# Bases du langage SQL
# DML (Data Manipulation Langage)

# Chap 1 : La sélection des données

## Résumé SELECT

L'utilisation la plus courante de SQL consiste à lire des données issues de la base de données. Cela s'effectue grâce à la commande SELECT, qui retourne des enregistrements dans un tableau de résultat. Cette commande peut sélectionner une ou plusieurs colonnes d'une table ou plusieurs tables et sous certaines conditions.

| | |
|---|---|
| SELECT * | # Sélection des colonnes |
| FROM table | # Nom d'une table |
| JOIN table2 on table.id = table2.fkid | # Jointure avec table2 (INNER, LEFT ou RIGHT) |
| WHERE condition | # Obtenir les résultats selon la condition |
| AND condition / OR condition | # Ajouter des conditions |
| GROUP BY expression | # Grouper les tables en groupe |
| HAVING condition | # Condition sur un groupe |
| ORDER BY expression | # Trier les résultats ASC ou DESC |
| LIMIT start, count | # Débuter à partir 'start' enregistrement et Limiter à count enregistrements |

## SQL SELECT

### Commande basique

L'utilisation basique de cette commande s'effectue de la manière suivante :

```
SELECT nom_du_champ
FROM nom_du_tableau
```

Cette requête va sélectionner (SELECT) le champ « nom_du_champ » provenant (FROM) du tableau appelé « nom_du_tableau ».

**Exemple**

Imaginons une base de données appelée « client » qui contient des informations sur les clients d'une entreprise.

**Table « client » :**

| identifiant | prenom | nom | ville |
|---|---|---|---|
| 1 | Pierre | Dupond | Paris |
| 2 | Sabrina | Durand | Nantes |
| 3 | Julien | Martin | Lyon |
| 4 | David | Bernard | Marseille |
| 5 | Marie | Leroy | Grenoble |

Si l'on veut avoir la liste de toutes les villes des clients, il suffit d'effectuer la requête suivante :

```
SELECT ville
FROM client
```

**Résultat :**

| ville |
|---|
| Paris |
| Nantes |
| Lyon |
| Marseille |
| Grenoble |

## Obtenir plusieurs colonnes

Avec la même table client il est possible de lire plusieurs colonnes à la fois. Il suffit tout simplement de séparer les noms des champs souhaités par une virgule. Pour obtenir les prénoms et les noms des clients il faut alors faire la requête suivante :

```
SELECT prenom, nom
FROM client
```

**Résultat :**

| prenom | nom |
|---|---|
| Pierre | Dupond |
| Sabrina | Durand |
| Julien | Martin |
| David | Bernard |
| Marie | Leroy |

## Obtenir toutes les colonnes d'un tableau

Il est possible de retourner automatiquement toutes les colonnes d'un tableau sans avoir à connaître le nom de toutes les colonnes. Au lieu de lister toutes les colonnes, il faut simplement utiliser le caractère « * » (étoile). C'est un joker qui permet de sélectionner toutes les colonnes. Il s'utilise de la manière suivante :

```
SELECT * FROM client
```

Cette requête retourne exactement les mêmes colonnes qu'il y a dans la base de données. Dans notre cas, le résultat sera donc :

| identifiant | prenom | nom | ville |
|---|---|---|---|
| 1 | Pierre | Dupond | Paris |
| 2 | Sabrina | Durand | Nantes |
| 3 | Julien | Martin | Lyon |
| 4 | David | Bernard | Marseille |
| 5 | Marie | Leroy | Grenoble |

# SQL ORDER BY

La commande ORDER BY permet de trier les lignes dans un résultat d'une requête SQL. Il est possible de trier les données sur une ou plusieurs colonnes, par ordre ascendant ou descendant.

### Syntaxe

Une requête où l'on souhaite filtrer l'ordre des résultats utilise la commande ORDER BY de la sorte :

```
SELECT colonne1, colonne2
FROM table
ORDER BY colonne1
```

Par défaut les résultats sont classés par ordre ascendant, toutefois il est possible d'inverser l'ordre en utilisant le suffixe DESC après le nom de la colonne. Par ailleurs, il est possible de trier sur plusieurs colonnes en les séparant par une virgule. Une requête plus élaborée ressemblerait alors cela :

```
SELECT colonne1, colonne2, colonne3
FROM table
ORDER BY colonne1 DESC, colonne2 ASC
```

**A noter** : il n'est pas obligé d'utiliser le suffixe « ASC » sachant que les résultats sont toujours classés par ordre ascendant par défaut. Toutefois, c'est plus pratique pour mieux s'y retrouver, surtout si on a oublié l'ordre par défaut.

