UNIVERSITÉ SAVOIE MONT BLANC

IUT ANNECY UNIVERSITÉ SAVOIE MONT BLANC

INFO

R2.06 – Exploitation des bases de données

CM#1 - Structured Query Language

- Langage de Manipulation des Données (LMD)
- Langage de Définition des Données (LDD)
- Langage de Contrôle des Transactions (LCT)

Christophe Lin-Kwong-Chon
Maitre de conférences – Université Savoie Mont-Blanc
christophe.lin-kwong-chon@univ-smb.fr

Ressource 2.06 – Exploitation des bases de données

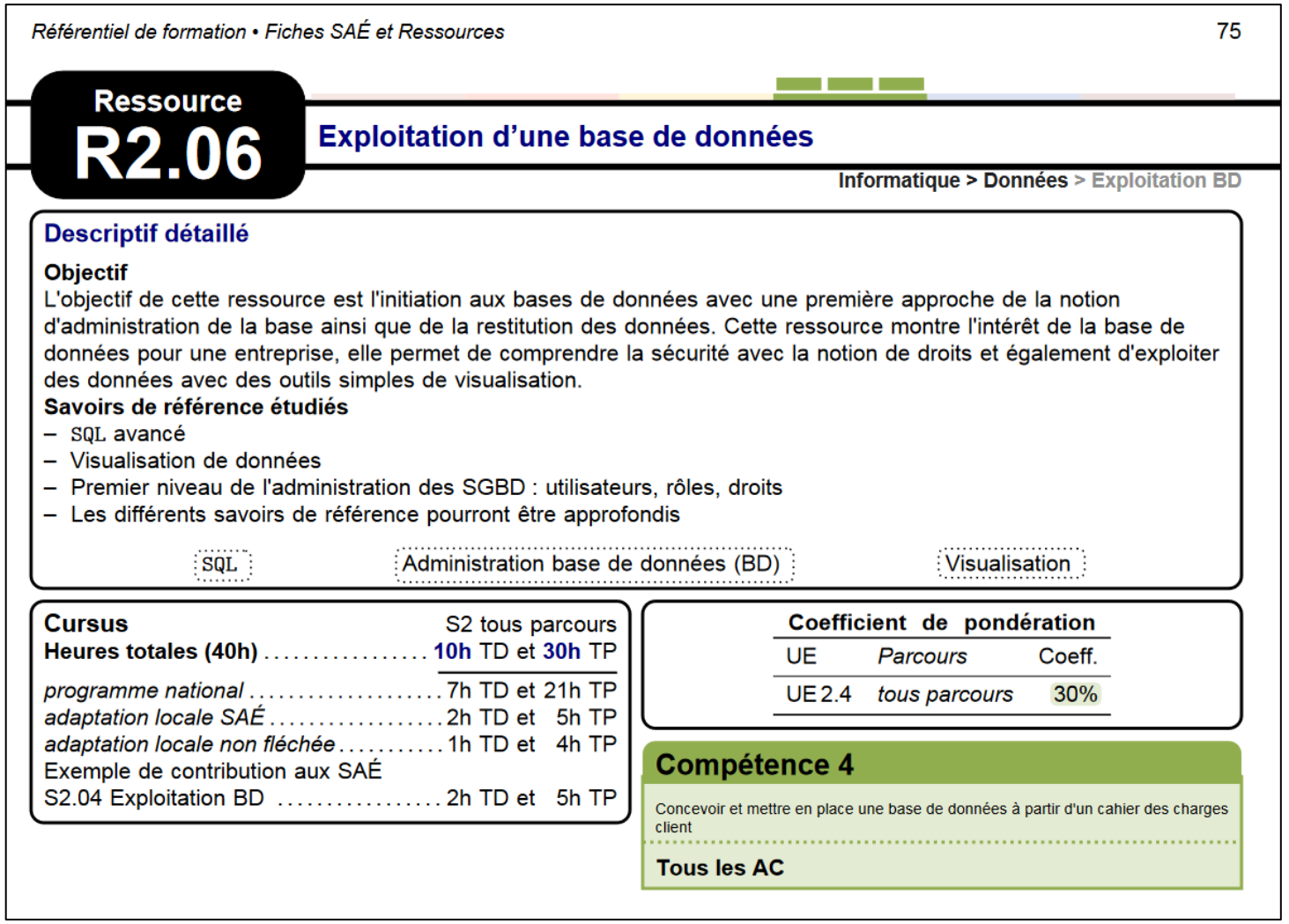
Syllabus – Programme

Pré-requis R1.05 - Introduction aux bases de données et SQL.

Evaluations 2 TPs notés + SAé (Rapports, Scripts, Présentation orale en anglais).

Outils PostgreSQL, Excel, Microsoft PowerBI, VSCode, pgAdmin.

Bibliographie *SQL Les fondamentaux du langage (avec exercices et corrigés), A.C. Bisson, ENI Editions, 2020, 4e édition.*



Référentiel de formation • Fiches SAÉ et Ressources 75

Ressource R2.06 — **Exploitation d'une base de données**

Informatique > Données > Exploitation BD

Descriptif détaillé

Objectif

L'objectif de cette ressource est l'initiation aux bases de données avec une première approche de la notion d'administration de la base ainsi que de la restitution des données. Cette ressource montre l'intérêt de la base de données pour une entreprise, elle permet de comprendre la sécurité avec la notion de droits et également d'exploiter des données avec des outils simples de visualisation.

Savoirs de référence étudiés

- SQL avancé
- Visualisation de données
- Premier niveau de l'administration des SGBD : utilisateurs, rôles, droits
- Les différents savoirs de référence pourront être approfondis

SQL | Administration base de données (BD) | Visualisation

Cursus — S2 tous parcours

Heures totales (40h)	**10h** TD et **30h** TP
programme national	7h TD et 21h TP
adaptation locale SAÉ	2h TD et 5h TP
adaptation locale non fléchée	1h TD et 4h TP
Exemple de contribution aux SAÉ	
S2.04 Exploitation BD	2h TD et 5h TP

Coefficient de pondération

UE	*Parcours*	Coeff.
UE 2.4	*tous parcours*	30%

Compétence 4

Concevoir et mettre en place une base de données à partir d'un cahier des charges client

Tous les AC

Syllabus – Programme

Responsables de cours

	Groupes
Christophe Lin-Kwong-Chon	TD1 (TP11)
Gilbert Nicolas	TD1 (TP12)
Pascal Colin	TD2, TD3
Thibaut Daugin	TD4

Intervenants du cours

Thirioux François	TD1
Martin Hugo	TD2
Kossakowski Roman	TD3
Laissard Gerard	TD4
Vibrac Stephanie	TD1, TD2, TD3, TD4

Autonomie SAé

Type	Durée	Thème	Besoins
CM1	2 H	LMD (Rappels, Sous-requêtes simples) & LDD & Transactions (commit, rollback)	Amphi
TD1	2 H	Requêtes SQL simples	Salle normale
TP1	2 H	Requêtes SQL simples	Salle machine
TP2	2 H	Requêtes SQL simples	Salle machine
TP3	2 H	LMD, Transactions, LDD	Salle machine
TP4	2 H	LMD, Transactions, LDD	Salle machine
TP5	2 H	Requêtes SQL simples sur modèle TP3&4	Salle machine
CM2	2 H	Requêtes SQL complexes, Dataviz	Amphi
TD2	2 H	Requêtes SQL complexes	Salle normale
TP6	2 H	Requêtes SQL complexes	Salle machine
TP7	2 H	Requêtes SQL complexes	Salle machine
TD	2 H	SAE LN	Salle machine
TD	2 H	SAE STATS	Salle machine
TD	2 H	SAE	Salle machine
TD	1 H	TP noté	Salle machine
TP8	2 H	Visualisation de données avec PowerBI	Salle machine
TP9	2 H	Visualisation de données avec PowerBI	Salle machine
TP	2 H	SAE	Salle machine
CM3	2 H	Administration SGBDR	Amphi
TD	4 H	Administration SGBDR	Salle machine
TD	1 H	TP noté	Salle machine
Oral		SAE	Anglais

