

SHAKE THE FUTURE



Bases de Données

SQL

JY Martin

Plan

- 1 Introduction à SQL
- 2 Le Langage de Manipulation des Données - SELECT
- 3 Le Langage de Manipulation des Données - INSERT, UPDATE, DELETE
- 4 Langage de Définition de Données
- 5 Le langage de contrôle
- 6 Autres éléments

SQL = Structured Query Language

Ensemble de langages normalisé

- 1986, 1989 : SQL 86 : Normalisation
- 1989 : SQL 1 - SQL 89
- 1992 : SQL 2 - SQL 92
- 1999 : SQL 3 - SQL 99
- 2003 : SQL 2003
- 2008 : SQL 2008
- 2011 : SQL 2011

LE standard d'accès aux bases de données relationnelles

Que contiennent les versions de SQL

- SQL86 - SQL89 :
 - Requêtes compilées puis exécutés depuis un programme d'application.
 - Opérations ensemblistes restreintes (UNION).
- SQL92 ou SQL2 :
 - Requêtes dynamiques : exécution différée ou immédiate
 - Différents types de jointures : jointure naturelle, jointure externe
 - Opérations ensemblistes : différence (EXCEPT), intersection (INTERSECT)
 - Renommage des attributs dans la clause SELECT
- SQL 99 ou SQL3 :
 - Expressions rationnelles, requêtes récursives, déclencheurs

Que contiennent les versions de SQL

- SQL 2003 :
 - Introduction de la manipulation XML
- SQL 2008 :
 - Ajout de quelques fonctions (MERGE, DIAGNOSTIC, ...) TRUNCATE TABLE, ...
- SQL 2011
 - Ajout des mécanismes d'historisation

SQL : plusieurs fonctionnalités

- Langage de Définition de Données (LDD / DDL)
Aspects création de la base de données
- Langage de Manipulation des Données (LMD / DML)
CRUD = Créer, Prendre, Modifier, Supprimer des données
- Langage de Contrôle de Données (LCD / DCL)
Création de rôles, protection des données

Quelques remarques

- Ce n'est pas parce que SQL est normalisé que tous les SGBD implémentent toutes les fonctionnalités
- Le typage des données peut changer d'un SGBD à l'autre (impact sur les scripts de création)

Conclusion :

- Vous devez savoir sur quel SGBD vous travaillez avant de mettre en oeuvre des scripts / des requêtes
- Testez vos requêtes avant de les mettre en production

Un peu de vocabulaire

Modèle Relationnel

- Relation
- Attribut
- Identifiant
- Lien externe
- Tuple

SGBD

- Table
- Colonne
- Clé primaire
- Clé étrangère
- Ligne

Plan

- 1 Introduction à SQL
- 2 Le Langage de Manipulation des Données - SELECT**
- 3 Le Langage de Manipulation des Données - INSERT, UPDATE, DELETE
- 4 Langage de Définition de Données
- 5 Le langage de contrôle
- 6 Autres éléments

Principe de SQL

- S'appuie sur l'algèbre relationnelle
- La sélection ne comporte qu'une seule instruction
- Toute sélection a pour résultat une table (temporaire)

La Sélection

Sélectionner les colonnes d'une table

= L'opérateur de projection de l'algèbre relationnelle

```
SELECT liste_de_colonnes FROM table
```

La sélection crée une table avec TOUTES les lignes correspondant à la requête

=> il peut y avoir des duplicata (pas de clé dans la table temporaire).

Les lignes sont remontées dans un ordre quelconque.

Exemple de sélection

| Eleve | | |
|-----------------|-----------|--------------|
| <u>Eleve_ID</u> | Eleve_Moy | Eleve_Statut |
| 134 | 13.84 | - |
| 128 | 11.92 | Apprenti |
| 384 | 8.95 | - |

```
SELECT Eleve_ID, Eleve_Statut FROM Eleve
```

| Eleve_ID | Eleve_Statut |
|----------|--------------|
| 134 | - |
| 128 | Apprenti |
| 384 | - |

Le joker *

* permet de sélectionner toutes les colonnes d'une table

```
SELECT * FROM table
```

Exemple de sélection avec *

| Eleve | | |
|-----------------|-----------|--------------|
| <u>Eleve_ID</u> | Eleve_Moy | Eleve_Statut |
| 134 | 13.84 | - |
| 128 | 11.92 | Apprenti |
| 384 | 8.95 | - |

```
SELECT * FROM Eleve
```

| <u>Eleve_ID</u> | Eleve_Moy | Eleve_Statut |
|-----------------|-----------|--------------|
| 134 | 13.84 | - |
| 128 | 11.92 | Apprenti |
| 384 | 8.95 | - |

Remarque sur l'utilisation de *

Considérons la table suivante :

| Eleve | | | | |
|----------|------------|------------|-----------|----------|
| ID | Nom | Prenom | Numero | DateN |
| 4 octets | 128 octets | 128 octets | 12 octets | 4 octets |

- 12000 lignes
- réseau à 100 Mbits/s => 12,5 Mo/s

Penser à RAM, c'est l'argent !!!

| | | |
|-------------------------------------|------------------|----------|
| SELECT * FROM Eleve | 3 312 000 octets | 0.0316 s |
| SELECT ID, Numero, DateN FROM Eleve | 240 000 octets | 0.0023 s |

Conclusion : éviter l'* en version prod.

Colonnes aux valeurs fixes

Il est possible d'utiliser des valeurs fixes dans les colonnes sélectionnées.

Elles seront considérées comme des colonnes ne comportant que cette valeur.

| Eleve | | |
|-----------------|-----------|--------------|
| <u>Eleve_ID</u> | Eleve_Moy | Eleve_Statut |
| 134 | 13.84 | - |
| 128 | 11.92 | Apprenti |
| 384 | 8.95 | - |

```
SELECT Eleve_ID, Eleve_Moy, 'Eleve' FROM Eleve
```


Retirer les lignes identiques

```
SELECT DISTINCT liste_de_colonnes FROM table
```

La requête s'assure que les lignes de la table temporaire générée par le SELECT sont toutes différentes.

Restrictions sur les lignes

Ne sélectionner que certaines lignes d'une table
= L'opérateur de sélection de l'algèbre relationnelle

```
SELECT liste_de_colonnes FROM table WHERE Conditions
```

Conditions donne la condition à vérifier sur chaque ligne.

Condition de restriction sur les lignes

Conditions simples :

- Permet de comparer :
 - Une colonne et une valeur
 - Deux colonnes
- Opérateurs utilisables :
 - = <> < <= > >=
 - AND, OR, NOT

Exemple

| Eleve | | |
|-----------------|-----------|--------------|
| <u>Eleve_ID</u> | Eleve_Moy | Eleve_Statut |
| 134 | 13.84 | - |
| 128 | 11.92 | Apprenti |
| 384 | 8.95 | - |

```
SELECT Eleve_ID, Eleve_Moy FROM Eleve WHERE Eleve_ID > 300
```

| Eleve_ID | Eleve_Statut |
|----------|--------------|
| 384 | 8.95 |

Outils plus complexes de comparaison : BETWEEN

Colonne BETWEEN x AND y

\Leftrightarrow Colonne $\geq x$ AND Colonne $\leq y$

Outils plus complexes de comparaison : IN

Colonne IN (a1, a2, ..., an

\Leftrightarrow Colonne = a1 OR Colonne = A2 OR ... OR Colonne = an

Et bien sûr :

Colonne NOT IN (a1, a2, ..., an)

Les chaînes de caractères

- Apparaissent entre **apostrophes**
- Si une apostrophe apparaît dans la chaîne de caractères, elle doit être **doublée**. Certains SGBD acceptent le \'.
Exemple : aujourd'hui -> 'aujourd'hui'
- La vérification de l'égalité tient compte de la casse (majuscule minuscule)
- Les chaînes de caractères sont ordonnées suivant l'ordre lexicographique

Outils plus complexes de comparaison : LIKE

LIKE est un outil de comparaison de chaînes de caractères disposant de 2 jokers :

- % = toute chaîne de caractères
- _ = tout caractère

Exemples :

- 'A%' = commence par un A, peut comporter d'autres caractères.
- '%N' = se termine par un N
- 'A%N' = commence par A et se termine par N
- 'A%T%N' = commence par A, se termine par N, et comporte un T
- 'A_ION' = A, puis n'importe quel caractère puis ION.