### Exemple

Pour l'ensemble de nos exemples, nous allons prendre un base « utilisateur » de test, qui contient les données suivantes :

| id | nom | prenom | date_inscription | tarif_total |
|---|---|---|---|---|
| 1 | Durand | Maurice | 2012-02-05 | 145 |
| 2 | Dupond | Fabrice | 2012-02-07 | 65 |
| 3 | Durand | Fabienne | 2012-02-13 | 90 |
| 4 | Dubois | Chloé | 2012-02-16 | 98 |
| 5 | Dubois | Simon | 2012-02-23 | 27 |

Pour récupérer la liste de ces utilisateurs par ordre alphabétique du nom de famille, il est possible d'utiliser la requête suivante :

```
SELECT *
FROM utilisateur
ORDER BY nom
```

**Résultat :**

| id | nom | prenom | date_inscription | tarif_total |
|---|---|---|---|---|
| 4 | Dubois | Chloé | 2012-02-16 | 98 |
| 5 | Dubois | Simon | 2012-02-23 | 27 |
| 2 | Dupond | Fabrice | 2012-02-07 | 65 |
| 1 | Durand | Maurice | 2012-02-05 | 145 |
| 3 | Durand | Fabienne | 2012-02-13 | 90 |

En utilisant deux méthodes de tri, il est possible de retourner les utilisateurs par ordre alphabétique ET pour ceux qui ont le même nom de famille, les trier par ordre décroissant d'inscription. La requête serait alors la suivante :

```
SELECT *
FROM utilisateur
ORDER BY nom, date_inscription DESC
```

**Résultat :**

| id | nom | prenom | date_inscription | tarif_total |
|---|---|---|---|---|
| 5 | Dubois | Simon | 2012-02-23 | 27 |
| 4 | Dubois | Chloé | 2012-02-16 | 98 |
| 2 | Dupond | Fabrice | 2012-02-07 | 65 |
| 3 | Durand | Fabienne | 2012-02-13 | 90 |
| 1 | Durand | Maurice | 2012-02-05 | 145 |

# SQL LIMIT

La clause LIMIT est à utiliser dans une requête SQL pour spécifier le nombre maximum de résultats que l'on souhaite obtenir. Cette clause est souvent associée à un OFFSET, c'est-à-dire effectuer un décalage sur le jeu de résultat. Ces 2 clauses permettent par exemple d'effectuer des systèmes de pagination (exemple : récupérer les 10 articles de la page 4).

### Syntaxe simple

La syntaxe commune aux principaux systèmes de gestion de bases de données est la suivante :

```
SELECT *
FROM table
LIMIT 10
```

Cette requête permet de récupérer seulement les 10 premiers résultats d'une table. Bien entendu, si la table contient moins de 10 résultats, alors la requête retournera toutes les lignes.

**Bon à savoir :** la bonne pratique lorsque l'on utilise LIMIT consiste à utiliser également la clause ORDER BY pour s'assurer que quoi qu'il en soit-ce sont toujours les bonnes données qui sont présentées. En effet, si le système de tri est non spécifié, alors il est en principe inconnu et les résultats peuvent être imprévisible.

### Limit et Offset

La syntaxe avec MySQL est la suivante :

```
SELECT *
FROM table
LIMIT 5, 10;
```

Cette requête retourne les enregistrements 6 à 15 d'une table. Le premier nombre est l'OFFSET tandis que le suivant est la limite.

## SQL WHERE

La commande WHERE dans une requête SQL permet d'extraire les lignes d'une base de données qui respectent une condition. Cela permet d'obtenir uniquement les informations désirées.

### Syntaxe

La commande WHERE s'utilise en complément à une requête utilisant SELECT. La façon la plus simple de l'utiliser est la suivante :

```
SELECT nom_colonnes
FROM nom_table WHERE condition
```

**Exemple**

Imaginons une base de données appelée « client » qui contient le nom des clients, le nombre de commandes qu'ils ont effectués et leur ville :

| id | nom | nbr_commande | ville |
|---|---|---|---|
| 1 | Paul | 3 | paris |
| 2 | Maurice | 0 | rennes |
| 3 | Joséphine | 1 | toulouse |
| 4 | Gérard | 7 | paris |

Pour obtenir seulement la liste des clients qui habitent à Paris, il faut effectuer la requête suivante :

```
SELECT *
FROM client
WHERE ville = 'paris'
```

**Résultat :**

| id | nom | nbr_commande | nbr_commande |
|---|---|---|---|
| 1 | Paul | 3 | paris |
| 4 | Gérard | 7 | paris |

**Attention :** dans notre cas tout est en minuscule donc il n'y a pas eu de problème. Cependant, si une table est sensible à la casse, il faut faire attention aux majuscules et minuscules.

## Opérateurs de comparaisons

Il existe plusieurs opérateurs de comparaisons. La liste ci-jointe présente quelques-uns des opérateurs les plus couramment utilisés.

| Opérateur | Description |
|---|---|
| = | Égale |
| <> | Pas égale |
| != | Pas égale |
| > | Supérieur à |
| < | Inférieur à |
| >= | Supérieur ou égale à |
| <= | Inférieur ou égale à |
| IN | Liste de plusieurs valeurs possibles |
| BETWEEN | Valeur comprise dans un intervalle donnée (utile pour les nombres ou dates) |
| LIKE | Recherche en spécifiant le début, milieu ou fin d'un mot. |
| IS NULL | Valeur est nulle |
| IS NOT NULL | Valeur n'est pas nulle |

# SQL AND & OR

Une requête SQL peut être restreinte à l'aide de la condition WHERE. Les opérateurs logiques AND et OR peuvent être utilisées au sein de la commande WHERE pour combiner des conditions.

