en lecture ou en écriture. Les SGBD mitigent ces problèmes au possible et il existe plusieurs manières de se protéger.

Voici quelques exemples :

4.3.1 Perte d'opération

Situation initiale $A = 3$

Transaction T1	**Transaction T2**
	– x2 ← Read(A)
x1 ← Read(A)	
	– x2 ← x2 + 3
x1 ← x1 + 1	
	– Write(x2) → A
Write(x1) → A	
A = 4	A = 6

Il y a un problème car les transactions lisent et écrivent en même temps ce qui rend le résultat incorrect en fin de transaction. En principe, le SGBD gère ce genre d'erreurs pour nous.

4.3.2 Lecture sale

Situation initiale $A = 3$

Transaction T1	**Transaction T2**
	–Write(5) → A
x1 ← Read(A)	
	`ROLLBACK`
A = 5	A = 3

On a `ROLLBACK` avant de valider quoi que ce soit, dans ce cas la transaction `T1` a une valeur de A qui n'existe pas.

4.3.3 Écriture sale

Situation initiale $A = 3$

Transaction T1	**Transaction T2**
	–Write(5) → A
Write(7) → A	
	`ROLLBACK`
A = 7	A = 3

Dans ce cas, `T1` a une valeur qui n'existe pas. La modification de `T1` est annulée par le `ROLLBACK` de `T2`, ce qui cause une écriture sale.

4.3.4 Non reproductibilité des lectures

Situation initiale $A = 3$

Transaction 1	Transaction 2
	– x2 ← Read(A)
x1 ← Read(A)	
	– x2 ← x2*2
	– Write(x2) → A
x3 ← Read(A)	
x1 = 3 et x3 = 6	
x1 = A = x3 ? ? ?	

Situation obtenue, $A = 6$ La même donné a plusieurs valeurs pour `T1`. Des `Read` à des moments différents lors d'une même transactions ne produisent pas la même valeur.

On obtient cette erreur si par exemple on lit une valeur, qu'on la modifie et puis qu'on la lit à nouveau.

4.3.5 Analyse incohérente

Contrainte d'intégrité : $A = B$, Situation initiale : $A = B = 5$

Transaction T1	Transaction T2
x1 ← Read(A)	
x1 ← x1 + 1	
Write(x1) → A	
	– x2 ← Read(B)
x3 ← Read(B)	
	– x4 ← Read(A)
x3 ← x3 + 1	
Write(x3) → B	
A = B	A = 6 et B = 5

Dans ce cas, $A = B = 6$ et `T2` observe une incohérence qui n'existe pas.

On peut introduire plusieurs problèmes d'incohérence en lisant et écrivant sur les mêmes valeurs dans des transactions en parallèle.

4.4 Concurrence

Pour remédier aux problèmes évoqués plus haut, on va utiliser un **contrôleur d'accès concurrents**, c'est le module chargé de contrôler les accès concurrents aux données.

On parle de concurrence d'accès quand au moins 2 transactions accèdent aux mêmes données.

L'unité de données contrôlée par le contrôleur de est appelée **granule de concurrence** et pouvant être, suivant le SGBD et/ou les configurations :

— La base de données
— La table
— Le bloc
— Le tuple

Plus le granule sera grand, plus le nombre de concurrences augmentera et moins le contrôleur aura de granules à surveiller.

Il existe plusieurs méthodes pour gérer la concurrence.

Définition 4.1 : Technique pessimiste

Le contrôleur empêche l'apparition de conflits en créant des files d'attente des granules sollicités.

Définition 4.2 : Technique optimiste

Le contrôleur laisse travailler chacune des transactions et lors de la détection de conflits amènera l'annulation de transactions conflictuelles.

Si `T1` et `T2` veulent lire ou écrire `A` au même moment, on a des problèmes :

Couple de transactions	Explications
Read - Read	T1 et T2 veulent écrire A → ok
Read - Write	T1 lit puis T2 écrit A → analyse incohérente et lecture non renouvelable
Write - Read	T1 écrit et puis T2 lit A → lecture sale
Write - Write	T1 et T2 écrivent A → écriture sale

4.5 Solutions

Une des solutions les plus courantes est l'utilisation de verrous, le contrôleur placera des verrous de différents types sur les granules sollicités. Nous allons en aborder 4.