Outils plus complexes de comparaison : ILIKE

Attention : ILIKE n'est pas disponible sur tous les SBGD.

ILIKE fonctionne comme LIKE, mais il ne tient pas compte de la casse.

Exemple

| Eleve | | |
|----------|-----------|--------------|
| Eleve_ID | Eleve_Moy | Eleve_Statut |
| 134 | 13.84 | - |
| 128 | 11.92 | Apprenti |
| 384 | 8.95 | - |

```
SELECT Eleve_ID, Eleve_Statut  
FROM Eleve  
WHERE Eleve_Statut ILIKE 'app'
```

| Eleve_ID | Eleve_Statut |
|----------|--------------|
| 134 | - |
| 128 | Apprenti |
| 384 | - |

Les DATE, TIME et TIMESTAMP

Les données sous forme de **date et heure** sont représentées suivant la norme **ISO 8601**.

- DATE = une date
'AAAA-MM-DD'
- TIME = une heure
'HH :MM :SS'
- TIMESTAMP = une date + une heure
'AAAA-MM-DD HH :MM :SS'

Les autres formats de la forme ISO 8601 sont également utilisables.

Les opérateurs sur les DATE, TIME et TIMESTAMP

- Opérateurs de comparaison classiques (=, ...)
- TO_DATE(unedate, format) : convertit une date au format indiqué
- EXTRACT(unedonnée FROM unedate)
 - CENTURY, DECADE
 - YEAR, MONTH, DAY (jour du mois)
 - DOW (Jour de la semaine)
 - HOUR, MINUTE, SECOND
 - MICROSECONDS
 - TIMEZONE
 - ...

Exemple : EXTRACT(YEAR FROM unTimeZone)

Cas particulier de NULL

NULL traduit une absence d'information.

=> Comparer simplement une colonne avec NULL n'a pas de sens.

On utilise :

Colonne IS NULL
Colonne IS NOT NULL

Les jointures

Jointure = effectuer un produit cartésien sur les lignes de 2 tables.

- Construire une nouvelle table comportant les colonnes des tables jointes
- Population constituée de toutes les combinaisons des tables jointes

Exemple de jointure

| Eleve | | |
|----------|-----------|--------------|
| Eleve_ID | Eleve_Moy | Eleve_Statut |
| 134 | 13.84 | - |
| 128 | 11.92 | Apprenti |
| 384 | 8.95 | - |

| Club | |
|---------|----------|
| Club_ID | Club_Nom |
| 1 | Manga |
| 2 | JdR |

La jointure résultante :

| Eleve_ID | Eleve_Moy | Eleve_Statut | Club_ID | Club_Nom |
|----------|-----------|--------------|---------|----------|
| 134 | 13.84 | - | 1 | Manga |
| 134 | 13.84 | - | 2 | JdR |
| 128 | 11.92 | Apprenti | 1 | Manga |
| 128 | 11.92 | Apprenti | 2 | JdR |
| 384 | 8.95 | - | 1 | Manga |
| 384 | 8.95 | - | 2 | JdR |

SQL et les jointures

```
SELECT liste_colonnes FROM liste_tables WHERE liste_contraintes
```

2 syntaxes :

- SQL 1 :
 - produit cartésien uniquement
 - liste_tables est une liste de tables séparées par des ,
 - les conditions de jointure sont exprimées dans les contraintes
- SQL 2 et + :
 - plusieurs types de jointures
 - liste_tables est exprimé sous forme d'une suite de jointures avec une syntaxe particulière
 - les conditions de jointure sont exprimées pour chaque jointure

Exemple

| Eleve | | |
|----------|-----------|--------------|
| Eleve_ID | Eleve_Moy | Eleve_Statut |
| 134 | 13.84 | - |
| 128 | 11.92 | Apprenti |
| 384 | 8.95 | - |

| Membre | |
|-----------|------------|
| Membre_ID | Club_Nom |
| 134 | Manga |
| 134 | JdR |
| 134 | Astronomie |
| 128 | Manga |
| 128 | Echecs |
| 384 | e-sport |

Objectif : mettre en correspondance Eleve_ID de Eleve et Membre_ID de Membre.

Jointure en SQL1

| Eleve | | |
|-----------------|-----------|--------------|
| <u>Eleve_ID</u> | Eleve_Moy | Eleve_Statut |
| 134 | 13.84 | - |
| 128 | 11.92 | Apprenti |
| 384 | 8.95 | - |

| Membre | |
|------------------|------------|
| <u>Membre_ID</u> | Club_Nom |
| 134 | Manga |
| 134 | JdR |
| 134 | Astronomie |
| 128 | Manga |
| 128 | Echecs |
| 384 | e-sport |

```
SELECT * FROM Eleve, Membre WHERE Eleve_ID=Membre_ID
```

Jointure en SQL2

| Eleve | | |
|-----------------|-----------|--------------|
| <u>Eleve_ID</u> | Eleve_Moy | Eleve_Statut |
| 134 | 13.84 | - |
| 128 | 11.92 | Apprenti |
| 384 | 8.95 | - |

| Membre | |
|------------------|------------|
| <u>Membre_ID</u> | Club_Nom |
| 134 | Manga |
| 134 | JdR |
| 134 | Astronomie |
| 128 | Manga |
| 128 | Echecs |
| 384 | e-sport |

```
SELECT * FROM Eleve INNER JOIN Membre ON (Eleve_ID=Membre_ID)
```

Jointure SQL1 / SQL2 qu'est-ce qui a changé ?

- Les conditions sur la ligne et les conditions de jointure ne sont pas mélangées
- La condition jointure est exprimée à chaque jointure.
Impossible de l'oublier

Les autres formes de jointure de SQL2

- CROSS JOIN : produit cartésien
- [INNER] JOIN table ON (condition_jointure)
Jointure avec une nouvelle table, la condition de jointure est exprimée dans le ON
- NATURAL JOIN table
Jointure avec une nouvelle table, la condition de jointure impose l'égalité des colonnes de table avec toutes les colonnes précédentes ayant le même nom

Les autres formes de jointure de SQL2

- FULL | LEFT | RIGHT [OUTER] JOIN table ON (condition_jointure)
Jointures pleines, droites et gauche.
- FULL | LEFT | RIGHT NATURAL JOIN table
Jointures naturelles pleines, droites et gauche.

Les tables à joindre sont ajoutées les unes à la suite des autres.

CROSS JOIN

CROSS JOIN table

Produit cartésien avec les tables précédentes. Le fonctionnement est identique à la virgule de SQL1.

```
SELECT *  
FROM Eleve  
CROSS JOIN Membre
```

INNER JOIN

INNER JOIN table ON (condition)

La condition à appliquer pour joindre une ligne est précisée dans le ON ().

Exemple INNER JOIN

```
SELECT *  
FROM Eleve  
INNER JOIN Membre ON (Eleve_ID=Membre_ID)  
WHERE Eleve_Moy >= 10
```

| Eleve_ID | Eleve_Moy | Eleve_Statut | Membre_ID | Club_Nom |
|----------|-----------|--------------|-----------|------------|
| 134 | 13.84 | - | 134 | Manga |
| 134 | 13.84 | - | 134 | JdR |
| 134 | 13.84 | - | 134 | Astronomie |
| 128 | 11.92 | Apprenti | 128 | Manga |
| 128 | 11.92 | Apprenti | 128 | Echecs |

NATURAL JOIN

NATURAL JOIN table

Condition de jointure implicite :

- Si une colonne de la table a le même nom qu'une autre colonne (dans les tables précédentes), on vérifie l'égalité de cette colonne avec les colonnes correspondantes.
- Si aucune colonne ne porte le même nom qu'une colonne de la table, on applique un produit cartésien.