Sommaire

1. LMD

Basic	*SELECT, FROM, WHERE, DISTINCT, JOIN*
Combine	*UNION, EXCEPT*
Aggregate	*COUNT, SUM, AVG, MIN, MAX, GROUP BY, HAVING*
Nested	*IN, ALL, ANY*
Manipulation	*INSERT, UPDATE, DELETE*
PostgreSQL functions	Scalaires, Conditionnal, Date and Time, Conversion

2. LDD

	CREATE	***ALTER***	***DROP***
Table	R1.05	R1.05	R1.05
Sequence	Today	Today	Today
View	CM2	CM2	CM2
Index	S3-S4	S3-S4	S3-S4

3. LCT

A	Atomicity
C	Consistency
I	Isolation
D	Durability

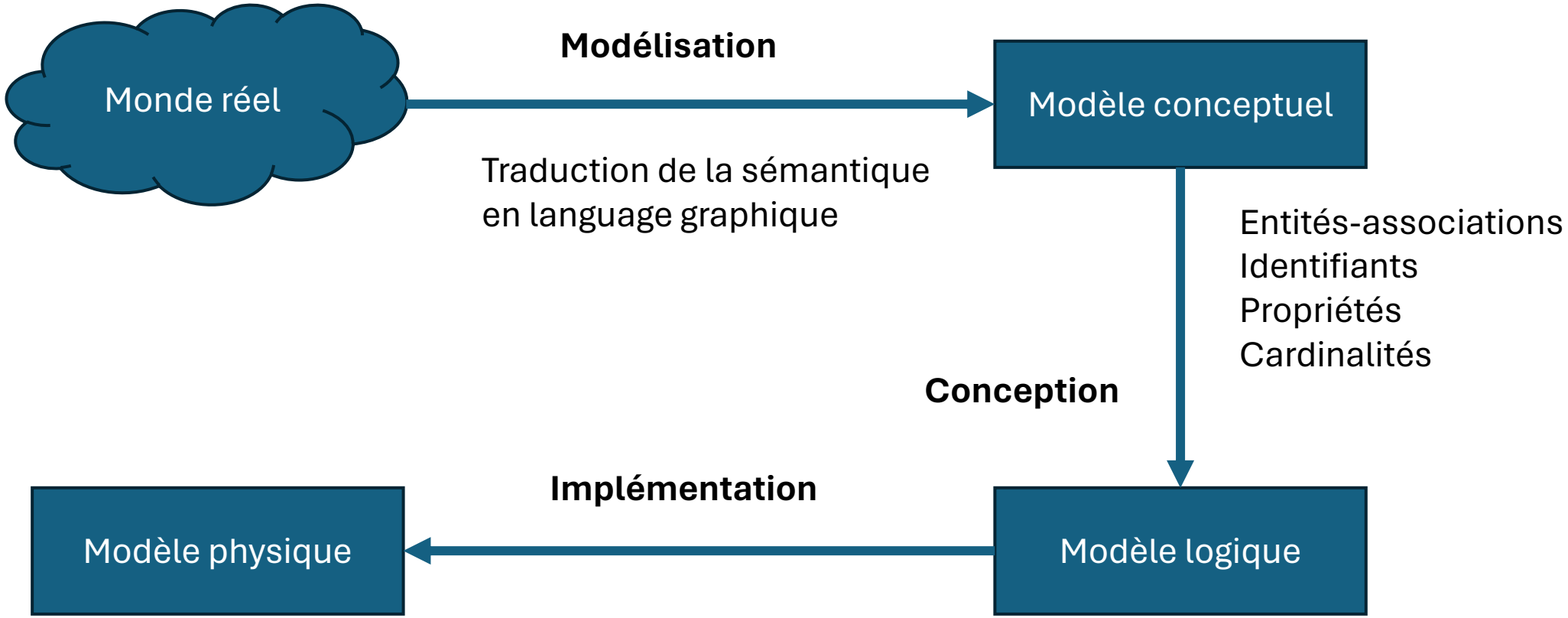
COMMIT, ROLLBACK, SAVEPOINT

Sommaire

1. LMD

Basic	*SELECT, FROM, WHERE, DISTINCT, JOIN*
Combine	*UNION, EXCEPT*
Aggregate	*COUNT, SUM, AVG, MIN, MAX, GROUP BY, HAVING*
Nested	*IN, ALL, ANY*
Manipulation	*INSERT, UPDATE, DELETE*
PostgreSQL functions	Scalaires, Conditionnal, Date and Time, Conversion

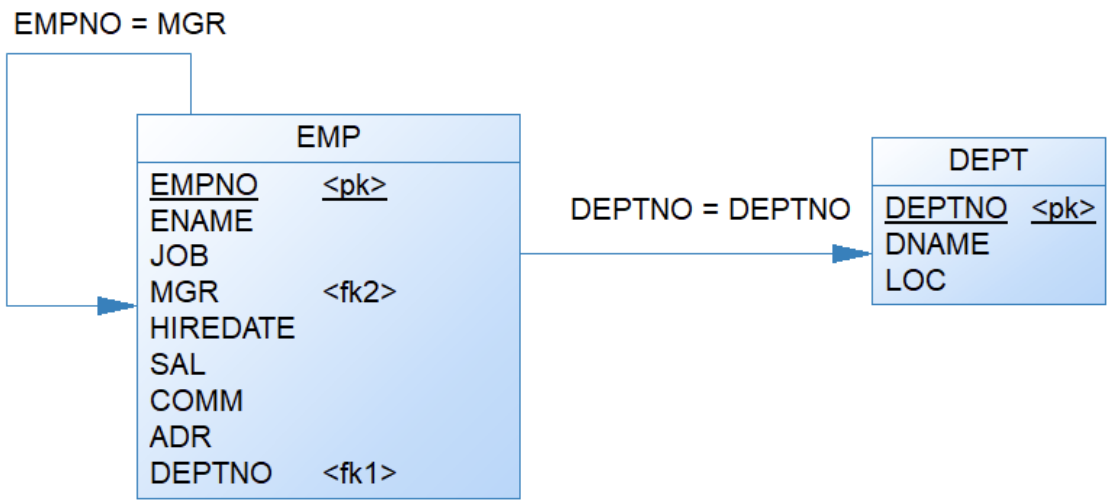
Rappels


Monde réel
Modélisation
Traduction de la sémantique
en language graphique
Modèle conceptuel
Entités-associations
Identifiants
Propriétés
Cardinalités
Conception
Implémentation
Modèle physique
Modèle logique
Script dépendant du SGBD
* Oracle, MySQL, PostgreSQL,
Cassandra, Apache, ...
Tables-Relations
Clés primaires, étrangères
Attributs, Méthodes

Rappels

Schéma de la base de données EMPLOYEES :

- DEPT(DEPTNO,DNAME,LOC)
- EMP(EMPNO,ENAME,ADR,JOB,#MGR,HIREDATE, SAL,COMM,#DEPTNO)



Rappels

Consultation des données

SELECT <liste des attributs projetés>
FROM <liste des relations touchées par la question>
[WHERE <liste des critères de restriction>
GROUP BY <liste des attributs d'agrégation>
HAVING <liste des critères de restriction sur les agrégats>
ORDER BY <liste des attributs de tri du résultat>**]**

Il est obligatoire de respecter cet ordre !

[ ] = clauses optionnelles

Rappels

Consultation des données

`IS NULL` (= valeur non renseignée)

```
SELECT * FROM emp WHERE comm IS NULL;
```

`LIKE`

A n'utiliser que si `%` et/ou `_` (**sinon utiliser =**)

```
SELECT ename FROM emp WHERE ename LIKE '_AR%';
```

`DISTINCT` (suppression des doublons)

```
SELECT DISTINCT job FROM emp;
```

Et non :

```
SELECT job FROM emp
GROUP BY job;
```

`IN`

```
SELECT ename, deptno FROM emp WHERE deptno IN
(10,20);
```

Rappels

Jointure interne (`join` ou `inner join`)

EMPNO	ENAME	DEPTNO
7369	SMITH	20
7499	ALLEN	30
7521	WARD	30

DEPTNO	DNAME	LOC
~~10~~	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO

SQL1 :

```
SELECT  ename, dname
FROM    emp, dept
WHERE   emp.deptno = dept.deptno;
```

SQL2 :

```
SELECT ename, dname
FROM   emp
 JOIN dept ON emp.deptno=dept.deptno;
```

```
SELECT ename, dname
FROM   dept
 INNER JOIN emp ON dept.deptno=emp.deptno;
```

Le mot clé `INNER` est optionnel.