## Syntaxe d'utilisation des opérateurs AND et OR

Les opérateurs sont à ajoutés dans la condition WHERE. Ils peuvent être combinés à l'infini pour filtrer les données comme souhaités.

L'opérateur AND permet de s'assurer que la condition1 ET la condition2 sont vrai :

```
SELECT nom_colonnesFROM nom_table
WHERE condition1 AND condition2
```

L'opérateur OR vérifie quant à lui que la condition1 OU la condition2 est vrai :

```
SELECT nom_colonnes FROM nom_table
WHERE condition1 OR condition2
```

Ces opérateurs peuvent être combinés à l'infini et mélangés. L'exemple ci-dessous filtre les résultats de la table « nom_table » si condition1 ET condition2 OU condition3 est vrai :

```
SELECT nom_colonnes FROM nom_table
WHERE condition1 AND (condition2 OR condition3)
```

**Attention** : il faut penser à utiliser des parenthèses lorsque c'est nécessaire. Cela permet d'éviter les erreurs car et ça améliore la lecture d'une requête par un humain.

## Exemples

Pour illustrer les prochaines commandes, nous allons considérer la table « produit » suivante :

| id | nom | categorie | stock | prix |
|---|---|---|---|---|
| 1 | ordinateur | informatique | 5 | 950 |
| 2 | clavier | informatique | 32 | 35 |
| 3 | souris | informatique | 16 | 30 |
| 4 | crayon | fourniture | 147 | 2 |

## Exemple opérateur AND

L'opérateur AND permet de joindre plusieurs conditions dans une requête. En gardant la même table que précédemment, pour filtrer uniquement les produits informatiques qui sont presque en rupture de stock (moins de 20 produits disponible) il faut exécuter la requête suivante :

```
SELECT * FROM produit
WHERE categorie = 'informatique' AND stock < 20
```

**Résultat :**

| id | nom | categorie | stock | prix |
|---|---|---|---|---|
| 1 | ordinateur | informatique | 5 | 950 |
| 3 | souris | informatique | 16 | 30 |

## Exemple opérateur OR

Pour filtrer les données pour avoir uniquement les données sur les produits « ordinateur » ou « clavier » il faut effectuer la recherche suivante :

```
SELECT * FROM produit
WHERE nom = 'ordinateur' OR nom = 'clavier'
```

**Résultat :**

| id | nom | categorie | stock | prix |
|---|---|---|---|---|
| 1 | ordinateur | informatique | 5 | 950 |
| 2 | clavier | informatique | 32 | 35 |

## Exemple combinaison AND et OR

Il ne faut pas oublier que les opérateurs peuvent être combinés pour effectuer de puissantes recherches. Il est possible de filtrer les produits « informatique » avec un stock inférieur à 20 et les produits « fourniture » avec un stock inférieur à 200 avec la recherche suivante :

```
SELECT * FROM produit
WHERE ( categorie = 'informatique' AND stock < 20 )OR ( categorie =
'fourniture' AND stock < 200 )
```

**Résultat :**

| id | nom | categorie | stock | prix |
|---|---|---|---|---|
| 1 | ordinateur | informatique | 5 | 950 |
| 2 | clavier | informatique | 32 | 35 |
| 4 | crayon | fourniture | 147 | 2 |

# SQL LIKE

L'opérateur LIKE est utilisé dans la clause WHERE des requêtes SQL. Ce mot-clé permet d'effectuer une recherche sur un modèle particulier. Il est par exemple possible de rechercher les enregistrements dont la valeur d'une colonne commence par telle ou telle lettre. Les modèles de recherches sont multiples.

## Syntaxe

La syntaxe à utiliser pour utiliser l'opérateur LIKE est la suivante :

```
SELECT *
FROM table
WHERE colonne LIKE modele
```

Dans cet exemple le « modèle » n'a pas été défini, mais il ressemble très généralement à l'un des exemples suivants :

- **LIKE '%a'** : le caractère « % » est un caractère joker qui remplace tous les autres caractères. Ainsi, ce modèle permet de rechercher toutes les chaines de caractère qui se termine par un « a ».
- **LIKE 'a%'** : ce modèle permet de rechercher toutes les lignes de « colonne » qui commence par un « a ».
- **LIKE '%a%'** : ce modèle est utilisé pour rechercher tous les enregistrements qui utilisent le caractère « a ».
- **LIKE 'pa%on'** : ce modèle permet de rechercher les chaines qui commence par « pa » et qui se terminent par « on », comme « pantalon » ou « pardon ».
- **LIKE 'a_c'** : peu utilisé, le caractère « _ » (underscore) peut être remplacé par n'importe quel caractère, mais un seul caractère uniquement (alors que le symbole pourcentage « % » peut être remplacé par un nombre incalculable de caractères . Ainsi, ce modèle permet de retourner les lignes « aac », « abc » ou même « azc ».