Exemple NATURAL JOIN

| Eleve | | |
|-----------------|-----------|--------------|
| <u>Eleve_ID</u> | Eleve_Moy | Eleve_Statut |
| 134 | 13.84 | - |
| 128 | 11.92 | Apprenti |
| 384 | 8.95 | - |

| Membre | |
|-----------------|------------|
| <u>Eleve_ID</u> | Club_Nom |
| 134 | Manga |
| 134 | JdR |
| 134 | Astronomie |
| 128 | Manga |
| 128 | Echecs |
| 384 | e-sport |

```
SELECT * FROM Eleve NATURAL JOIN table WHERE Eleve_ID >=10
```

Exemple NATURAL JOIN

```
SELECT * FROM Eleve NATURAL JOIN table WHERE Eleve_ID >=10
```

| Eleve_ID | Eleve_Moy | Eleve_Statut | Club_Nom |
|----------|-----------|--------------|------------|
| 134 | 13.84 | - | Manga |
| 134 | 13.84 | - | JdR |
| 134 | 13.84 | - | Astronomie |
| 128 | 11.92 | Apprenti | Manga |
| 128 | 11.92 | Apprenti | Echecs |

OUTER JOIN / NATURAL FULL JOIN

Est-il possible de garder les lignes d'une table et compléter par une autre ?

- LEFT OUTER JOIN / NATURAL LEFT OUTER JOIN
 - On garde les lignes de la table de départ et on complète par la nouvelle table.
 - S'il n'y a pas de correspondance, on remplace par NULL
- RIGHT OUTER JOIN / NATURAL RIGHT OUTER JOIN
 - On garde les lignes de la nouvelle table et on complète par la table de départ.
 - S'il n'y a pas de correspondance, on remplace par NULL
- FULL OUTER JOIN / NATURAL FULL OUTER JOIN
 - On garde les lignes de la table de départ et de la nouvelle table

Exemple OUTER JOIN / NATURAL FULL OUTER JOIN

| Eleve | | |
|----------|-----------|--------------|
| Eleve_ID | Eleve_Moy | Eleve_Statut |
| 134 | 13.84 | - |
| 128 | 11.92 | Apprenti |
| 384 | 8.95 | - |

| Membre | |
|----------|----------|
| Eleve_ID | Club_Nom |
| 134 | Manga |
| 134 | JdR |
| 128 | Echecs |
| 128 | JdR |

```
SELECT * FROM Eleve NATURAL LEFT OUTER JOIN Membre
```

| Eleve_ID | Eleve_Moy | Eleve_Statut | Club_Nom |
|----------|-----------|--------------|----------|
| 134 | 13.84 | - | Manga |
| 134 | 13.84 | - | JdR |
| 128 | 11.92 | Apprenti | Echecs |
| 128 | 11.92 | Apprenti | JdR |
| 384 | 8.95 | - | NULL |

Identification des colonnes

Que se passe t'il quand des colonnes portent le même nom ?

| Eleve | | |
|-----------------|-----------|--------------|
| <u>Eleve_ID</u> | Eleve_Moy | Eleve_Statut |

| Membre | |
|-----------------|----------|
| <u>Eleve_ID</u> | Club_Nom |

```
SELECT * FROM Eleve  
INNER JOIN Membre ON (Eleve_ID=Eleve_ID)
```

Solution : préfixer par la table d'appartenance

```
SELECT * FROM Eleve  
INNER JOIN Membre ON (Eleve.Eleve_ID=Membre.Eleve_ID)
```

Utiliser une table plusieurs fois dans la même requête

| Eleve | |
|-----------------|-----------|
| <u>Eleve_ID</u> | Eleve_Nom |

| Club | | | |
|----------------|----------|-----------|-----------|
| <u>Club_ID</u> | Club_Nom | Club_Resp | Club_Tres |

Qui sont les responsables et les trésoriers de chacun des clubs ?

```
SELECT Club_Nom, Eleve_Nom, Eleve_Nom  
FROM Club  
INNER JOIN Eleve ON (Club_Resp=Eleve_ID)  
INNER JOIN Eleve ON (Club_Tres=Eleve_ID)
```


Utiliser une table plusieurs fois dans la même requête... Et si...

| Club | | | |
|----------------|----------|-----------|-----------|
| <u>Club_ID</u> | Club_Nom | Club_Resp | Club_Tres |

| Eleve_Resp | |
|-----------------|-----------|
| <u>Eleve_ID</u> | Eleve_Nom |

| Eleve_Tres | |
|-----------------|-----------|
| <u>Eleve_ID</u> | Eleve_Nom |

Qui sont les responsables et les trésoriers de chacun des clubs ?

```
SELECT Club_Nom, Eleve_Resp.Eleve_Nom, Eleve_Tres.Eleve_Nom
FROM Club
INNER JOIN Eleve_Resp ON (Club_Resp=Eleve_Resp.Eleve_ID)
INNER JOIN Eleve_Tres ON (Club_Tres=Eleve_Tres.Eleve_ID)
```

Utiliser une table plusieurs fois dans la même requête

| Eleve | |
|-----------------|-----------|
| <u>Eleve_ID</u> | Eleve_Nom |

| Club | | | |
|----------------|----------|-----------|-----------|
| <u>Club_ID</u> | Club_Nom | Club_Resp | Club_Tres |

Qui sont les responsables et les trésoriers de chacun des clubs ?

Solution : renommer les tables

```
SELECT Club_Nom, Eleve_Resp.Eleve_Nom, Eleve_Tres.Eleve_Nom
FROM Club
INNER JOIN Eleve Eleve_Resp ON (Club_Resp=Eleve_Resp.Eleve_ID)
INNER JOIN Eleve Eleve_Tres ON (Club_Tres=Eleve_Tres.Eleve_ID)
```

Simplification de l'écriture des jointures

Jointure :

table1 INNER JOIN table2 ON (condition_de_jointure)

Que se passe t'il si la condition de jointure porte sur une colonne qui porte le même nom dans les 2 tables ?

table1 INNER JOIN table2 ON (table1.une_colonne = table2.une_colonne)

Simplification :

table1 INNER JOIN table2 USING (une_colonne)

Opérations ensemblistes

Les opérations ensemblistes permettent de faire des opérations sur les tables temporaires, résultats de requêtes SQL.

Contrainte : les schémas des 2 tables temporaires concaténées doivent être identiques.

Les opérations :

- UNION : concatène les 2 tables
- EXCEPT : supprime de la première tables les lignes de la 2e table
- INTERSECT : prend les lignes communes aux 2 tables - non standart

Exemple d'opération ensembliste

```
(SELECT Club_Nom, Eleve.Eleve_Nom  
FROM Club  
INNER JOIN Eleve ON (Club_Resp=Eleve.Eleve_ID))  
UNION  
(SELECT Club_Nom, Eleve.Eleve_Nom  
FROM Club  
INNER JOIN Eleve ON (Club_Tres=Eleve.Eleve_ID))
```

Imbrication de requêtes

Il est également possible d'utiliser le résultat d'une requête dans une autre requête.

- Ceci fonctionne exclusivement avec le **IN**.
- La requête imbriquée ne peut avoir qu'une seule colonne résultat
- Les requêtes imbriquées peuvent utiliser les tables de la requête qui les imbrique.

Inbrication de requêtes

```
SELECT Club_Nom, Eleve.Eleve_ID  
FROM Club  
INNER JOIN Eleve ON (Club_Resp=Eleve.Eleve_ID)  
WHERE Eleve.Personne_ID IN  
(SELECT Personne_ID FROM Personne WHERE Personne_Nom LIKE 'A%')
```

Autres mécanismes d'imbrication : ANY

ANY permet de comparer la valeur d'un attribut à tous les résultats d'une requête. Si la comparaison est vraie **pour l'une au moins des valeurs**, la condition est vraie.

```
SELECT Fourniture_Nom  
FROM Fourniture  
WHERE Fourniture_Prix < ANY  
(SELECT Fourniture_Prix FROM Fourniture WHERE Fourniture_Type = 'BUREAU')
```

Nom de la fourniture telle qu'il existe une fourniture de type bureau dont le prix est inférieur à celle de la fourniture.