Rappels

Auto-jointure

Opération de jointure d'une relation à elle-même.

Le synonyme (ou alias) de relation est indispensable.

Exemple :
Nom et salaire des employés ayant le même salaire que 'FORD'

```
SELECT e2.ename, e2.sal
FROM emp e1
  JOIN emp e2 ON e1.sal = e2.sal
WHERE e1.ename = 'FORD';
```

Equivalence en sous-requête

```
SELECT e2.ename, e2.sal
FROM emp e2
WHERE e2.sal = (
  SELECT e1.sal
  FROM emp e1
  WHERE e1.ename = 'FORD'
);
```

Rappels

Jointure externe (`outer join`)

Les lignes qui ne peuvent pas être jointes ne sont pas éliminées et restent intégrés.

Le mot clé `OUTER` est optionnel.

La commande `FULL OUTER JOIN` n'est pas implantée dans MySQL.

```
EMP FULL OUTER JOIN DEPT ON dept.deptno=emp.deptno;
```

EMP	EMPNO	ENAME	DEPTNO	SAL
	7369	SMITH	20	800.00
	7499	ALLEN	30	1600.00
	7521	WARD	40	1250.00
	7566	JONES	20	2975.00

DEPT	DEPTNO	DNAME
	10	ACCOUNTING
	20	RESEARCH
	30	SALES

EMPNO	ENAME	DEPTNO	SAL	DNAME
7369	SMITH	20	800.00	RESEARCH
7499	ALLEN	30	1600.00	SALES
7521	WARD	40	1250.00	*NULL*
7566	JONES	20	2975.00	RESEARCH
NULL	*NULL*	10	*NULL*	ACCOUNTING

Rappels

Jointure externe
(outer join)

EMP	EMPNO	ENAME	DEPTNO	SAL
	7369	SMITH	20	800.00
	7499	ALLEN	30	1600.00
	7521	WARD	40	1250.00
	7566	JONES	20	2975.00

DEPT	DEPTNO	DNAME
	10	ACCOUNTING
	20	RESEARCH
	30	SALES

```
EMP FULL OUTER JOIN DEPT ON dept.deptno=emp.deptno;
```

EMPNO	ENAME	DEPTNO	SAL	DNAME
7369	SMITH	20	800.00	RESEARCH
7499	ALLEN	30	1600.00	SALES
7521	WARD	40	1250.00	*NULL*
7566	JONES	20	2975.00	RESEARCH
NULL	*NULL*	10	*NULL*	ACCOUNTING

```
EMP LEFT OUTER JOIN DEPT ON dept.deptno=emp.deptno;
```

EMPNO	ENAME	DEPTNO	SAL	DNAME
7369	SMITH	20	800.00	RESEARCH
7499	ALLEN	30	1600.00	SALES
7521	WARD	40	1250.00	*NULL*
7566	JONES	20	2975.00	RESEARCH

```
EMP RIGHT OUTER JOIN DEPT ON dept.deptno=emp.deptno;
```

EMPNO	ENAME	DEPTNO	SAL	DNAME
7369	SMITH	20	800.00	RESEARCH
7499	ALLEN	30	1600.00	SALES
7566	JONES	20	2975.00	RESEARCH
NULL	*NULL*	10	*NULL*	ACCOUNTING

Rappels

Jointures

Jointure interne : `INNER JOIN` ou `JOIN`

```
SELECT ename, dname
FROM emp
     JOIN dept ON
          emp.deptno=dept.deptno;
```

Jointure externe :
A gauche : `LEFT OUTER JOIN` ou `LEFT JOIN`

```
SELECT ename, dname
FROM emp
LEFT JOIN dept ON emp.deptno=dept.deptno;
```

A droite : `RIGHT OUTER JOIN` ou `RIGHT JOIN`

```
SELECT ename, dname
FROM emp
RIGHT JOIN dept ON emp.deptno=dept.deptno;
```

Sommaire

1. LMD

Basic	*SELECT, FROM, WHERE, DISTINCT, JOIN*
Combine	*UNION, EXCEPT*
Aggregate	*COUNT, SUM, AVG, MIN, MAX, GROUP BY, HAVING*
Nested	*IN, ALL, ANY*
Manipulation	*INSERT, UPDATE, DELETE*
PostgreSQL functions	Scalaires, Conditionnal, Date and Time, Conversion

UNION

- COMMANDE(NUMCOMMANDE, NOMCLIENT, NOMPRODUIT, QUANTITE)
- FOURNITURE(NOMPRODUIT, NOMFOURNISSEUR, PRIX)

```
SELECT NOMPRODUIT
      FROM FOURNITURE
WHERE PRIX >= 1000
UNION
SELECT NOMPRODUIT
      FROM COMMANDE
WHERE NOMCLIENT = 'Jean'
```

`UNION` est utilisé pour unir 2 ensembles portant sur des tables différentes.

Le nombre de champs dans chaque SELECT doit être identique et les champs de même type.

L'union élimine les doublons. Pour garder les doublons on utilise l'opération `UNION ALL`.

DIFFERENCE

```
SELECT DEPTNO FROM DEPT
EXCEPT
SELECT DEPTNO FROM EMP
```

`EXCEPT` dans SQL Server et PostgreSQL ;
`MINUS` dans Oracle ;
La commande n'est pas implantée dans MySQL.

Utilisé pour faire la différence entre 2 ensembles.

Le nombre de champs dans chaque SELECT doit être identique et les champs de même type.

La différence élimine les doublons. Pour garder les doublons on utilise l'opération `EXCEPT ALL`.

Peut être remplacé par une sous interrogation `NOT IN` ou `NOT EXISTS` (CM#2).

Sommaire

1. LMD

Basic	*SELECT, FROM, WHERE, DISTINCT, JOIN*
Combine	*UNION, EXCEPT*
Aggregate	*COUNT, SUM, AVG, MIN, MAX, GROUP BY, HAVING*
Nested	*IN, ALL, ANY*
Manipulation	*INSERT, UPDATE, DELETE*
PostgreSQL functions	Scalaires, Conditionnal, Date and Time, Conversion

Fonction de groupe
==
Fonction d'agrégat

Principales fonctions d'agrégat

- `COUNT` : Nombre de valeurs non nulles de l'attribut spécifié
- `SUM` : Total des valeurs de l'attribut spécifié (somme)
- `AVG` : Moyenne des valeurs de l'attribut spécifié
- `MIN` : Valeur minimale de l'attribut spécifié
- `MAX` : Valeur maximale de l'attribut spécifié

Attention :

Les fonctions d'agrégats ignorent les valeurs nulles.

Les calculs qui font référence à des valeurs nulles restituent une valeur nulle.