Verrou court (C) est posé durant l'action

- Court partagé (**CS**)
- Court exclusif (**CX**)

Verrou long (L) est posé jusqu'à la fin de la transaction

- Long partagé (**LS**)
- Long exclusif (**LX**)

Verrou partagé S lock (de lecture) peut être posé sur un granule libre ou sur lequel est déjà posé un verrou partagé.

Verrou exclusif X lock (d'écriture) ne peut être posé que sur un granule libre (sur lequel n'est posé aucun verrou).

	X	**S**	**-**
X	Non	Non	Oui
S	Non	Oui	Oui
-	Oui	Oui	Oui

Le problème qui se pose donc est qu'on veut garder de bonnes performances en permettant des accès rapides et parallèles à la base de données tout en conservant la sécurité qui empêche les problèmes de concurrence.

En pratique, les verrous partagés permettent d'avoir plusieurs lectures parallèles et empêchent l'apparition d'un verrou exclusif tandis que les verrous exclusifs, en général pour l'écriture, empêche toute autre action. Dans ce cas il faudra attendre la fin de l'action ou de la transaction pour voir de nouveau accès à la base de données que ce soit en lecture ou en écriture. Quand une transaction tente de déposer un verrou sur une granule déjà occupée, elle est mise en attente. Les SGBD gèrent ça de manière différente vie des *time-out* ou pas et des détections d'inter-blocages.

Voici un exemple.

4.5.1 Protocole de verrouillage à 2 phases strict

1. Avant d'agir sur un objet A, une transaction doit obtenir un verrou sur cet objet A.
2. Après l'abandon d'un verrou, une transaction ne doit plus jamais pouvoir obtenir de verrous.

Exemple :

- Phase 1 : demande des verrous
  - Si une transaction veut faire une lecture de A, demande préalable de verrou LS sur A.
  - Si une transaction veut faire une écriture sur A : demande préalable de verrou LX sur A (si elle a déjà un verrou LS, demande de le promouvoir en LX sur A).
  - Si la demande ne peut être satisfaite, la transaction est mise en attente.
- Phase 2 : abandon des verrous
  - Les verrous sont relâchés à la fin de la transaction.

4.5.2 Perte d'opération

Situation initiale $A = 3$

Transaction T1	Transaction T2
	– Verrou LS sur A
	– x2 ← Read(A)
Verrou LS sur A	
x1 ← Read(A)	
	– x2 ← x2 + 3
x1 ← x1 + 1	
	- demande de verrou **LX** sur A
demande de verrou **LX** sur A	– Attente
– Attente	– Attente
– Attente	– Attente

Situation obtenue : **Deadlock** car chaque transaction attend que l'autre ait terminé pour lever son verrou.

4.5.3 Lecture sale

Situation initiale $A = 3$

Transaction T1	Transaction T2
	– Verrou **LX** sur A
	– Write (5) → A
Demande de verrou **LS** sur A	
Attente –	
Attente –	
	– `ROLLBACK`
Verrou **LS** sur A	
x1 ← Read(A)	
A = 3	A = 3

Situation obtenue : $A = 3$

OK ⇒ plus de lecture sale.

4.5.4 Écriture sale

Situation initiale $A = 3$

Transaction T1	Transaction T2
	– Verrou **LX** sur A
	– Write (5) → A
Demande de verrou **LX** sur A	
Attente –	
Attente –	
	– `ROLLBACK`
Verrou **LX** sur A	
Write(7) → A	

OK ⇒ plus d'écriture sale.

4.5.5 Analyse incohérente

Contrainte d'intégrité : $A = B$, Situation initiale $A = B = 5$

Transaction T1	Transaction T2
Verrou **LS** sur A	
x1 ← Read(A)	
x1 ← x1 + 1	
Verrou **LX** sur A	
Write(x1) → A	– Verrou **LS** sur B
	– x2 ← Read(B)
Verrou **LS** sur B	
x3 ← Read(B)	– Demande de verrou **LS** sur A
	– Attente
x3 ← x3 +1	– Attente
Demande de verrou **LX** sur B	– Attente
– Attente	– Attente
– Attente	– Attente

Situation obtenue : **Deadlock**. On est en attente de la résolution du verrou **LX** sur A qui n'a jamais été libéré.