Supérieur

Autres mécanismes d'imbrication : ALL

ALL permet de comparer la valeur d'un attribut à tous les résultats d'une requête. Si la comparaison est vraie **pour toutes les valeurs**, la condition est vraie.

```
SELECT Fourniture_Nom  
FROM Fourniture  
WHERE Fourniture_Prix < ALL  
(SELECT Fourniture_Prix FROM Fourniture WHERE Fourniture_Type = 'BUREAU')
```

Nom de la fourniture telle que son prix est inférieur à toutes les fournitures de type bureau.

Autres mécanismes d'imbrication : EXISTS

EXISTS est vraie si la requête imbriquée possède au moins une ligne comme résultat.

```
SELECT Fourniture_Nom  
FROM Fourniture  
WHERE EXISTS (SELECT Fourniture_Prix FROM Fourniture WHERE  
Fourniture_Type = 'BUREAU')
```

Agrégation : GROUP BY

L'agrégation consiste à regrouper des lignes suivant les valeurs d'une ou plusieurs colonnes. L'objectif est d'appliquer certaines fonctions sur ces sous-groupes de lignes.

```
SELECT informations  
FROM liste_de_tables  
WHERE conditions  
GROUP BY critère
```

Agrégation : exemple

| Eleve | | |
|-----------------|--------|-----|
| <u>Eleve_ID</u> | Groupe | Moy |
| 134 | A | 15 |
| 128 | A | 13 |
| 384 | B | 16 |
| 256 | A | 17 |
| 312 | C | 13 |
| 192 | B | 12 |

Moyenne de chacun des groupes ?

Agrégation : exemple

Moyenne de chacun des groupes ?

| Eleve_ID | Groupe | Moy |
|----------|--------|-----|
| 134 | A | 15 |
| 128 | A | 13 |
| 256 | A | 17 |
| 384 | B | 16 |
| 192 | B | 12 |
| 312 | C | 13 |

->

| Groupe | avg |
|--------|-----|
| A | 15 |
| B | 14 |
| C | 13 |

```
SELECT Groupe, AVG(Moy) FROM Eleve GROUP BY Groupe
```

Agrégation : les opérateurs de calcul

- AVG : moyenne
- COUNT : nombre de lignes
- MAX : maximum
- MIN : minimum
- SUM : somme

Agrégation - Remarques

- Les colonnes qui apparaissent dans le résultat du SELECT et qui ne sont pas agrégées (pas de fonction appliquée) **DOIVENT** apparaître dans le GROUP BY
- Le regroupement est fait pour chaque groupe distinct de valeurs des colonnes du GROUP BY

Agrégation - Condition d'application

Une fois les données regroupées, il arrive que l'application d'une fonction de calcul ne soit pas pertinent.

Dans notre exemple, si les notes d'un groupe ne sont que partiellement saisies, est-il pertinent de faire un calcul de la moyenne du groupe ?

```
SELECT informations  
FROM liste_de_tables  
WHERE conditions  
GROUP BY critère  
HAVING condition d'application
```


Ordonnancement : ORDER BY

Par défaut, les lignes sont dans un ordre quelconque.
Possibilité de les ordonner suivant certains critères ?

```
SELECT informations  
FROM liste_de_tables  
WHERE conditions  
ORDER BY critère
```

Ordonnement : critère d'ordonnement

Le critère d'ordonnement est constitué d'une liste de colonnes de la liste d'information du SELECT, avec leur mécanisme d'ordonnement.

- Le tri est fait sur la première colonne mentionnée
- Si 2 lignes ont même valeur de la première colonne, on utilise la deuxième
- puis la troisième
- ...

Ordonnement : critère d'ordonnement

Les mécanisme d'ordonnement sont :

- ASC = ascendant
- DESC = descendant
- si aucun des 2 n'est mentionné, ASC est appliqué.

Exemple d'ordonnement

| Eleve | | |
|-----------------|-----------|--------------|
| <u>Eleve_ID</u> | Eleve_Moy | Eleve_Statut |
| 134 | 13.84 | - |
| 128 | 15.92 | Apprenti |
| 384 | 8.95 | - |
| 255 | 12.34 | Apprenti |

```
SELECT Eleve_Statut, Eleve_Moy  
FROM Eleve  
ORDER BY Eleve_Statut, Eleve_Moy DESC
```

Quelques fonctions utilisables dans les requêtes

- UPPER : met un attribut ou une valeur en majuscule
- LOWER : met un attribut ou une valeur en minuscules
- CAPS : Met une majuscule à chacun des mots d'une chaîne de caractères (non standard)
- CONCAT - || : concaténation
- SUBSTRING - SUBSTR : extrait une sous chaîne
- TRIM : retire les espaces du début et de la fin de la chaîne
LTRIM, RTRIM

Transtypage

Les résultats d'une requête correspondent au schéma de la relation du résultat.

=> les colonnes ET leur domaine de définition définissent le résultat.

Parfois : soucis de typage de données.

-> Comment changer le type d'une colonne ?

Transtypage

CAST (expression as Type)

Convertit une expression en fonction du type indiqué.

NB : le type indiqué doit être compatible

```
SELECT Nom, CAST ( DateNaissance AS VARCHAR ) FROM Personne
```

Plan

- 1 Introduction à SQL
- 2 Le Langage de Manipulation des Données - SELECT
- 3 Le Langage de Manipulation des Données - INSERT, UPDATE, DELETE**
- 4 Langage de Définition de Données
- 5 Le langage de contrôle
- 6 Autres éléments

Les instructions de modification

Trois types d'instructions

- INSERT : ajouter des lignes
- UPDATE : mettre à jour des lignes
- DELETE : supprimer des lignes

Chacune de ces instructions s'applique sur une unique table

Insertion : INSERT

```
INSERT INTO table(attrib1, attrib2, ... attribn) VALUES  
(val1, val2, ... valn)
```

- L'association des éléments se fait en fonction de la position de l'élément dans la liste
- Les attributs de la table non mentionnés et qui ont une valeur par défaut recevront la valeur par défaut
- Les autres attributs de la table non mentionnés dans l'insertion seront mis à NULL
- Si les contraintes d'intégrité ne sont pas respectées, l'insertion provoque une erreur et l'insertion n'est pas effectuée.

Insertion : Autre forme - déconseillée

```
INSERT INTO table  
VALUES (val1, val2, ... valn)
```

L'insertion porte sur **tous les attributs de la table**, dans **l'ordre où ils ont été définis**

Insertion multiple

```
INSERT INTO table(attrib1, attrib2, ... attribn) VALUES  
(val11, val12, ... val1n) ,  
(val21, val22, ... val2n) ,  
...  
(valm1, valm2, ... valmn)
```

- Les insertions sont faites dans l'ordre où elles apparaissent.
- Si l'une des insertions provoque une erreur, aucune des insertions ne sera effectuée.

Exemple insertion

| Personne | | |
|--------------------|--------------|-----------------|
| <u>Personne_ID</u> | Personne_Nom | Personne_Prenom |
| 1 | ALBAN | Jacques |

```
INSERT INTO Personne(Personne_ID, Personne_Nom, Personne_Prenom)  
VALUES ( 2, 'DELILLE', 'Roger' )
```

| Personne | | |
|--------------------|--------------|-----------------|
| <u>Personne_ID</u> | Personne_Nom | Personne_Prenom |
| 1 | ALBAN | Jacques |
| 2 | DELILLE | Roger |

Insertion : variantes

INSERT INTO table(liste_attributs) SELECT liste_attributs FROM ..
WHERE ...

- Toutes les lignes obtenues via la sélection sont insérées dans la table
- Les colonnes résultats du SELECT doivent correspondre au schéma d'insertion

Insertion : retourner la valeur d'une colonne

- Oracle, PostgreSQL

Possibilité d'utiliser l'instruction **RETURNING** pour retourner la valeur d'une colonne (en général la clé primaire) créée par l'INSERT

```
INSERT INTO table (col1, col2, ... coln)
VALUES (val1, val2, ... valn)
RETURNING uneColonne
```

- MySQL

Utilisation de LAST_INSERT_ID

Permet d'avoir la dernière valeur insérée

Mise à jour : UPDATE

UPDATE Table
SET ListeAffectation
[WHERE ListeConditions]

- Chacune des lignes respectant la condition est modifiée en appliquant la ListeAffectation
- Si une contrainte d'intégrité n'est pas respectée, la mise à jour provoque une erreur. Aucune ligne n'est modifiée.
- ListeAffectation est une liste d'affectation de valeurs aux colonnes de la table : colonne1=valeur1, colonne2=valeur2, ...