Exemple :

```
SELECT AVG(sal) "Moyenne" FROM emp
Moyenne
----------
2073.21429
```

COUNT Compte le nombre de lignes du résultat d'une requête sans élimination des doublons ni vérification des valeurs nulles.

```
SELECT COUNT(empno) "Nb employés"
FROM  emp;
 Nb employés
----------
        14
```

```
SELECT COUNT(sal) "Nb salaires",
     AVG(sal) "Moyenne", MIN(sal) "Sal mini",
     MAX(sal) "Sal maxi", SUM(sal) "Somme sal"
FROM emp
Nb salaires    Moyenne   Sal mini   Sal maxi  Somme sal
----------- ---------- ---------- ---------- ----------
         14 2073.21429        800       5000      29025
```

```
SELECT COUNT(DISTINCT deptno) "NB dept",
       COUNT(DISTINCT job) "NB job"
FROM emp;
   NB dept     NB job
---------- ----------
         3          5
```

```
SELECT AVG(comm)
FROM emp
WHERE deptno = 40;
AVG(COMM)
---------
```

GROUP BY

La clause `GROUP BY` :

- organise les données en groupes,
- est toujours utilisée avec les fonctions d'agrégat (`SUM`, `AVG`, ...),
- le regroupement peut être effectué sur les valeurs d'une ou plusieurs colonnes ou une expression.

```
SELECT deptno, AVG(sal)
FROM emp
⇒ ERROR at line 1
```

→

```
SELECT deptno, AVG(sal)
FROM emp
GROUP BY deptno
```

Tous les champs du SELECT hormis celui sur lequel porte la fonction d'agrégat doivent être dans la clause GROUP BY

```
SELECT d.deptno, d.dname, AVG(e.sal) "Moyenne"
FROM emp e
    JOIN dept d ON e.deptno=d.deptno;
GROUP BY d.deptno, d.dname
```

WHERE
HAVING

La clause `WHERE` associée à `GROUP BY` :

- Supprime les lignes avant regroupement
- **N'admet pas de fonctions d'agrégat**

La clause `HAVING` :

- Sert à éliminer des valeurs de groupe une fois celles-ci calculées
  => intègre des fonctions d'agrégat
- Il est possible d'utiliser des sous-interrogations

```
SELECT    deptno, COUNT(*) "Nb"
FROM      emp
WHERE     job <> 'SALESMAN'
GROUP BY  deptno
HAVING    COUNT(*) > 3;
```

GROUP BY Remarques

D'après la norme SQL, toutes les colonnes autres que celles de type agrégat se trouvant au niveau du SELECT doivent se retrouver dans le GROUP BY. Autrement dit, la clause SELECT ne doit comporter que des colonnes figurant dans la clause GROUP BY ou des colonnes de types agrégat.

```
SELECT      t1.attr1, t1.attr2, sum(t1.attr3),…
FROM t1
    JOIN t2 ON …
GROUP BY    t1.attr1, t1.attr2
```

Ce n'est pas le cas sous MySQL, ce qui peut engendrer de graves erreurs notamment de calcul (somme aberrante,…) où un seul attribut suffit.

```
SELECT      t1.attr1, t1.attr2, sum(t1.attr3),…
FROM t1
    JOIN t2 ON …
GROUP BY    t1.attr1
```

Sommaire

1. LMD

Basic	*SELECT, FROM, WHERE, DISTINCT, JOIN*
Combine	*UNION, EXCEPT*
Aggregate	*COUNT, SUM, AVG, MIN, MAX, GROUP BY, HAVING*
Nested	*IN, ALL, ANY*
Manipulation	*INSERT, UPDATE, DELETE*
PostgreSQL functions	Scalaires, Conditionnal, Date and Time, Conversion

Sous-interrogation

Elles permettent :

- de structurer un problème par décomposition en sous-problèmes indépendants
- de réaliser des tâches qui ne pourraient se réaliser avec une jointure

Les sous-interrogations simples sont évaluées par l'analyseur avant l'interrogation principale
Opérateurs :

`IN, NOT IN`
`{=, !=, > , >=, <, <=}`
`{ANY, ALL}`

Remarques :

Les sous-interrogations peuvent apparaître dans les ordres `INSERT`, `DELETE`, `UPDATE`, `CREATE TABLE AS ...`
Certains SGBD n'acceptent pas les clauses `ORDER BY` dans les sous-interrogations.

Sous-interrogation

Employés travaillant à DALLAS

```
SELECT e.ename, e.sal, e.comm
FROM emp e
    JOIN dept d ON e.deptno = d.deptno
WHERE d.loc = 'DALLAS';

SELECT ename, sal, comm
FROM emp
WHERE deptno = (SELECT deptno FROM dept WHERE loc = 'DALLAS');
```

1 attribut de même type ! Généralement, le lien se fait entre la PK et la FK

La 1ère utilise uniquement une jointure, la 2nde utilise une sous-interrogation => la jointure s'exprime par 2 blocs SFW imbriqués.

Toutes les données ramenées dans la clause SELECT appartiennent à la relation EMP. Dans la clause FROM seule cette relation devrait figurer, la sous-interrogation servant uniquement à retrouver le numéro de service affecté à 'DALLAS'.

La sous-interrogation est indépendante de l'interrogation principale.

Sous-interrogation

Employés habitant au même endroit que leur lieu de travail

```
SELECT e.ename, e.adr
FROM emp e
    JOIN dept d ON e.deptno = d.deptno AND e.adr=d.loc;

SELECT e.ename, e.adr
FROM emp e
WHERE (e.deptno, e.adr) IN (SELECT deptno, loc FROM dept);
```

```
ENAME      ADR
---------- ----------
KING       NEW YORK
SMITH      DALLAS
JONES      DALLAS
MARTIN     CHICAGO
```

n attributs de même type !

Cette requête ne fonctionne pas sous MySQL.

IN

Si une requête est susceptible de renvoyer plus d'une valeur, utiliser `IN` au lieu de =

Employés embauchés le premier semestre 81

```
SELECT ename, job
FROM  emp
WHERE  hiredate IN
           (SELECT  hiredate
           FROM    emp
           WHERE  hiredate BETWEEN
                  '1981-01-01' AND '1981-06-30')
```

Employés basés à Dallas ou gagnant plus de 2000 €

```
SELECT ename, deptno
FROM emp
WHERE empno IN
        (SELECT empno
         FROM emp
         WHERE deptno IN
                   (SELECT deptno
                    FROM dept
                    WHERE loc = 'DALLAS')
         OR sal > 2000)
```

ALL

Employés ayant un salaire supérieur à tous les salaires des 'SALESMAN' ne s'appelant pas 'ALLEN'

```
SELECT ename, sal
FROM emp
WHERE sal > ALL
     (SELECT sal
     FROM     emp
     WHERE    job = 'SALESMAN'
     AND      ename != 'ALLEN')
```

```
ENAME             SAL
---------- ---------
KING             5000
CLARK            2450
BLAKE            2850
JONES            2975
ALLEN            1600
SCOTT            3000
FORD             3000
```

La condition $f\ \theta$ `ALL (SELECT ... FROM ...)` est vraie ssi
la comparaison $f\ \theta\ v$ est vraie pour **toutes** les valeurs v du résultat de la sous-requête.

ANY

Employés ayant un salaire supérieur à un des salaires des employés 'SALESMAN' ne s'appelant pas 'ALLEN'

```
SELECT ename, sal
FROM emp
WHERE sal > ANY
     (SELECT sal
     FROM     emp
     WHERE    job = 'SALESMAN'
     AND      ename != 'ALLEN')
```

```
ENAME             SAL
---------- ---------
KING             5000
CLARK            2450
BLAKE            2850
JONES            2975
ALLEN            1600
SCOTT            3000
FORD             3000
```

La condition $f\ \theta$ `ANY (SELECT ... FROM ...)` est vraie ssi
la comparaison $f\ \theta\ v$ est vraie pour **une** valeur v du résultat de la sous-requête.

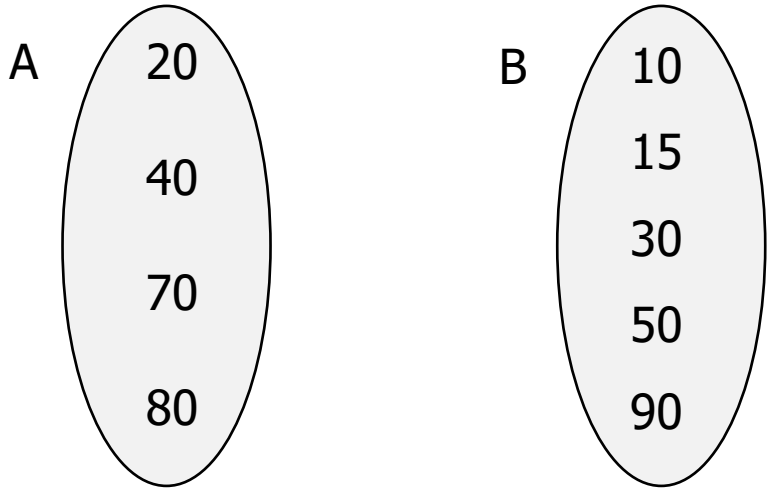
ALL
!=
ANY

```
SELECT ID FROM B WHERE ID > ALL (SELECT ID FROM A)
```

RETOURNE toutes les valeurs de B supérieures à la plus grande des valeurs de A => 90

```
SELECT ID FROM B WHERE ID > ANY (SELECT ID FROM A)
```

RETOURNE chaque valeur de B supérieure à l'une des valeurs de A, autrement dit toutes les valeurs de B supérieures à la valeur minimale de A => 30, 50 et 90.