**Exemple**

Imaginons une table « client » qui contient les enregistrements d'utilisateurs :

| id | nom | ville |
|---|---|---|
| 1 | Léon | Lyon |
| 2 | Odette | Nice |
| 3 | Vivien | Nantes |
| 4 | Etienne | Lille |

Obtenir les résultats qui commencent par « N »

Si l'on souhaite obtenir uniquement les clients des villes qui commencent par un « N », il est possible d'utiliser la requête suivante :

```
SELECT *
FROM client
WHERE ville LIKE 'N%'
```

Avec cette requête, seul les enregistrements suivants seront retournés :

| id | nom | ville |
|---|---|---|
| 2 | Odette | Nice |
| 3 | Vivien | Nantes |

Obtenir les résultats se terminant par « e »
**Requête :**

```
SELECT *
FROM client
WHERE ville LIKE '%e'
```

**Résultat :**

| id | nom | ville |
|---|---|---|
| 2 | Odette | Nice |
| 4 | Etienne | Lille |

# SQL IS NULL / IS NOT NULL

Dans le langage SQL, l'opérateur IS permet de filtrer les résultats qui contiennent la valeur NULL. Cet opérateur est indispensable car la valeur NULL est une valeur inconnue et ne peut par conséquent pas être filtrée par les opérateurs de comparaison (cf. égal, inférieur, supérieur ou différent).

## Syntaxe

Pour filtrer les résultats où les champs d'une colonne sont à NULL il convient d'utiliser la syntaxe suivante :

```
SELECT *
FROM `table`
WHERE nom_colonne IS NULL
```

A l'inverse, pour filtrer les résultats et obtenir uniquement les enregistrements qui ne sont pas null, il convient d'utiliser la syntaxe suivante :

```
SELECT *
FROM `table`
WHERE nom_colonne IS NOT NULL
```

**A savoir** : l'opérateur IS retourne en réalité un booléen, c'est à dire une valeur TRUE si la condition est vraie ou FALSE si la condition n'est pas respectée. Cet opérateur est souvent utilisé avec la condition WHERE mais peut aussi trouver son utilité lorsqu'une sous-requête est utilisée.

**Exemple**

Imaginons une application qui possède une table contenant les utilisateurs. Cette table possède 2 colonnes pour associer les adresses de livraison et de facturation à un utilisateur (grâce à une clé étrangère). Si cet utilisateur n'a pas d'adresse de facturation ou de livraison, alors le champ reste à NULL.

**Table « utilisateur » :**

| id | nom | date_inscription | fk_adresse_livraison_id | fk_adresse_facturation_id |
|---|---|---|---|---|
| 23 | Grégoire | 2013-02-12 | 12 | 12 |
| 24 | Sarah | 2013-02-17 | NULL | NULL |
| 25 | Anne | 2013-02-21 | 13 | 14 |
| 26 | Frédérique | 2013-03-02 | NULL | NULL |

**Exemple 1 : utilisateurs sans adresse de livraison**

Il est possible d'obtenir la liste des utilisateurs qui ne possèdent pas d'adresse de livraison en utilisant la requête SQL suivante :

```
SELECT *
FROM `utilisateur`
WHERE `fk_adresse_livraison_id` IS NULL
```

**Résultat :**

| id | nom | date_inscription | fk_adresse_livraison_id | fk_adresse_facturation_id |
|---|---|---|---|---|
| 24 | Sarah | 2013-02-17 | NULL | NULL |
| 26 | Frédérique | 2013-03-02 | NULL | NULL |

Les enregistrements retournés montrent bien que seuls les utilisateurs ayant la valeur NULL pour le champ de l'adresse de livraison.

**Exemple 2 : utilisateurs avec une adresse de livraison**

Pour obtenir uniquement les utilisateurs qui possèdent une adresse de livraison il convient de lancer la requête SQL suivante :

```
SELECT *
FROM `utilisateur`
WHERE `fk_adresse_livraison_id` IS NOT NULL
```

**Résultat :**

| id | nom | date_inscription | fk_adresse_livraison_id | fk_adresse_facturation_id |
|---|---|---|---|---|
| 23 | Grégoire | 2013-02-12 | 12 | 12 |
| 25 | Anne | 2013-02-21 | 13 | 14 |

Les lignes retournés sont exclusivement celles qui n'ont pas une valeur NULL pour le champ de l'adresse de livraison.

# SQL GROUP BY

La commande GROUP BY est utilisée en SQL pour grouper plusieurs résultats et utiliser une fonction sur un groupe de résultat. Sur une table qui contient toutes les ventes d'un magasin, il est par exemple possible de regrouper les ventes par clients identiques et d'obtenir le coût total des achats pour chaque client.

## Syntaxe d'utilisation de GROUP BY

De façon générale, la commande GROUP BY s'utilise de la façon suivante :

```
SELECT colonne1, fonction(colonne2)
FROM table
GROUP BY colonne1
```

**A noter :** cette commande doit toujours s'utiliser après la commande **WHERE** et avant la commande **HAVING**.

### Exemple d'utilisation

Prenons en considération une table « achat » qui résume les ventes d'une boutique :

| id | client | tarif | date |
|---|---|---|---|
| 1 | Pierre | 102 | 2012-10-23 |
| 2 | Simon | 47 | 2012-10-27 |
| 3 | Marie | 18 | 2012-11-05 |
| 4 | Marie | 20 | 2012-11-14 |
| 5 | Pierre | 160 | 2012-12-03 |

Ce tableau contient une colonne qui sert d'identifiant pour chaque ligne, une autre qui contient le nom du client, le coût de la vente et la date d'achat.