Exemple de mise à jour

| Personne | | |
|--------------------|--------------|-----------------|
| <u>Personne_ID</u> | Personne_Nom | Personne_Prenom |
| 1 | ALBAN | Jacques |
| 2 | DELILLE | Roger |

```
UPDATE Personne SET Personne_Nom='Alexis' WHERE Personne_ID=2
```

| Personne | | |
|--------------------|--------------|-----------------|
| <u>Personne_ID</u> | Personne_Nom | Personne_Prenom |
| 1 | ALBAN | Jacques |
| 2 | DELILLE | Alexis |

Jointures dans les mise à jour

Il est rare que pour une mise à jour on n'ait pas à s'appuyer sur des données situées dans une autre table.

| Personne | | | Membre | |
|--------------------|---------|---------|--------------------|----------|
| <u>Personne_ID</u> | Nom | Prenom | <u>Personne_ID</u> | Club_Nom |
| 1 | ALBAN | Jacques | 1 | Manga |
| 2 | DELILLE | Alexis | 1 | JdR |
| | | | 2 | Echecs |
| | | | 2 | JdR |

Alexis DELILLE n'est pas membre du club JdR mais du club Manga.

Jointures dans les mise à jour

- requêtes imbriquées :

```
UPDATE Membre SET Club_Nom='Manga'  
WHERE Personne_ID IN (  
    SELECT Personne_ID FROM Personne  
    WHERE Nom='DELILLE' AND Prenom='Alexis')  
AND Club_Nom = 'JdR'
```

- jointure explicite :

```
UPDATE Membre SET Club_Nom='Manga'  
FROM Personne  
WHERE Membre.Personne_ID=Personne.Personne_ID  
AND Nom='DELILLE' AND Prenom='Alexis'  
AND Club_Nom = 'JdR'
```

Suppression : DELETE

```
DELETE FROM table  
[WHERE ListeConditions]
```

- Chacune des lignes respectant la condition est supprimée
- Si une contrainte d'intégrité n'est pas respectée, la suppression provoque une erreur. Aucune ligne n'est supprimée.

Exemple de suppression

| Personne | | |
|--------------------|--------------|-----------------|
| <u>Personne_ID</u> | Personne_Nom | Personne_Prenom |
| 1 | ALBAN | Jacques |
| 2 | DELILLE | Alexis |

DELETE FROM Personne WHERE Personne_ID=2

| Personne | | |
|--------------------|--------------|-----------------|
| <u>Personne_ID</u> | Personne_Nom | Personne_Prenom |
| 1 | ALBAN | Jacques |

Jointures dans les suppressions

De la même manière que les mise-à-jour, les suppressions se font parfois sur la base d'informations situées dans d'autres tables.

| Personne | | | Membre | |
|--------------------|---------|---------|--------------------|----------|
| <u>Personne_ID</u> | Nom | Prenom | <u>Personne_ID</u> | Club_Nom |
| 1 | ALBAN | Jacques | 1 | Manga |
| 2 | DELILLE | Alexis | 1 | JdR |
| | | | 2 | Echecs |
| | | | 2 | JdR |

Alexis DELILLE n'est plus membre du club JdR.

Jointures dans les suppressions

- requêtes imbriquées :

```
DELETE FROM Membre  
WHERE Personne_ID IN (  
    SELECT Personne_ID FROM Personne  
    WHERE Nom='DELILLE' AND Prenom='Alexis')  
AND Club_Nom = 'JdR'
```

- jointure explicite :

```
DELETE FROM Membre  
USING Personne  
WHERE Membre.Personne_ID=Personne.Personne_ID  
AND Nom='DELILLE' AND Prenom='Alexis'  
AND Club_Nom = 'JdR'
```

Notion de transaction

Un utilisateur est rarement seul à exploiter une base de données.
=> Lors d'une modification il peut y avoir des conflits entre utilisateurs.

Comment gérer une modification simultanée d'une information par plusieurs utilisateurs ?

Notion de transaction : pourquoi ?

| Personne | | |
|--------------------|---------|--------|
| <u>Personne_ID</u> | Nom | Prenom |
| 1 | ALBAN | Roger |
| 2 | DELILLE | Alexis |

Utilisateur 1 :

- UPDATE Personne SET Nom='ALBAN' WHERE Prenom='Alexis' ;
- UPDATE Personne SET Nom='DELILLE' WHERE Prenom='Roger' ;

| Personne | | |
|--------------------|---------|--------|
| <u>Personne_ID</u> | Nom | Prenom |
| 1 | DELILLE | Roger |
| 2 | ALBAN | Alexis |

Notion de transaction : pourquoi ?

| Personne | | |
|--------------------|---------|--------|
| <u>Personne_ID</u> | Nom | Prenom |
| 1 | ALBAN | Roger |
| 2 | DELILLE | Alexis |

Utilisateur 2 :

- UPDATE Personne SET Prenom='Alexis' WHERE Nom='ALBAN' ;
- UPDATE Personne SET Prenom='Roger' WHERE Nom='DELILLE' ;

| Personne | | |
|--------------------|---------|--------|
| <u>Personne_ID</u> | Nom | Prenom |
| 1 | ALBAN | Alexis |
| 2 | DELILLE | Roger |

Notion de transaction : pourquoi ?

| Personne | | |
|--------------------|---------|--------|
| <u>Personne_ID</u> | Nom | Prenom |
| 1 | ALBAN | Roger |
| 2 | DELILLE | Alexis |

```
UPDATE Personne SET Nom='ALBAN' WHERE Prenom='Alexis';  
UPDATE Personne SET Nom='DELILLE' WHERE Prenom='Roger';  
UPDATE Personne SET Prenom='Alexis' WHERE Nom='ALBAN';  
UPDATE Personne SET Prenom='Roger' WHERE Nom='DELILLE';
```

| Personne | | |
|--------------------|---------|--------|
| <u>Personne_ID</u> | Nom | Prenom |
| 1 | DELILLE | Roger |
| 2 | ALBAN | Alexis |

Notion de transaction : pourquoi ?

| Personne | | |
|--------------------|---------|--------|
| <u>Personne_ID</u> | Nom | Prenom |
| 1 | ALBAN | Roger |
| 2 | DELILLE | Alexis |

```
UPDATE Personne SET Prenom='Alexis' WHERE Nom='ALBAN';  
UPDATE Personne SET Prenom='Roger' WHERE Nom='DELILLE';  
UPDATE Personne SET Nom='ALBAN' WHERE Prenom='Alexis';  
UPDATE Personne SET Nom='DELILLE' WHERE Prenom='Roger';
```

| Personne | | |
|--------------------|---------|--------|
| <u>Personne_ID</u> | Nom | Prenom |
| 1 | ALBAN | Alexis |
| 2 | DELILLE | Roger |

Notion de transaction : pourquoi ?

| Personne | | |
|--------------------|---------|--------|
| <u>Personne_ID</u> | Nom | Prenom |
| 1 | ALBAN | Roger |
| 2 | DELILLE | Alexis |

```
UPDATE Personne SET Nom='ALBAN' WHERE Prenom='Alexis';  
UPDATE Personne SET Prenom='Alexis' WHERE Nom='ALBAN';  
UPDATE Personne SET Nom='DELILLE' WHERE Prenom='Roger';  
UPDATE Personne SET Prenom='Roger' WHERE Nom='DELILLE';
```

| Personne | | |
|--------------------|-------|--------|
| <u>Personne_ID</u> | Nom | Prenom |
| 1 | ALBAN | Alexis |
| 2 | ALBAN | Alexis |

Notion de transaction : D'où vient le problème ?