Sommaire

1. LMD

Basic	*SELECT, FROM, WHERE, DISTINCT, JOIN*
Combine	*UNION, EXCEPT*
Aggregate	*COUNT, SUM, AVG, MIN, MAX, GROUP BY, HAVING*
Nested	*IN, ALL, ANY*
Manipulation	*INSERT, UPDATE, DELETE*
PostgreSQL functions	Scalaires, Conditionnal, Date and Time, Conversion

MANIPULATION

`INSERT`	Insertion d'une ou plusieurs lignes dans une table existante.
`DELETE`	Destruction d'une ou plusieurs lignes d'une table existante.
`UPDATE`	Mise à jour d'une ou plusieurs lignes d'une table existante.

L'ordre `SELECT` (sous-requête) peut être utilisé dans les ordres `INSERT`, `UPDATE` et `DELETE`

Pour insérer des n-uplets :

$$\texttt{INSERT INTO R(}A_1\texttt{,}A_2\texttt{,...,}A_n\texttt{) VALUES (}v_1\texttt{, }v_2\texttt{,... }v_n\texttt{)}$$

Donc on donne deux listes: celles des attributs (les A_i) de la table et celle des valeurs respectives de chaque attribut (les v_i).

- Chaque A_i doit être un attribut de R.
- Les attributs non-indiqués restent à **NULL** ou à leur valeur par défaut.
- On doit toujours indiquer une valeur pour un attribut déclaré **NOT NULL**

INSERT

- Exemple sans nom de colonne :

```
INSERT INTO dept
VALUES (50, null, null)
1 row created
```

- Exemple avec nom de colonne :

```
INSERT INTO dept (deptno)
VALUES (50)
1 row created
```

Pour une meilleure évolutivité, il est préférable de nommer les colonnes utilisées.

INSERT

- Exemples insert avec select (permet d'insérer plusieurs lignes) :

```
INSERT INTO NewEmp
SELECT *
FROM   emp
14 rows created.

INSERT INTO NewEmp (deptno)
SELECT DISTINCT deptno
FROM   emp
3 rows created.
```

- Exemple d'insertion avec des chaînes de caractères :

```
INSERT INTO DEPT (deptno, dname)
VALUES (50, 'COURS')
```

INSERT

Ne pas mixer `VALUES` et `SELECT` dans les `INSERT`.

On utilise soit :

`VALUES` si les données ne proviennent pas d'une table
`SELECT` au sein d'une sous-requête si les (ou certaines) données proviennent d'une table

- Exemples :

```
INSERT INTO emp(empno, ename, deptno)
    VALUES (1, 'BERNAUD', 10);
INSERT INTO emp (empno, ename, deptno)
    SELECT 1, 'BERNAUD', deptno FROM dept
    WHERE dname='ACCOUNTING';
```

ET NON :

```
INSERT INTO emp (empno, ename, deptno)
    VALUES (1, 'BERNAUD', (SELECT deptno FROM dept WHERE
            dname='ACCOUNTING'));
```

UPDATE

On modifie une table avec la commande UPDATE :

```
UPDATE R SET A1=v1,A2=v2,...,An=vn
WHERE condition
```

Contrairement à `INSERT`, `UPDATE` s'applique à un ensemble de lignes.

- On énumère les attributs que l'on veut modifier.
- On indique à chaque fois la nouvelle valeur.
- La clause `WHERE` *condition* permet de spécifier les lignes auxquelles s'applique la mise à jour. Elle est identique au `WHERE` du `SELECT`

Bien entendu, on ne peut pas violer les contraintes sur la table.

UPDATE

- Exemple sans `WHERE` : augmentation des salaires de 5 %

```
UPDATE emp
SET    sal = sal * 1.05
14 rows updated
```

- Exemple avec `WHERE` : augmentation du salaire des vendeurs

```
UPDATE emp
SET    sal = sal * 1.05
WHERE  job = 'SALESMAN'
4 rows updated.
```

UPDATE

- Exemple : sous-requête synchronisée

```
UPDATE emp e
SET    sal = (SELECT AVG(e2.sal)
              FROM   emp e2
              WHERE  e.deptno = e2.deptno)
```

- Exemple : avec jointure

```
UPDATE emp
SET    Sal = Sal + 100
FROM   emp e
   JOIN dept d ON  e.deptno = d.deptno
WHERE    d.loc = 'NEW YORK’
```

La jointure ne marche pas dans tous les SGBD, privilégiez la sous-requête.

DELETE

On détruit une ou plusieurs lignes dans une table avec la commande `DELETE` :

```
DELETE FROM R WHERE condition
```

C'est la plus simple des commandes de mise-à-jour puisque elle s'applique à des lignes et non à des attributs.

Comme précédemment, la clause WHERE *condition* est identique au `WHERE` du SELECT (=restriction des nombres de lignes impliquées)

DELETE

- Exemple sans WHERE : suppression de tous les services

```
DELETE FROM dept
4 rows deleted.
```

- Exemple avec WHERE : suppression des employés ayant une commission égale à zéro

```
DELETE FROM emp
WHERE  comm = 0
1 row deleted.
```

- Exemple avec WHERE et SELECT : suppression des employés travaillant à DALLAS

```
DELETE FROM emp
WHERE deptno =
    (SELECT deptno
     FROM dept
     WHERE  loc = 'DALLAS')
5 rows deleted.
```

Sommaire

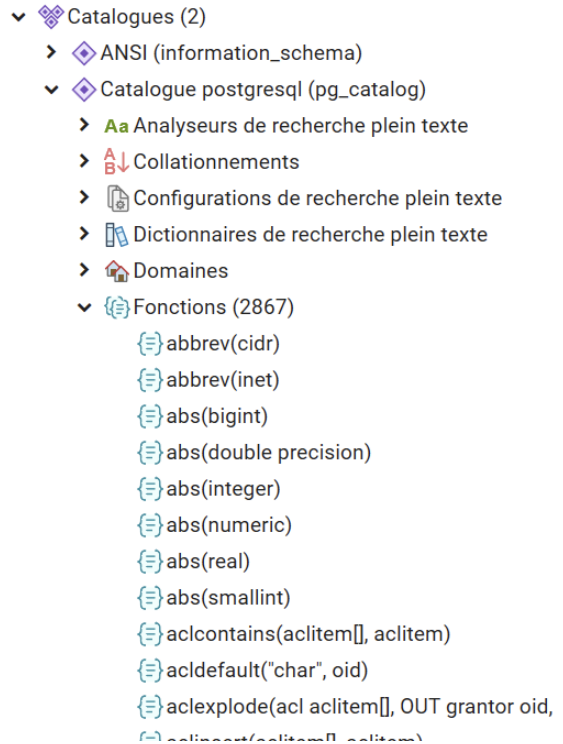
1. LMD

Basic	*SELECT, FROM, WHERE, DISTINCT, JOIN*
Combine	*UNION, EXCEPT*
Aggregate	*COUNT, SUM, AVG, MIN, MAX, GROUP BY, HAVING*
Nested	*IN, ALL, ANY*
Manipulation	*INSERT, UPDATE, DELETE*
PostgreSQL functions	Scalaires, Conditionnal, Date and Time, Conversion

Fonctions simples

Contrairement à une fonction de groupe, une fonction scalaire ne s'applique pas sur un ensemble de lignes, mais ligne par ligne

Elles :
- ramènent une valeur par ligne sélectionnée,
- peuvent s'imbriquer les unes-les autres sans limite,
- peuvent être présentes dans toutes les clauses des ordres `SELECT`, `UPDATE`, `DELETE`, `INSERT`.
- permettent de transformer des valeurs simples ou des valeurs issues d'expressions.