4.5.6 Deadlock

On peut tenter de gérer les deadlocks de plusieurs manières :

- Soit
  - Détecter le blocage (= détecter un cycle dans un graphe d'attente) puis supprimer le blocage (= choisir une victime)
- Soit
  - Supprimer un blocage en utilisant un mécanisme de délai (time-out) pour les transactions inactives.
- Soit
  - Éviter les blocages (Wait-Die Wound-Wait)
    - Wait-Die : Si `T1` est plus ancienne que `T2` alors `T1` attend sinon `T1` meurt (`T1` est annulée puis relancéeà
    - Wound-Wait : Si `T1` est plus récente que `T2` alors `T1` attend sinon `T1` blesse `T2` (`T2` est annulée puis relancée).

L'outil essentiel de ce type de technique est la gestion des versions des données telles que gérées par les différentes transactions.

4.6 Niveaux d'isolation

4.6.1 Sérialisation

- Un ordonnancement des transactions est sérialisable et correct **SSI** il est équivalent à un ordonnancement séquentiel.
- Soit $T = T_1, T_2, T_3, ..., T_n$ ensembles des transactions concurrentes alors une exécution sérialisable de `T` doit donné le même résultat qu'une quelconque exécution séquentielle de `T` (par exemple `T2` puis `T1` puis `T3`,... puis T_n).
- Un ordonnancement est une exécution de transactions entrelacée ou séquentielle.
- Prises séparément les transactions sont supposées être correctes.
- Remarquons que deux ordonnancements séquentiels différents des mêmes transactions peuvent produire des résultats différents.
- Le verrouillage en 2 phases garanti la sérialisation.

Donc, un verrouillage correcte des données garantit la sérialisabilité des transactions (ce qui évite les problèmes).

Cependant, le protocole de verrouillage en 2 phases strict est contraignant, dans la mesure où il bloque potentiellement l'accès aux données durant une longue période de temps. Ne peut-on assouplir ce protocole ?

SQL2 définit trois types de problèmes potentiels :

Dirty read une transaction peut lire des données non validées par une autre transaction : une transaction peut donc être amenée à travailler sur le 'brouillon' d'autres transactions.

Non-repeatable read une transaction peut obtenir des valeurs différentes pour des lectures du même tuple.

Phantom-read une transaction peut être amenée lors du même `SELECT` à recevoir un nombre de tuples différents.

1. Fantôme

Transaction T1	Transaction T2
Verrou **LS** sur la table A	
x1 ← Read(A_i)	
(A_1,A_2,A_3)	
	– Verrou **LX** sur la table A
	– `INSERT` sur un nouveau A4
	– `COMMIT`
Verrou **LS** sur la table A	
x3 ← Read(A_i)	
(A_1,A_2,A_3)	

Situation obtenue : A4 est un fantôme pour `T1`.

2. Isolation

SQL2 définit la notion de degré d'isolation dans lequel chaque promoteur de transaction pourra la faire exécuter. Attention, SQL2 n'impose rien quant à l'implémentation de cette notion.

(a) **Dirty read** : Dirty-read, non-repeatable read et phantom read sont possibles.

(b) **Commited read** : Dirty-read est impossible (non-repeatable read et phantom read sont possibles).

(c) **Repeatable read** : Dirty-read, non-repeatable sont impossibles et phantom read est possible.

(d) **Serializable** : Aucun des 3 problèmes évoqués ne sont possibles.

L'implémentation de la notion de degré d'isolation étant laissée libre, différents SGBD peuvent avoir des transactions se comportant différemment tout en respectant la norme !

Pour une implémentation simpliste, purement pessimiste, des 3 premiers degrés, le contrôleur déposera des :

(a) verrous longs exclusifs en écriture.

(b) verrous courts partagés en lecture et verrous longs exclusifs en écriture.

(c) verrous longs partagés en lecture et verrous longs exclusifs en écriture.

(d) Ici, la méthode des verrous n'est efficace qu'à condition de verrouiller les tables accédées (ou certaines portions d'index).

3. Exercices

Donnez les scénarios pour les trois premiers niveaux d'isolation des transactions simultanées suivantes :

Transaction 1	Transaction 2
x1 ← Read(A1)	
x1 ← Read(A2)	
	– x2 ← Read(A3)
	– Write(x2-10) → A3
	– x2 ← Read(A1)
	– Write(x2+10) → A1
	– `COMMIT`
x1 ← x1 + Read(A3)	
`COMMIT`	

Au niveau 0, on a des verrous longs exclusifs en écriture.