Tant que les requêtes s'effectuent séparément, pas de soucis.
Dès que les requêtes s'imbriquent ça ne fonctionne plus.

Chaque bloc de requête ne peut fonctionner que de manière indivisible.

Solution : Faire en sorte que chaque bloc de requête fonctionne comme s'il était seul.

= **Transaction.**

Transaction

BEGIN ;

...

COMMIT ;

Effectue les requête et
enregistre.

BEGIN ;

...

ROLLBACK ;

Effectue les requête et annule.

Transaction

```
BEGIN ;  
UPDATE Personne SET Nom='ALBAN' WHERE Prenom='Alexis' ;  
UPDATE Personne SET Nom='DELILLE' WHERE Prenom='Roger' ;  
COMMIT ;
```

```
BEGIN ;  
UPDATE Personne SET Prenom='Alexis' WHERE Nom='ALBAN' ;  
UPDATE Personne SET Prenom='Roger' WHERE Nom='DELILLE' ;  
COMMIT ;
```


Transaction

```
UPDATE Personne SET Nom='ALBAN' WHERE Prenom='Alexis';  
UPDATE Personne SET Nom='DELILLE' WHERE Prenom='Roger';  
UPDATE Personne SET Prenom='Alexis' WHERE Nom='ALBAN';  
UPDATE Personne SET Prenom='Roger' WHERE Nom='DELILLE';
```

```
UPDATE Personne SET Prenom='Alexis' WHERE Nom='ALBAN';  
UPDATE Personne SET Prenom='Roger' WHERE Nom='DELILLE';  
UPDATE Personne SET Nom='ALBAN' WHERE Prenom='Alexis';  
UPDATE Personne SET Nom='DELILLE' WHERE Prenom='Roger';
```

Propriétés ACID

Les transactions respectent les 4 propriétés ACID

- Atomicité : on exécute tout ou rien
- Cohérence : on passe d'un état cohérent de la base à un autre état cohérent
- Isolation : L'exécution simultanée de 2 transactions produit le même résultat que leur exécution l'une après l'autre
- Durabilité : si une transaction est confirmée, elle est enregistrée dans la base

Plan

- 1 Introduction à SQL
- 2 Le Langage de Manipulation des Données - SELECT
- 3 Le Langage de Manipulation des Données - INSERT, UPDATE, DELETE
- 4 Langage de Définition de Données**
- 5 Le langage de contrôle
- 6 Autres éléments

Langage de Définition de Données

- Permet de créer une base de données
- Permet de créer l'infrastructure de la base de données
- Permet de modifier cette infrastructure

Techniquement, mémorise des informations dans les méta-données du Système de Gestion de Base de Données.

Attention, un certain nombre de commandes dépendent du type de base dans lequel ils sont exécutés.

Les commandes de base

- CREATE = création
 - CREATE DATABASE : créer une base
 - CREATE TABLE : créer une table
 - CREATE VIEW : créer une vue particulière sur les données à partir d'un SELECT
- DROP = supprimer
 - DROP DATABASE : supprimer une base de données
 - DROP TABLE : supprimer une table
 - DROP VIEW : supprimer une vue
- ALTER = modifier
 - ALTER DATABASE / TABLE / VIEW

Commandes sur les bases de données

- **CREATE DATABASE** : créer une base
Options pour définir le possesseur, l'encodage.
- **DROP DATABASE** : supprimer une base
Supprime la base : son schéma, ses données, ...
- **ALTER DATABASE** : modifier une base
permet de changer son possesseur, son nom, ...

Commandes sur les tables

- **CREATE TABLE** : créer une table
Permet de définir tous les schéma de la table : ses colonnes, clé primaire, étrangères, ...
- **DROP TABLE** : supprimer une table
Supprime la table : son schéma, ses données, ...
Attention, ne peut pas s'effectuer si cela romp des contraintes d'intégrité
- **ALTER TABLE** : modifier une table
permet de changer son possesseur, son nom, ses colonnes, ...

Création d'une table

```
CREATE TABLE nom_table (  
  une_colonne,  
  une_colonne,  
  ...  
  contraintes sur la table  
)
```


Création d'une colonne dans une table

nom_colonne type_colonne [Contraintes]

Les types de données : syntaxe dépendant du type de SGBD, de sa version

| | ORACLE | PostgreSQL | MySQL-MariaDB |
|-----------|-----------|-------------------|---------------|
| Entier | NUMBER | INTEGER | int |
| Réel | FLOAT | NUMERIC | float |
| Caractère | CHARACTER | CHARACTER | char |
| Chaîne | VARCHAR2 | CHARACTER VARYING | varchar |
| Texte | - | TEXT | text |
| Date | DATE | DATE | date |
| Time | TIME | TIME | time |
| Timestamp | TIMESTAMP | TIMESTAMP | timestamp |
| ... | ... | ... | ... |

Les contraintes sur une colonne

- NOT NULL : La colonne ne peut contenir de valeur nulle
- PRIMARY KEY : La colonne est la clé primaire (identifiant de la relation)
- UNIQUE : La colonne est une clé candidate, mais n'est pas l'identifiant
- REFERENCES nom-table [(nom-col)] [action] : La colonne possède un lien externe vers une autre colonne, probablement d'une autre table
- CHECK (condition) : Une condition doit être vérifiée sur cette colonne pour toute ligne de la table

Les contraintes sur une table

- PRIMARY KEY (liste_de_colonnes) : permet de déclarer une clé primaire, en particulier quand celle-ci est composée de plusieurs colonnes
- UNIQUE (liste_de_colonnes) : permet de déclarer une clé candidate qui n'est pas l'identifiant
- FOREIGN KEY (liste_de_colonnes) REFERENCES nom-table [(Autre_liste_de_colonnes)] [action] : permet de déclarer une clé étrangère (lien externe)
- CHECK (condition) : permet de vérifier que chaque ligne vérifie la condition indiquée.

NULL / NOT NULL : une colonne peut ou non comporter la valeur NULL.

- Une colonne déclarée NOT NULL provoquera une erreur d'intégrité si on essaie de mettre NULL dans cette colonne.
- Une colonne appartenant à la clé primaire **DOIT** comporter la mention NOT NULL
- S'il n'y a pas de mention à NULL ou NOT NULL, par défaut il est possible de mettre la valeur NULL dans la colonne.

PRIMARY KEY : La clé primaire.

- Si la mention est portée sur une colonne, alors la clé primaire est cette colonne
- Si la clé primaire est constituée de plusieurs colonnes, elle est obligatoirement indiquée en fin de table.
- La mention PRIMARY KEY ne peut être présente plus d'une fois dans une table
- Toutes les colonnes de la clé primaire **DOIVENT** comporter la mention NOT NULL
- une table **DEVRAIT TOUJOURS** comporter une clé primaire.

UNIQUE : une clé candidate

- Si la mention est portée sur une colonne, alors les valeurs de cette colonne sont toutes différentes.
- Si la clé unique est constituée de plusieurs colonnes, elle est obligatoirement indiquée en fin de table.
- Toutes les colonnes de la clé unique DOIVENT comporter la mention NOT NULL
- Il peut y avoir autant de clés uniques que nécessaire. Il peut aussi ne pas y en avoir.
- Une clé unique ne peut pas être égale à la clé primaire.

REFERENCES / FOREIGN KEY ... REFERENCES : une clé étrangère

Par défaut, la référence est la clé primaire de l'autre table. Si ce n'est pas la clé primaire, ce soit être une clé unique.

- Si la mention REFERENCES est portée sur une colonne, alors les valeurs de la colonne sont soumises à leur existence dans la table référente.
- Si la référence porte sur plus d'une colonne, celle-ci doit être déclarée en fin de table avec FOREIGN KEY.
- La table référencée **DOIT** déjà avoir été définie.

DEFAULT

Permet de définir une valeur par défaut.

Si la colonne n'est pas mentionnée lors d'une insertion, la colonne prendra la valeur par défaut.

CHECK

Permet de vérifier, en cas d'insertion ou de modification, qu'une colonne ou un ensemble de colonnes respecte une contrainte prédéfinie.