Pour obtenir le coût total de chaque client en regroupant les commandes des mêmes clients, il faut utiliser la requête suivante :

```
SELECT client, SUM(tarif)
FROM achat
GROUP BY client
```

La fonction SUM() permet d'additionner la valeur de chaque tarif pour un même client. Le résultat sera donc le suivant :

| client | SUM(tarif) |
|---|---|
| Pierre | 262 |
| Simon | 47 |
| Marie | 38 |

La manière simple de comprendre le GROUP BY c'est tout simplement d'assimiler qu'il va éviter de présenter plusieurs fois les mêmes lignes. C'est une méthode pour éviter les doublons.

Juste à titre informatif, voici ce qu'on obtient de la requête sans utiliser GROUP BY.

**Requête :**

```
SELECT client, SUM(tarif)
FROM achat
```

**Résultat :**

| client | SUM(tarif) |
|---|---|
| Pierre | 262 |
| Simon | 47 |
| Marie | 38 |
| Marie | 38 |
| Pierre | 262 |

## Les fonctions d'agrégations

Il existe plusieurs fonctions qui peuvent être utilisées pour manipuler plusieurs enregistrements, les principales sont les suivantes :

- AVG() pour calculer la moyenne d'un set de valeur. Permet de connaître le prix du paniermoyen pour de chaque client
- COUNT() pour compter le nombre de lignes concernées. Permet de savoir combien d'achats a été effectué par chaque client
- MAX() pour récupérer la plus haute valeur. Pratique pour savoir l'achat le plus cher
- MIN() pour récupérer la plus petite valeur. Utile par exemple pour connaître la date du premierachat d'un client
- SUM() pour calculer la somme de plusieurs lignes. Permet par exemple de connaître le total de tous les achats d'un client

# SQL HAVING

La condition HAVING en SQL est presque similaire à WHERE à la seule différence que HAVING permet de filtrer en utilisant des fonctions telles que SUM(), COUNT(), AVG(), MIN() ou MAX().

## Syntaxe

L'utilisation de HAVING s'utilise de la manière suivante :

```
SELECT colonne1, SUM(colonne2)
FROM nom_table
GROUP BY colonne1
HAVING fonction(colonne2) operateur valeur
```

Cela permet donc de SÉLECTIONNER les colonnes DE la table « nom_table » en GROUPANT les lignes qui ont des valeurs identiques sur la colonne « colonne1" et que la condition de HAVING soit respectée.

**Important** : HAVING est très souvent utilisé en même temps que GROUP BY bien que ce ne soit pas obligatoire.

### Exemple

Pour utiliser un exemple concret, imaginons une table « achat » qui contient les achats de différents clients avec le coût du panier pour chaque achat.

| id | client | tarif | date_achat |
|---|---|---|---|
| 1 | Pierre | 102 | 2012-10-23 |
| 2 | Simon | 47 | 2012-10-27 |
| 3 | Marie | 18 | 2012-11-05 |
| 4 | Marie | 20 | 2012-11-14 |
| 5 | Pierre | 160 | 2012-12-03 |

Si dans cette table on souhaite récupérer la liste des clients qui ont commandé plus de 40€, toute commande confondue alors il est possible d'utiliser la requête suivante :

```
SELECT client, SUM(tarif)FROM achat
GROUP BY client
HAVING SUM(tarif) > 40
```

**Résultat :**

| client | SUM(tarif) |
|---|---|
| Pierre | 162 |
| Simon | 47 |

La cliente « Marie » a cumulé 38€ d'achat (un achat de 18€ et un autre de 20€) ce qui est inférieur à la limite de 40€ imposée par HAVING. En conséquent cette ligne n'est pas affichée dans le résultat.

# Chap 2 : Les jointures SQL

Les jointures en SQL permettent d'associer plusieurs tables dans une même requête. Cela permet d'exploiter la puissance des bases de données relationnelles pour obtenir des résultats qui combinent les données de plusieurs tables de manière efficace.

**Exemple**

En général, les jointures consistent à associer des lignes de 2 tables en associant l'égalité des valeurs d'une colonne d'une première table par rapport à la valeur d'une colonne d'une seconde table. Imaginons qu'une base de 2 données possède une table « utilisateur » et une autre table « adresse » qui contient les adresses de ces utilisateurs. Avec une jointure, il est possible d'obtenir les données de l'utilisateur et de son adresse en une seule requête.

On peut aussi imaginer qu'un site web possède une table pour les articles (titre, contenu, date de publication …) et une autre pour les rédacteurs (nom, date d'inscription, date de naissance …). Avec une jointure il est possible d'effectuer une seule recherche pour afficher un article et le nom du rédacteur. Cela évite d'avoir à afficher le nom du rédacteur dans la table « article ».

Il y a d'autres cas de jointures, incluant des jointures sur la même table ou des jointures d'inégalité. Ces cas étant assez particulier et pas si simple à comprendre, ils ne seront pas élaborés sur cette page.