PostgreSQL 13 intègre plus de 2800 fonctions.
Cf. doc PostgreSQL

Le format est généralement :

```
nom_fonction(argument1, ....)
```

Fonctions simples

- Arithmétiques

`SIGN(nbr)`	Retourne -1, 0 ou 1 en fonction du signe de nbr
`FLOOR(nbr)`	Retourne la plus grande valeur entière inférieure à nbr
`CEIL(nbr)`	Retourne la plus petite valeur entière supérieure à nbr
`ROUND(nbr)`	Retourne l'arrondi entier le plus proche
`ROUND(nbr,D)`	Retourne l'arrondi à *D* décimales le plus proche
`TRUNC(nbr,D)`	Retourne nbr tronqué à *D* décimales

- Pour chaînes de caractères

`chaîne1 || chaîne2`	Concatène les chaînes
`LENGTH(chaîne)`	Retourne la longueur de la chaîne
`LOWER(chaîne), UPPER(chaîne)`	Retourne la chaîne en minuscule ou majuscule
`STRPOS(chaîne, sous-chaîne)`	Retourne l'indice de la première occurrence de la sous-chaîne dans la chaîne
`SUBSTR(chaîne,n,m)`	

- Pour dates et heures
- De conversion de types
- Conditionnelles : agissent sur tous les types de données
  - `COALESCE`,
  - `NULLIF`,
  - `CASE` (expression conditionnelle générique)

Fonction conditionnelle case

- Permet :
  - le rapatriement de plusieurs colonnes de types différents en une seule
  - le passage d'une liste de valeurs en lignes en une liste de valeurs en colonnes (inversion de matrice)
  - la résolution d'alternatives : `CASE` équivaut à `SI...ALORS...SINON`

- Exemple : Nom des employés et commission. Si la commission est NULL on affiche la qualification "Non commissionné" ; si commision = 0 on affiche "Mauvais vendeur" ; sinon, on affiche "Commissionné".

```
SELECT ename AS "Nom", comm,
    CASE
            WHEN comm IS NULL THEN 'Non commissionné'
            WHEN comm=0 THEN 'Mauvais vendeur'
            Else 'Commissionné'
    END "Qualification"
FROM emp
```

Fonction date et heure

`+` :
Ex : `select date '2001-09-28' + 7` -> 2001-10-05
Ex : `select date '2001-09-28' + interval '2 months'` -> 2001-11-28

`-` :
Ex : `select date '2001-09-28' - 7` -> 2001-09-21
Ex : `select date '2001-09-28' - interval '2 months'` -> 2001-07-28

`age(date1, date2)` : calcule l'âge.

Ex : `select age(current_date,'1974-01-29');`

`date_part(text, date)` : partie d'une date.

Ex.: `select date_part('month', current_date)` retourne le n° du mois

`date_trunc(text, date)` : tronque une date.

Ex : `select date_trunc('month', current_date)` retourne le 1er jour du mois de l'année en cours. Ex. si date du jour = 30/10/2018, renvoie 2018-10-01 00:00:00

`CURRENT_DATE` : la date d'aujourd'hui
`CURRENT_TIME` : l'heure exacte courante
`NOW()` : la date et heure à l'instant

Fonctions de conversion

`TO_DATE(date,format)` : va formater la date suivant le format `format`.

Ex. : `to_date('05 Dec 2000', 'DD Mon YYYY')`

`TO_CHAR (date, format)` : convertit un champ de type date ou timestamp en chaîne en fonction de `format`

Ex. : `select to_char(current_date, 'MM');` -> renvoie le n° du mois actuel

`TO_NUMBER(text, text)`

`SELECT TO_NUMBER('12,454.8-', '99G999D9S')` retourne `-12454,8`

On peut utiliser les jokers suivants :

`DAY` (nom complet du jour)
`W` (n° de la semaine dans le mois)
`MM` (n° du mois)
`YYYY` (année, 4 chiffres)
`HH` (heure, de 01 à 12)
`DD` (jour du mois)
`MONTH` (nom complet du mois)
`Q` (trimestre)
`YY` (année, 2 chiffres)

`CAST(expression as type)`

Convertit `expression` vers le type demandé.

Exemple : `SELECT CAST('2003-03-30' as DATE)` retourne 2003-03-30

Sommaire

1. LMD

Basic	*SELECT, FROM, WHERE, DISTINCT, JOIN*
Combine	*UNION, EXCEPT*
Aggregate	*COUNT, SUM, AVG, MIN, MAX, GROUP BY, HAVING*
Nested	*IN, ALL, ANY*
Manipulation	*INSERT, UPDATE, DELETE*
PostgreSQL functions	Scalaires, Conditionnal, Date and Time, Conversion

2. LDD

	CREATE	***ALTER***	***DROP***
Table	R1.05	R1.05	R1.05
Sequence	Today	Today	Today
View	CM2	CM2	CM2
Index	S3-S4	S3-S4	S3-S4

Objectifs

- `CREATE` (création) :

  Tables avec des colonnes de types différents et contraintes (`CREATE TABLE`)
  Séquence (`CREATE SEQUENCE`)
  Vue (`CREATE VIEW`) (Cf. CM2)
  Index (`CREATE INDEX`) (Cf. année prochaine)

- `ALTER` (modification) :

  Tables et contraintes (`ALTER TABLE`)
  Séquence (`ALTER SEQUENCE`)
  Vue (`ALTER VIEW`) (Cf. CM2)
  Index (`ALTER INDEX`) (Cf. année prochaine)

- `DROP` (suppression) :

  Tables et contraintes (`DROP TABLE`)
  Séquence (`DROP SEQUENCE`)
  Vue (`DROP VIEW`) (Cf. CM2)
  Index (`DROP INDEX`) (Cf. année prochaine)

Tables

Cf. CM R1.05 M. Colin

Sequence

Générateur de nombre
⇔ `AUTO_INCREMENT` dans MySQL

Vous devez utiliser une séquence à chaque fois que vous voulez qu'un numéro s'incrémente automatiquement

Des tables différentes peuvent utiliser la même séquence. Sinon, possibilité de créer une séquence pour chaque table ayant un numéro automatique

Après création d'une séquence, les fonctions `nextval`, `currval` et `setval` sont utilisées pour agir sur la séquence.

Syntaxe (partielle) :

```
CREATE SEQUENCE nom [ INCREMENT [ BY ] incrément ][
      MINVALUE valeurmin | NO MINVALUE ]
      [ MAXVALUE valeurmax | NO MAXVALUE ]
      [ START [ WITH ] début ]
      [ OWNED BY { table.colonne | NONE } ]
```

Sequence

```
CREATE SEQUENCE nom [
  INCREMENT …][
    MINVALUE …]
    [ MAXVALUE …]
    [ START …]
    [ OWNED BY …}
]
```

- `INCREMENT BY` *`incrément`* : précise la valeur à ajouter à la valeur courante de la séquence pour créer une nouvelle valeur.
Valeur >0 (seq. croissante) ou <0 (seq. décroissante). 1 est la valeur par défaut.

- `MINVALUE` *`valeurmin`* : détermine la valeur minimale de la séquence.
Si cette clause n'est pas fournie ou si `NO MINVALUE`,
valeur par défaut = 1 (séquence croissante) ou $-2^{63}-1$ (séq. décroissante)

- `MAXVALUE` *`valeurmax`* : détermine la valeur maximale de la séquence. Si cette clause n'est pas fournie ou si `NO MAXVALUE` est specifié,
valeur par défaut = $2^{63}-1$ ou -1 pour les séquences ascendantes et descendantes.

- `START WITH` *`début`* : permet à la séquence de démarrer n'importe où. La valeur de début par défaut est `valeurmin` pour les séquences ascendantes et `valeurmax` pour les séquences descendantes.

- `OWNED BY` *`table.colonne`*, `OWNED BY NONE` : permet d'associer la séquence à une colonne de table spécifique. De cette façon, la séquence sera automatiquement supprimée si la colonne (ou la table entière) est supprimée.
`OWNED BY NONE`, valeur par défaut, indique qu'il n'y a pas d'association.