- Si CHECK est mentionné sur une colonne, on la contrainte porte sur la colonne en question
- Si CHECK est mentionné n fin de table, la contrainte peut porter sur n'importe quelle colonne ou ensemble de colonnes de la table.

Exemple de création de tables

```
CREATE TABLE Eleve (  
    id INTEGER NOT NULL PRIMARY KEY,  
    nom CHARACTER VARYING(120) NOT NULL  
);  
CREATE TABLE Inscription (  
    id INTEGER NOT NULL REFERENCES Eleve  
    Annee INTEGER NOT NULL,  
    PRIMARY KEY (id, Annee)  
);  
CREATE TABLE InscriptionMatiere (  
    id INTEGER NOT NULL REFERENCES Eleve  
    Annee INTEGER NOT NULL,  
    Matiere CHARACTER VARYING(24) NOT NULL,  
    PRIMARY KEY (id, Annee, Matiere)  
    FOREIGN KEY (id, Annee) REFERENCES Inscription  
);
```

Supprimer une table

```
DROP TABLE table
```

Attention aux clés étrangères encore valide **ça n'eserai pas possible à supprimer**

Modifier une table

ALTER TABLE table

- RENAME TO nouveau-nom-table
- ADD COLUMN nom-col type-col [contraintes]
- MODIFY COLUMN nom-col [type-col] [SET contraintes]
- DROP COLUMN nom-col [CASCADE CONSTRAINTS]
- RENAME COLUMN old-name TO new-name

La syntaxe peut légèrement varier en fonction des systèmes de gestion de bases de données.

Intégrité référentielle

- Ajout d'une ligne
 - La clé primaire ne doit pas déjà exister
 - Les clés unique doivent être unique
 - Les colonnes soumises aux clés étrangères doivent avoir leur équivalent dans les tables concernées
 - les contraintes (CHECK) doivent être valides


Intégrité référentielle

- Modification d'une ligne
 - La clé primaire ne doit pas déjà exister
 - Les clés unique doivent être unique
 - Les colonnes soumises aux clés étrangères doivent avoir leur équivalent dans les tables concernées
 - Les colonnes cibles de clés étrangères ne doivent pas rompre de référence
 - les contraintes (CHECK) doivent être valides

Intégrité référentielle

- Suppression d'une ligne
 - Les colonnes cibles de clés étrangères ne doivent pas rompre de référence

Exemples d'Intégrités référentielles



| Personne | | | Eleve | |
|--------------------|---------|--------|---------------------|-------------|
| <u>Personne_ID</u> | Nom | Prenom | <u>Eleve_Numero</u> | Personne_ID |
| 1 | ALBAN | Roger | 190235E | 1 |
| 2 | DELILLE | Alexis | 190118C | 2 |

- INSERT INTO Eleve(Eleve_ID, Personne_ID) VALUES (2, 3)
=> Erreur (pas de Personne_ID=3)
- UPDATE Personne SET Personne_ID=4 WHERE
Personne_ID=1
=> Erreur (Personne_ID=1 référencée)
- DELETE FROM Personne WHERE Prenom='Roger'
=> Erreur (Personne_ID=1 est référencé dans Eleve)

Comment propager les informations ?

Idée : pour certaines contraintes d'intégrité, au lieu de générer une erreur, peut-on propager l'action ?

Par exemple : supprimer une ligne dans une table supprime toutes les lignes qui y font référence.

REFERENCIAL TRIGGERED ACTION

- Basé sur la notion de TRIGGER
- Définit le comportement d'une contrainte d'intégrité en fonction des actions effectuées

REFERENCIAL TRIGGERED ACTION

2 circonstances

- ON UPDATE
- ON DELETE

4 Comportements

- RESTRICT
- CASCADE
- SET DEFAULT
- SET NULL

Lorsque la cible d'une clé étrangère reçoit l'événement concerné, on applique le comportement désigné sur la source de la clé étrangère.

REFERENCIAL TRIGGERED ACTION

```
CREATE TABLE Personne (  
    Personne_ID INTEGER NOT NULL PRIMARY KEY,  
    nom CHARACTER VARYING(120) NOT NULL,  
    prenom CHARACTER VARYING(120)  
);
```

```
CREATE TABLE Eleve (  
    Eleve_ID INTEGER NOT NULL PRIMARY KEY  
    Personne_ID INTEGER REFERENCES Personne  
        ON UPDATE CASCADE  
        ON DELETE SET NULL  
);
```

Exemples REFERENCIAL TRIGGERED ACTION

- INSERT INTO Eleve(Eleve_ID, Personne_ID) VALUES (2, 3)
=> Erreur (pas de Personne_ID=3)
- UPDATE Personne SET Personne_ID=4 WHERE
Personne_ID=1
=> Le Personne_ID=1 est remplacé par Personne_ID=4 dans
Eleve
- DELETE FROM Personne WHERE Prenom='Roger'
=> Le Personne_ID=4 est maintenant remplacé par NULL
dans Eleve

Plan

- 1 Introduction à SQL
- 2 Le Langage de Manipulation des Données - SELECT
- 3 Le Langage de Manipulation des Données - INSERT, UPDATE, DELETE
- 4 Langage de Définition de Données
- 5 Le langage de contrôle**
- 6 Autres éléments

Objectif

Les données des bases de données ne doivent pas être toutes accessibles à tout utilisateur.

N'importe qui ne doit pas pouvoir se connecter à une base de données.

Il faut donc des outils de protection.

Créer des utilisateurs

Les Systèmes de Gestion de bases de données ont un utilisateur par défaut. Cet utilisateur a un statut d'administrateur. Il faut donc réserver ce compte pour des utilisations exceptionnelles.

Créer/Révoquer un utilisateur :

- CREATE ROLE un_utilisateur ... / DROP ROLE un_utilisateur
- CREATE USER un_utilisateur ... / DROP USER un_utilisateur

Les droits d'accès d'un utilisateur

- LOGIN / NOLOGIN
- SUPERUSER / NOSUPERUSER
- CREATEDB / NOCREATEDB
- VALID UNTIL 'date et heure
- PASSWORD 'mot de passe'

Exemple de création d'utilisateur

```
CREATE USER prweb WITH LOGIN NOSUPERUSER CREATEDB PASSWORD  
'info2019prweb' ;
```

Possesseurs

Par défaut, les bases, tables ... appartiennent à celui qui les créent. Il peut modifier le schema et consulter les données des éléments qui lui appartiennent.

Vous pouvez changer ces droits :

```
ALTER DATABASE ma_base OWNER TO un_utilisateur
```

```
ALTER TABLE ma_table OWNER TO un_utilisateur
```

Accorder des droits

Il est aussi possible d'attribuer/révoquer des droits à d'autres utilisateurs. La finesse du réglage dépend du Système de Gestion de Bases de Données que vous utilisez.

GRANT operation ON objet TO utilisateur [WITH GRANT OPTION]

Quels droits ?

Voici un exemple de droits que vous pouvez accorder sur une table :

- SELECT
- INSERT
- UPDATE
- DELETE
- TRUNCATE (vider une table)
- ...

Exemple de définition de droits

```
ALTER DATABASE personnel OWNER TO persouser ;  
GRANT SELECT ON ALL TABLES TO autreperso ;  
GRANT ALL PRIVILEGES ON TABLE personne TO autreperso ;  
ALTER TABLE personne OWNER TO altuser ;
```

Plan

- 1 Introduction à SQL
- 2 Le Langage de Manipulation des Données - SELECT
- 3 Le Langage de Manipulation des Données - INSERT, UPDATE, DELETE
- 4 Langage de Définition de Données
- 5 Le langage de contrôle
- 6 Autres éléments

Apostrophes et guillemets

- Les chaînes de caractères sont **TOUJOURS** entre apostrophes
- Dans une chaîne de caractères, une apostrophe est doublée
- Les guillemets sont utilisés pour les noms de base, de table, ...
Ils permettent d'obliger le nom à respecter une écriture stricte

Utilisation des guillemets

Par défaut, les noms des éléments ne sont pas sensible à la casse parce que convertis systématiquement.