### Types de jointures

Il y a plusieurs méthodes pour associer 2 tables ensemble. Voici la liste des différentes techniques qui sont les plus utilisées :

- **INNER JOIN :** jointure interne pour retourner les enregistrements quand la condition est vraie dans les 2 tables. C'est l'une des jointures les plus communes.
- **LEFT JOIN (ou LEFT OUTER JOIN) :** jointure externe pour retourner tous les enregistrements de la table de gauche (LEFT = gauche) même si la condition n'est pas vérifiée dans l'autre table.
- **RIGHT JOIN (ou RIGHT OUTER JOIN) :** jointure externe pour retourner tous les enregistrements de la table de droite (RIGHT = droite) même si la condition n'est pas vérifiée dans l'autre table.

## SQL INNER JOIN

Dans le langage SQL la commande INNER JOIN, aussi appelée EQUIJOIN, est un type de jointures très communes pour lier plusieurs tables entre-elles. Cette commande retourne les enregistrements lorsqu'il y a au moins une ligne dans chaque colonne qui correspond à la condition.

### Syntaxe

Pour utiliser ce type de jointure il convient d'utiliser une requête SQL avec cette syntaxe :

```
SELECT *
FROM table1
INNER JOIN table2 ON table1.id = table2.fk_id
```

La syntaxe ci-dessus stipule qu'il faut sélectionner les enregistrements des tables table1 et

table2 lorsque les données de la colonne « id » de table1 est égal aux données de la colonne fk_id de table2.

La jointure SQL peut aussi être écrite de la façon suivante :

```
SELECT *
FROM table1
INNER JOIN table2
WHERE table1.id = table2.fk_id
```

La syntaxe avec la condition WHERE est une manière alternative de faire la jointure mais qui possède l'inconvénient d'être moins facile à lire s'il y a déjà plusieurs conditions dans le WHERE.

**Exemple**

Imaginons une application qui possède une table utilisateur ainsi qu'une table commande qui contient toutes les commandes effectuées par les utilisateurs.

**Table utilisateur :**

| id | prenom | nom | email | ville |
|---|---|---|---|---|
| 1 | Aimée | Marechal | aime.marechal@example.com | Paris |
| 2 | Esmée | Lefort | esmee.lefort@example.com | Lyon |
| 3 | Marine | Prevost | m.prevost@example.com | Lille |
| 4 | Luc | Rolland | lucrolland@example.com | Marseille |

**Table commande :**

| utilisateur_id | date_achat | num_facture | prix_total |
|---|---|---|---|
| 1 | 2013-01-23 | A00103 | 203.14 |
| 1 | 2013-02-14 | A00104 | 124.00 |
| 2 | 2013-02-17 | A00105 | 149.45 |
| 2 | 2013-02-21 | A00106 | 235.35 |
| 5 | 2013-03-02 | A00107 | 47.58 |

Pour afficher toutes les commandes associées aux utilisateurs, il est possible d'utiliser la requête suivante :

```
SELECT id, prenom, nom, date_achat, num_facture, prix_totalFROM utilisateur
INNER JOIN commande ON utilisateur.id = commande.utilisateur_id
```

**Résultats :**

| id | prenom | nom | date_achat | num_facture | prix_total |
|---|---|---|---|---|---|
| 1 | Aimée | Marechal | 2013-01-23 | A00103 | 203.14 |
| 1 | Aimée | Marechal | 2013-02-14 | A00104 | 124.00 |
| 2 | Esmée | Lefort | 2013-02-17 | A00105 | 149.45 |

| | | | | | |
|---|---|---|---|---|---|
| 2 | Esmée | Lefort | 2013-02-21 | A00106 | 235.35 |

Le résultat de la requête montre parfaite la jointure entre les 2 tables. Les utilisateurs 3 et 4 ne sont pas affichés puisqu'il n'y a pas de commandes associées à ces utilisateurs.

**Attention :** il est important de noter que si un utilisateur a été supprimé, alors on ne verra pas ses commandes dans la liste puisque INNER JOIN retourne uniquement les résultats ou la condition est vrai dans les 2 tables.

# SQL LEFT JOIN

Dans le langage SQL, la commande LEFT JOIN (aussi appelée LEFT OUTER JOIN) est un type de jointure entre 2 tables. Cela permet de lister tous les résultats de la table de gauche (left = gauche) même s'il n'y a pas de correspondance dans la deuxième table.

## Syntaxe

Pour lister les enregistrements de table1, même s'il n'y a pas de correspondance avec table2, il convient d'effectuer une requête SQL utilisant la syntaxe suivante.

```
SELECT *
FROM table1
LEFT JOIN table2 ON table1.id = table2.fk_id
```

La requête peut aussi s'écrire de la façon suivante :

```
SELECT *
FROM table1
LEFT OUTER JOIN table2 ON table1.id = table2.fk_id
```

Cette requête est particulièrement intéressante pour récupérer les informations de table1 tout en récupérant les données associées, même s'il n'y a pas de correspondance avec table2. A savoir, s'il n'y a pas de correspondance les colonnes de table2 vaudront toutes NULL.