Sequence

Exemples :

- Créer une séquence ascendante appelée `serie`, démarrant à 101 :

```
CREATE SEQUENCE serie START WITH 101;
```

- Récupérer le prochain numéro de cette séquence :

```
SELECT nextval('serie');
    nextval
    ---------
    101
```

- Récupérer le prochain numéro d'une séquence :

```
SELECT nextval('serie');
    nextval
    ---------
    102
```

- Utiliser cette séquence dans une commande INSERT :

```
INSERT INTO matable (num, val) VALUES (nextval('serie'), 'toto');
    -- Insère 103, 'toto' dans MATABLE

INSERT INTO matable2 (numero, valeur) VALUES (nextval('serie'),
'titi');
    -- Insère 104, 'titi' dans MATABLE2
```

Sequence

Exemples :

- Créer une séquence ascendante appelée `seq_dept` (valeur max = 100), associée à la table `DEPT` :

```
CREATE SEQUENCE seq_dept MAXVALUE 100 OWNED BY DEPT.deptno;
```

- Insérer un nouveau département :

```
INSERT INTO DEPT(deptno, dname, loc) VALUES
(nextval('seq_dept'), 'Sce marketing', 'Annecy');
-- Insère 1, 'Sce marketing', 'Annecy'
```

Suppression de la table `DEPT` :

```
DROP TABLE DEPT;
```

-- Supprime également la séquence car option OWNED BY utilisée, sinon :

```
DROP SEQUENCE seq_dept;
```

Sequence

Il est possible de remplacer une séquence en utilisant un type sérié `SERIAL` ou `BIGSERIAL`

- `SERIAL` (entier à incrémentation automatique variant de 1 à 2147483647). Utiliser un type `INTEGER` dans la clé étrangère.

- `BIGSERIAL` (entier de grande taille à incrémentation automatique variant de 1 à 92233720368547758O7). Utiliser un type `BIGINT` dans la clé étrangère.

Exemple :

```
CREATE SEQUENCE seq_dept; -- Créé la séquence seq_dept
CREATE TABLE DEPT(
   DEPTNO NUMERIC(2) NOT NULL DEFAULT nextval('seq_dept'),
   DNAME VARCHAR(14),
   LOC VARCHAR(13) );
ALTER SEQUENCE seq_dept OWNED BY DEPT.deptno; -- Modifie la séquence pour
l'associer au champ deptno de DEPT
⇔
CREATE TABLE DEPT(
   DEPTNO SERIAL NOT NULL,
   DNAME VARCHAR(14),
   LOC VARCHAR(13) );
```

-- NOTICE: CREATE TABLE will create implicit sequence "dept_deptno_seq" for serial column "dept.deptno"

Sommaire

1. LMD

Basic	*SELECT, FROM, WHERE, DISTINCT, JOIN*
Combine	*UNION, EXCEPT*
Aggregate	*COUNT, SUM, AVG, MIN, MAX, GROUP BY, HAVING*
Nested	*IN, ALL, ANY*
Manipulation	*INSERT, UPDATE, DELETE*
PostgreSQL functions	Scalaires, Conditionnal, Date and Time, Conversion

2. LDD

	CREATE	***ALTER***	***DROP***
Table	R1.05	R1.05	R1.05
Sequence	Today	Today	Today
View	CM2	CM2	CM2
Index	S3-S4	S3-S4	S3-S4

3. LCT

A	Atomicity
C	Consistency
I	Isolation
D	Durability

COMMIT, ROLLBACK, SAVEPOINT

SGBD relationnel et transactionnels

La plupart des SGBD Relationnels sont transactionnels :

- Oracle
- PostgreSQL
- Microsoft SQL Server
- MySQL à condition d'utiliser le moteur InnoDB ou Maria
- MariaDB
- Etc.

Dans ce cas, un langage de contrôle des transactions est disponible : TCL (*Transaction Control Language*)

Permet de gérer des transactions pour les valider ou les annuler.

Propriétés

Les **propriétés ACID** sont quatre propriétés que doit respecter un SGBD dans le traitement des transactions

- ***A****tomicité* : une transaction (ensemble d'instruction) doit soit être complètement validée ou complètement annulée. Une mise à jour ne doit pas être partielle.

- ***C****ohérence* : Une base de données qui est dans un état cohérent au début d'une transaction est laissée dans un état cohérent par la transaction.

- ***I****solation* : une transaction ne peut voir aucune autre transaction en cours d'exécution. Tant qu'une transaction n'est pas validée (commit), elle n'est pas visible par les autres utilisateurs.

- ***D****urabilité* : Après validation d'une transaction, les résultats de celle-ci ne disparaîtront pas.

Ces propriétés garantissent la fiabilité du SGBD

Transactions

Ensembles d'étapes de manipulation de données traitées comme une unité de travail unique

- Permettent de regrouper plusieurs instructions (INSERT, UPDATE et/ou DELETE)
- A utiliser lorsque plusieurs clients accèdent simultanément à des enregistrements à partir de la même table

Réussite de toutes les étapes ou d'aucune d'entre elles

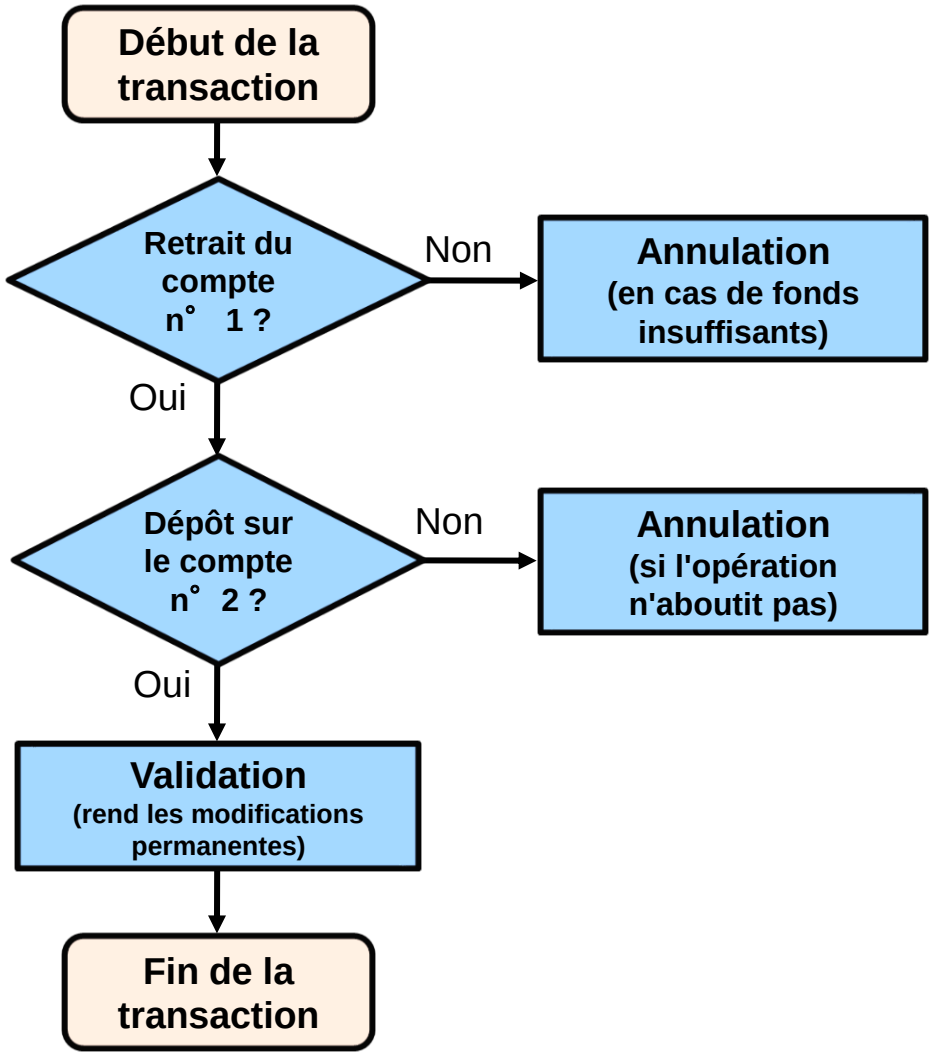
- Validation et enregistrement des modification si toutes les étapes sont correctes
- Annulation si des étapes comportent des erreurs ou sont incomplètes