Exemple : Sur PostgreSQL, les noms des éléments sont convertis en minuscule.

Et donc :

- Eleve = ELEVE = eleve = ElEvE
- l'élément est mémorisé sous l'appellation eleve

Si un élément est défini avec des guillemets, son écriture **DOIT** être conservée.

Exemple d'utilisation des guillemets

```
CREATE TABLE "ElEvE" ...
```

crée une table ElEvE. cette écriture devra être respectée pour toute utilisation de l'élément.

Les écritures suivantes sont incorrectes (table n'existe pas) :

- **SELECT * FROM Eleve**
- **SELECT * FROM ELEVE**
- **SELECT * FROM eleve**
- **SELECT * FROM ElEvE**

La seule écriture correcte est : **SELECT * FROM "ElEvE"**

Notion d'index

Rechercher es informations dans une grande table peut prendre beaucoup de temps.

Solutions possibles :

- Trier les éléments : oui, mais couteux en temps.
- Utiliser une référence indirecte type table de hashage

La 2e solution est privilégiée -> index.

Les index

- Mis en place sur les clés primaires par défaut
- possibilité d'en ajouter en fonction des besoins
 - ORACLE : **CREATE INDEX nom_index ON TABLE une_table (element_indexé)**
 - PostgreSQL : **CREATE INDEX nom_index ON TABLE une_table USING BTREE (element_indexé)**
 - MySQL : **ALTER TABLE une_table ADD INDEX nom_index (element_indexé)**
 - ...

Quelques considérations sur les index

- en parallèle de l'index sont mémorisés des statistiques de fonctionnement
- un index accélère la recherche d'informations
- un index ralentit les opérations d'ajout, modification, suppression
- un index occupe une certaine place sur le support de stockage

Incrémentation automatique

Certaines clés primaires reposent sur un entier avec un mécanisme d'incrément automatique.

Comment le mettre en oeuvre ?

- MySQL, MariaDB, ... : Utiliser la contrainte AUTOINCREMENT lors de la création
- PostgreSQL : Utiliser le type SERIAL
- Utiliser une séquence

Séquence

Une séquence est un objet géré par le Système de Gestion de Base de Données.

- Créer une séquence : `CREATE SEQUENCE nom_sequence`
- Possibilité de modifier certains éléments à la création (incrément, ...)
- Existence d'une fonction d'incrémentation indivisible (2 appels simultanés fourniront 2 résultats différents)
- Syntaxe :
 - ORACLE : `nom-sequence.nextval`
 - PostgreSQL : `nextval(nom-sequence)`
- Fonctions courantes : `nextval`, `curval`, ...

Exemple d'utilisation de séquence

Création d'une séquence :

```
CREATE SEQUENCE seq_personne ;  
CREATE TABLE Personne (  
    Personne_ID INTEGER NOT NULL  
        DEFAULT nextval(seq_personne) PRIMARY KEY,  
    Personne_Nom CHARACTER VARYING(255) NOT NULL  
);
```


Exemple d'utilisation de séquence

Utilisation d'une séquence :

```
INSERT INTO Personne('DUPOND');
```

La table Personne comporte 1 ligne :

- Personne_ID = 1
- Personne_Nom = DUPOND

La séquence a été incrémentée et fournit la valeur 1, qui est insérée par défaut. Un second appel mettrait la valeur 2 dans Personne_ID,

...

Notion de Vue

Comment fournir à un utilisateur des éléments pour faire des requêtes sans qu'il ait à connaître le schéma de la base ou les droits pour accéder aux informations contenues dans certaines tables.

Solution : masquer le schéma en prédéfinissant des requêtes.
=VUE

Notion de Vue

Créer une vue :

```
CREATE VIEW ma_vue AS  
SELECT ...
```

Pour un utilisateur, la vue se comporte comme une **table virtuelle** dont les colonnes sont celles remontées dans le SELECT.

La table virtuelle ne comporte physiquement aucune ligne : la requête est relancée à chaque utilisation.

Exemple de définition d'une Vue

| Eleve | | |
|-------------|-------|----------|
| <u>E_ID</u> | E_Nom | E_Prenom |

| Membre | |
|-------------|-------------|
| <u>E_ID</u> | <u>C_ID</u> |

| Club | |
|-------------|-------|
| <u>C_ID</u> | C_Nom |

```
CREATE VIEW MembreClub AS  
SELECT E_ID, E_Nom, E_Prenom, C_Nom  
FROM Eleve  
INNER JOIN Membre USING (E_ID)  
INNER JOIN Club USING (C_ID)
```

Exemple de d'utilisation de la vue

```
SELECT * FROM MembreClub
```

```
SELECT * FROM MembreClub WHERE E_Nom LIKE 'DUPOND'
```

| | | | |
|------|-------|----------|-------|
| E_ID | E_Nom | E_Prenom | C_Nom |
|------|-------|----------|-------|

Utiliser les vues

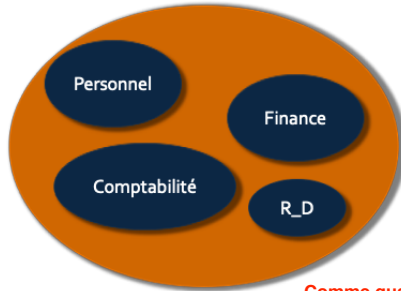
- Lors de l'utilisation d'une vue, un requêteur ne voit que les colonnes indiquées par le SELECT
- Il n'a pas besoin de connaître le schéma de la base pour l'utiliser
- Vous pouvez modifier le schéma de la base et la vue sans que le requêteur ne soit impacté
- Les droits sur la vue sont ceux du requêteur. Vous pouvez protéger les tables comme vous le souhaitez, cela n'impactera pas la vue.

Notion de Schema

Une requête SQL ne fonctionne que sur une seule base de données.

Problème : comment construire des bases de données disjointe, utilisable par des utilisateurs, et pourtant pouvoir faire des requêtes sur l'ensemble ?

Notion de Schema



Comme quatre mini base de données

Comment permettre à chaque service d'utiliser sa base de donnée, et permettre à la DSI de consolider des données dans chacune des bases ?

Notion de Schema

Une solution : schema

- La base de données est commune à tous les utilisateurs
- Dans la base, on crée autant de schema que nécessaire (= sous-base)
- Chaque schema a ses propres tables, ses propres données
- Les noms des tables sont préfixées par le nom du schéma dans lesquelles elles sont.
- Des droits peuvent être définis sur les schemas
- Il reste possible de faire de requêtes sur des tables situées dans des schemas différents.
- Par défaut, il y a toujours un schema : le schema **public**.

Créer un schema

```
CREATE SCHEMA unSchema
```

Utiliser un schema

```
CREATE TABLE unSchema.nom_table (...)
```

```
SELECT * FROM unSchema.nom_table ...
```

```
SELECT *  
FROM Personnel.Personne  
INNER JOIN R_D.Projets USING (Personne_ID)  
WHERE Personnel.Personne.Personne_Nom = 'DUPOND'
```

Importer un fichier CSV

Attention, vous devez avoir des droits administrateurs pour pouvoir faire ce type de manipulation

Objectif : Importer les lignes d'un fichier CSV dans une table

```
COPY table(liste_de_colonne)  
FROM 'chemin_acces_fichier'  
DELIMITER 'délimiteur' CSV ;
```

Exemple d'import de fichier CSV

```
CREATE TABLE public.test
(
  id integer NOT NULL,
  name character varying(256) COLLATE pg_catalog."default",
  value character varying(32) COLLATE pg_catalog."default",
  CONSTRAINT test_pkey PRIMARY KEY (id)
)
```

```
1;abcd;test
2;Ceci est un test;test2
3;"avec un ";test3
```

`COPY test(id, name, value) FROM 'chemin_absolu' DELIMITER ';' CSV ;`

| | id [PK] integer | name character varying (256) | value character varying (32) |
|---|--------------------|---------------------------------|---------------------------------|
| 1 | 1 | abcd | test |
| 2 | 2 | Ceci est un test | test2 |
| 3 | 3 | avec un ; | test3 |