### Exemple

Imaginons une application contenant des utilisateurs et des commandes pour chacun de ces utilisateurs. La base de données de cette application contient une table pour les utilisateurs et sauvegarde leurs achats dans une seconde table. Les 2 tables sont reliées grâce à la colonne utilisateur_id de la table des commandes. Cela permet d'associer une commande à un utilisateur.

**Table utilisateur :**

| id | prenom | nom | email | ville |
|---|---|---|---|---|
| 1 | Aimée | Marechal | aime.marechal@example.com | Paris |
| 2 | Esmée | Lefort | esmee.lefort@example.com | Lyon |
| 3 | Marine | Prevost | m.prevost@example.com | Lille |
| 4 | Luc | Rolland | lucrolland@example.com | Marseille |

**Table commande :**

| utilisateur_id | date_achat | num_facture | prix_total |
|---|---|---|---|
| 1 | 2013-01-23 | A00103 | 203.14 |
| 1 | 2013-02-14 | A00104 | 124.00 |
| 2 | 2013-02-17 | A00105 | 149.45 |
| 2 | 2013-02-21 | A00106 | 235.35 |
| 5 | 2013-03-02 | A00107 | 47.58 |

Pour lister tous les utilisateurs avec leurs commandes et afficher également les utilisateurs qui n'ont pas effectuées d'achats, il est possible d'utiliser la requête suivante :

```
SELECT *
FROM utilisateur
LEFT JOIN commande ON utilisateur.id = commande.utilisateur_id
```

**Résultats :**

| id | prenom | nom | date_achat | num_facture | prix_total |
|---|---|---|---|---|---|
| 1 | Aimée | Marechal | 2013-01-23 | A00103 | 203.14 |
| 1 | Aimée | Marechal | 2013-02-14 | A00104 | 124.00 |
| 2 | Esmée | Lefort | 2013-02-17 | A00105 | 149.45 |
| 2 | Esmée | Lefort | 2013-02-21 | A00106 | 235.35 |
| 3 | Marine | Prevost | NULL | NULL | NULL |
| 4 | Luc | Rolland | NULL | NULL | NULL |

Les dernières lignes montrent des utilisateurs qui n'ont effectuée aucune commande. La ligne retourne la valeur NULL pour les colonnes concernant les achats qu'ils n'ont pas effectués.

## Filtrer sur la valeur NULL

Attention, la valeur NULL n'est pas une chaîne de caractère. Pour filtrer sur ces caractères il faut utiliser la commande IS NULL. Par exemple, pour lister les utilisateurs qui n'ont pas effectués d'achats il est possible d'utiliser la requête suivante.

```
SELECT id, prenom, nom, utilisateur_id
FROM utilisateur
LEFT JOIN commande ON utilisateur.id = commande.utilisateur_id
WHERE utilisateur_id IS NULL
```

**Résultats :**

| id | prenom | nom | utilisateur_id |
|---|---|---|---|
| 3 | Marine | Prevost | NULL |
| 4 | Luc | Rolland | NULL |

# SQL RIGHT JOIN

En SQL, la commande RIGHT JOIN (ou RIGHT OUTER JOIN) est un type de jointure entre 2 tables qui permet de retourner tous les enregistrements de la table de droite (right = droite) même s'il n'y a pas de correspondance avec la table de gauche. S'il y a un enregistrement de la table de droite qui ne trouve pas de correspondance dans la table de gauche, alors les colonnes de la table de gauche auront NULL pour valeur.

## Syntaxe

L'utilisation de cette commande SQL s'effectue de la façon suivante :

```
SELECT *
FROM table1
RIGHT JOIN table2 ON table1.id = table2.fk_id
```

La syntaxe de cette requête SQL peut aussi s'écrire de la façon suivante :

```
SELECT *
FROM table1
RIGHT OUTER JOIN table2 ON table1.id = table2.fk_id
```

Cette syntaxe stipule qu'il faut lister toutes les lignes du tableau table2 (tableau de droite) et afficher les données associées du tableau table1 s'il y a une correspondance entre ID de table1 et FK_ID de table2. S'il n'y a pas de correspondance, l'enregistrement de table2 sera affiché et les colonnes de table1 vaudront toutes NULL.

### Exemple

Prenons l'exemple d'une base de données qui contient des utilisateurs et un historique d'achat de ces utilisateurs. Cette 2 tables sont reliées entre grâce à la colonne utilisateur_id de la table des commandes. Cela permet de savoir à quel utilisateur est associé un achat.