Tout ordre `INSERT`, `UPDATE` ou `DELETE` utilise une transaction

- Soit une nouvelle (en la démarrant)
- Soit en utilisant celle démarrée précédemment et non encore terminée

Conformité avec les propriétés **ACID**

Transactions



Instructions

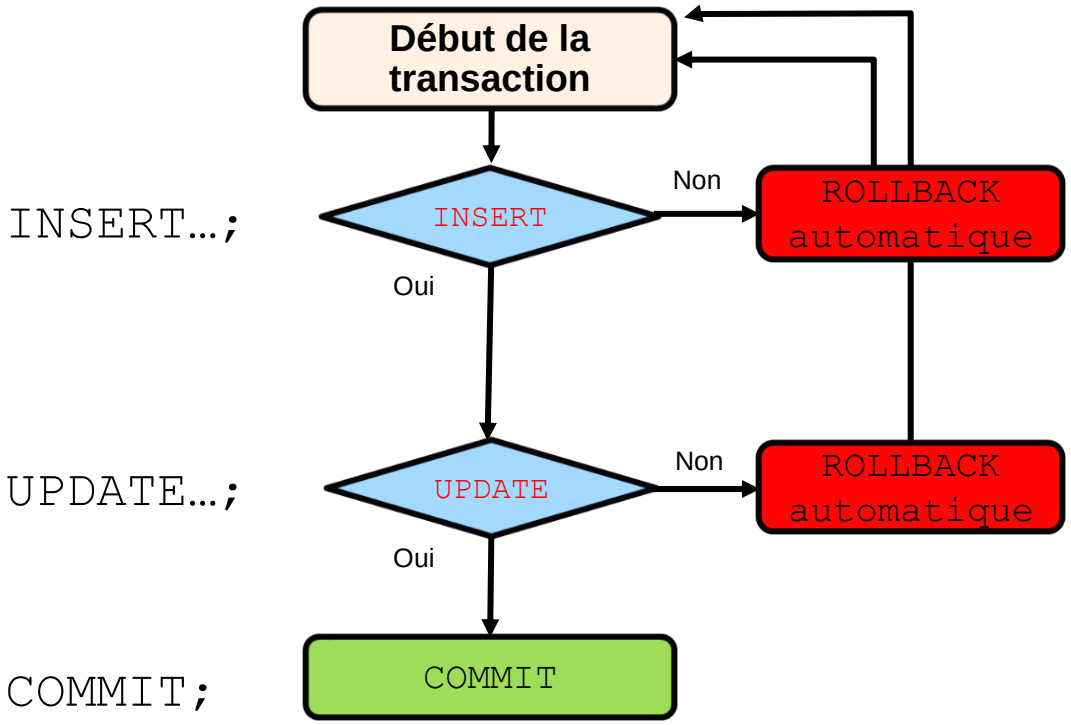
`COMMIT` : Rend permanentes les modifications de la transaction en cours = Validation

`ROLLBACK` : Annule les modifications de la transaction en cours

`SAVEPOINT` : Affecte un emplacement au cours d'une transaction à des fins de référence

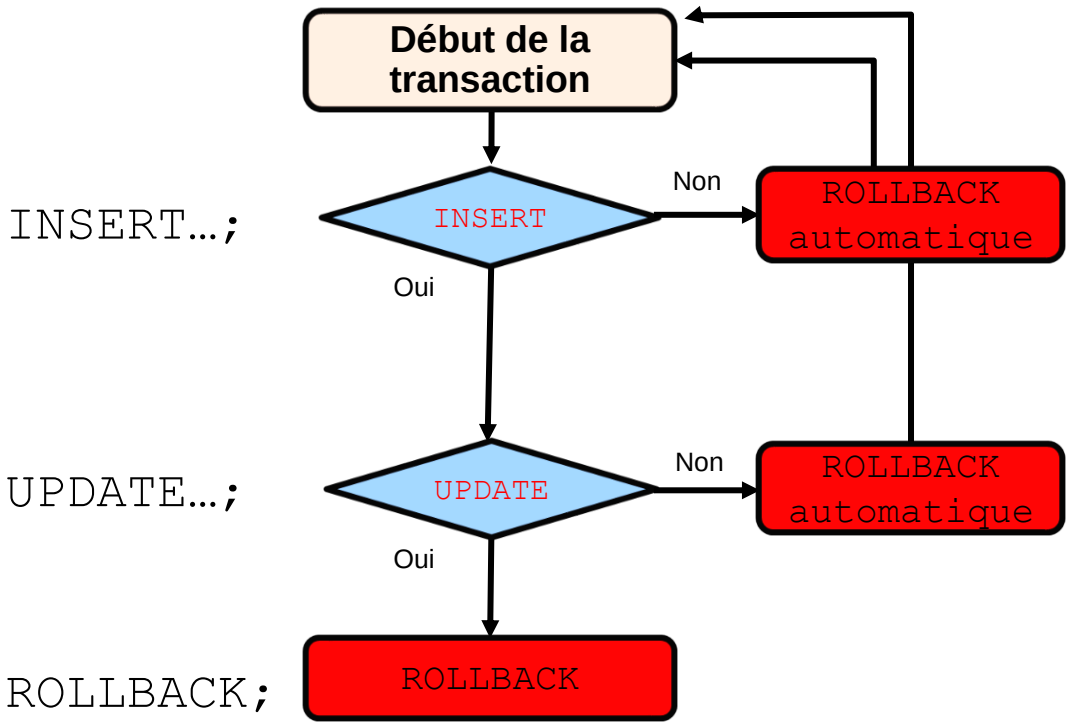
`ROLLBACK TO SAVEPOINT` : Annule les modifications effectuées après le point de sauvegarde

Exemple 1



1. SGBD démarre une nouvelle transaction. ID = 123
2. SGBD tente d'insérer des données. ID transaction = 123. Si erreur, Annulation automatique (Rollback) et fin de la transaction 123. Sinon, on passe à l'instruction suivante.
3. SGBD tente de MAJ des données. ID transaction = 123. Si erreur, Annulation automatique (Rollback) et fin de la transaction 123. Dans ce cas, l'INSERT précédent est également annulé.
4. L'instruction COMMIT est exécutée par l'utilisateur et valide les modifications (INSERT et UPDATE). Fin de la transaction 123.

Exemple 2



1. SGBD démarre une nouvelle transaction. ID = 124
2. SGBD tente d'insérer des données. ID transaction = 123. Si erreur, Annulation automatique (Rollback) et fin de la transaction 124. Sinon, on passe à l'instruction suivante.
3. SGBD tente de MAJ des données. ID transaction = 124. Si erreur, Annulation automatique (Rollback) et fin de la transaction 124. Dans ce cas, l'INSERT précédent est également annulé.
4. L'instruction ROLLBACK est exécutée par l'utilisateur et annule toutes les actions réalisées dans la transaction => INSERT et UPDATE sont annulés. Fin de la transaction 124.

Auto commit

Déterminer quand et comment de nouvelles transactions sont démarrées.

Dans PostgreSQL, mode AUTOCOMMIT est activé par défaut :

- Valide implicitement chaque instruction UPDATE, DELETE ou INSERT en tant que transaction => 1 instruction = 1 transaction.
- Ainsi, si la commande SQL fonctionne, celle-ci est automatiquement validée (commit) et les modifications enregistrées dans la base;
  => Impossible de revenir en arrière.

Remarque : il est possible de désactiver le mode AUTOCOMMIT de PostgreSQL dans les fichiers de paramétrage ou via PgAdmin 4

Dans Oracle, le mode AUTOCOMMIT est désactivé par défaut :

- Les transactions comprennent plusieurs instructions par défaut.
- Vous pouvez terminer une transaction avec COMMIT ou ROLLBACK quand vous voulez.

Dans Vscode, le mode AUTOCOMMIT est activé par défaut :

```
BEGIN … COMMIT
```

Comment participer ?

Copier le lien de participation

1 Allez sur wooclap.com

2 Entrez le code d'événement dans le bandeau supérieur

Code d'événement
GZAZIN

Activer les réponses par SMS

wooclap

105 %

2