Donnez les scénarios pour les trois premiers niveaux d'isolation des transactions simultanées suivantes :

Transaction 1	Transaction 2
x1 ← Read(A)	
x1 ← x1 + 1	
	– x2 ← Read(A)
	– x2 ← x2 * 2
Write(x1) → A	
	– Write(x2) → A
COMMIT	
	COMMIT

4.7 Problèmes restants

Il reste malgré tout des problèmes où parfois même le niveau **serializable** ne suffit pas.

Exemple :

```
CREATE TABLE A (x int);
CREATE TABLE B (x int);
```

Session 1	Session 2
ALTER SESSION SET ISOLATION_LEVEL = SERIALIZABLE	
	ALTER SESSION SET ISOLATION_LEVEL = SERIALIZABLE
INSERT INTO A SELECT COUNT(*) FROM B;	
	INSERT INTO B SELECT COUNT(*) FROM A;
COMMIT	
	COMMIT

5 Index

5.1 Qu'est-ce qu'un index ?

Un index est une structure de données utilisée et entretenue par le système de gestion de base de données pour lui permettre de retrouver rapidement des données. L'utilisation de l'index permet, dans certains cas, d'accélérer les opérations de tri, jointure et agrégation.

Il faut garder en tête que créer et maintenir un index a un coût en mémoire et ne rend pas forcément toutes les requêtes plus rapides.

La structure de données sous-jacente à l'index du SGBD est le **B-tree**, qui est similaire à un **arbre binaire de rechercher**. Plus précisément :

> Un arbre binaire de recherche (ABR) est un arbre binaire dans lequel chaque nœud possède une clé, telle que chaque nœud du sous-arbre gauche ait une clé inférieure ou égale à celle du nœud considéré, et que chaque nœud du sous-arbre droit possède une clé supérieure ou égale à celle-ci — selon la mise en œuvre de l'ABR, on pourra interdire ou non des clés de valeur égale.

5.2 Création d'un index en SQL

Sans surprise, pour créer un index, on utilise :

```
CREATE INDEX {nomIndex}
ON {nomTable} ("colonne 1", "colonne 2"...);

CREATE UNIQUE INDEX {nomIndex}
ON {nomTable} ("colonne 1", "colonne 2"...);
```

La dernière requête précise que les colonnes doivent être uniques.

Sur quels attributs créer un index ?

— Sur des attributs souvent utilisés comme critères de recherche (avec comme opérateur de comparaison : $=$, $>$, . . . , mais pas $\neq$).
— Sur des attributs constituant une clé étrangère.
— Sur des attributs peu modifiés (mettre à jour le B-tree a un coût).

Sur quel attributs ne **pas** créer l'index ?

— Sur les attributs d'une "petite table".
— Si l'attribut n'est jamais utilisé comme critère de recherche.
— Sur des attributs présentant "peu" de valeurs différentes.

On supprime un index avec

```
DROP INDEX {nomIndex}
```

Remarque : Le SGBD crée automatiquement un index pour chaque clé primaire d'une table donnée.

5.2.1 Index concaténé

On peut créer un index basé sur plusieurs colonnes :

```
CREATE INDEX last_first_name
ON member (last_name, first_name);
```

Le SGBD va considérer chaque colonne selon sa position dans la définition de l'index. La première colonne sera le premier critère de recherche, la seconde sera le second critère de recherche (comme dans une recherche dans le dictionnaire.)

Il y a quelques points à noter :

— On peut rechercher sur la première composante ou sur les deux premières, mais pas uniquement sur la deuxième.
— La commutativité de l'opérateur AND reste préservée grâce à la planification du SGBD.
— On ne peut pas utiliser la deuxième composante d'un index isolément.

Il est possible de demander au SGBD d'afficher le plan d'exécution d'une requête donnée. Ce plan montre les différentes étapes prises pour l'exécution de la requête (avec, notamment, les éventuels index utilisés ainsi qu'une estimation du "coût" de chaque étape).

6 PL/SQL

6.1 SGBD client-serveur

6.2 PL/SQL

6.2.1 Types de données

6.2.2 Blocs

6.2.3 Structures de données

6.2.4 Insertion de requête SQL

6.2.5 Curseurs

6.2.6 Outils utiles

6.2.7 Exceptions

6.3 Les procédures et fonctions stockées

6.4 Les déclencheurs