**Table utilisateur :**

| id | prenom | nom | email | ville | actif |
|---|---|---|---|---|---|
| 1 | Aimée | Marechal | aime.marechal@example.com | Paris | 1 |
| 2 | Esmée | Lefort | esmee.lefort@example.com | Lyon | 0 |
| 3 | Marine | Prevost | m.prevost@example.com | Lille | 1 |
| 4 | Luc | Rolland | lucrolland@example.com | Marseille | 1 |

**Table commande :**

| utilisateur_id | date_achat | num_facture | prix_total |
|---|---|---|---|
| 1 | 2013-01-23 | A00103 | 203.14 |
| 1 | 2013-02-14 | A00104 | 124.00 |
| 2 | 2013-02-17 | A00105 | 149.45 |
| 3 | 2013-02-21 | A00106 | 235.35 |
| 5 | 2013-03-02 | A00107 | 47.58 |

Pour afficher toutes les commandes avec le nom de l'utilisateur correspondant il est normalement d'habitude d'utiliser INNER JOIN en SQL. Malheureusement, si l'utilisateur a été supprimé de la table, alors ça ne retourne pas l'achat. L'utilisation de RIGHT JOIN permet de retourner tous les achats et d'afficher le nom de l'utilisateur s'il existe. Pour cela il convient d'utiliser cette requête :

```
SELECT id, prenom, nom, utilisateur_id, date_achat, num_factureFROM utilisateur
RIGHT JOIN commande ON utilisateur.id = commande.utilisateur_id
```

**Résultats :**

| id | prenom | nom | utilisateur_id | date_achat | num_facture |
|---|---|---|---|---|---|
| 1 | Aimée | Marechal | 1 | 2013-01-23 | A00103 |
| 1 | Aimée | Marechal | 1 | 2013-02-14 | A00104 |
| 2 | Esmée | Lefort | 2 | 2013-02-17 | A00105 |
| 3 | Marine | Prevost | 3 | 2013-02-21 | A00106 |
| NULL | NULL | NULL | 5 | 2013-03-02 | A00107 |

Ce résultat montre que la facture A00107 est liée à l'utilisateur numéro 5. Or, cet utilisateur n'existe pas ou n'existe plus. Grâce à RIGHT JOIN, l'achat est tout de même affiché mais les informations liées à l'utilisateur sont remplacées par NULL.

# SQL Sous-requête

Dans le langage SQL une sous-requête (aussi appelé « requête imbriquée » ou « requête en cascade ») consiste à exécuter une requête à l'intérieur d'une autre requête. Une requête imbriquée est souvent utilisée au sein d'une clause WHERE ou de HAVING pour remplacer une ou plusieurs constantes.

### Syntaxe

Il y a plusieurs façons d'utiliser les sous-requêtes. De cette façon il y a plusieurs syntaxes envisageables pour utiliser des requêtes dans des requêtes.

### Requête imbriquée qui retourne un seul résultat

L'exemple ci-dessous est un exemple typique d'une sous-requête qui retourne un seul résultat à la requête principale.

```
SELECT *
FROM `table`
WHERE
    `nom_colonne` =
    (SELECT `valeur`
    FROM  `table2`
    LIMIT 1
  )
```

Cet exemple montre une requête interne (celle sur « table2") qui renvoi une seule valeur. La requête externe quant à elle, va chercher les résultats de « table » et filtre les résultats à partir de la valeur retournée par la requête interne.

**A noter :** il est possible d'utiliser n'importe quel opérateur d'égalité tel que =, >, <, >=, <= ou <>.

## Requête imbriquée qui retourne une colonne

Une requête imbriquée peut également retournée une colonne entière. Dès lors, la requête externe peut utiliser la commande IN pour filtrer les lignes qui possèdent une des valeurs retournées par la requête interne. L'exemple ci-dessous met en évidence un tel cas de figure :

```
SELECT *
FROM `table`
WHERE
    `nom_colonne` IN
    (SELECT `colonne`
    FROM `table2`
    WHERE `cle_etrangere` = 36
  )
```

# Chap 3 : Ajouter, modifier, supprimer

## SQL INSERT INTO

L'insertion de données dans une table s'effectue à l'aide de la commande INSERT INTO. Cette commande permet au choix d'inclure une seule ligne à la base existante ou plusieurs lignes d'un coup.

**Attention** : lorsque le champ à remplir est de type VARCHAR ou TEXT il faut indiquer le texte entre guillemet simple. En revanche, lorsque la colonne est un numérique, il n'y a pas besoin d'utiliser de guillemet, il suffit juste d'indiquer le nombre.

### Insertion d'une ligne à la fois

Pour insérer des données dans une base, il y a 2 syntaxes principales :

- Insérer une ligne en indiquant les informations pour chaque colonne existante (en respectant l'ordre)
- Insérer une ligne en spécifiant les colonnes que vous souhaitez compléter. Il est possible d'insérer une ligne en ne renseignant seulement qu'une partie des colonnes

#### Insérer une ligne en spécifiant toutes les colonnes

La syntaxe pour remplir une ligne avec cette méthode est la suivante :

```
INSERT INTO nom_table
VALUES ('valeur 1', 'valeur 2', …)
```

Cette syntaxe possède les avantages et inconvénients suivants :

- Obliger de remplir toutes les données, tout en respectant l'ordre des colonnes
- Il n'y a pas le nom de colonne, donc les fautes de frappe sont limitées. Par ailleurs, les colonnes peuvent être renommées sans avoir à changer la requête
- L'ordre des colonnes doit rester identique sinon certaines valeurs prennent le risque d'être complétée dans la mauvaise colonne

#### Insérer une ligne en spécifiant seulement les colonnes souhaitées

Cette deuxième solution est très similaire, excepté qu'il faut indiquer le nom des colonnes avant « VALUES ». La syntaxe est la suivante :

```
INSERT INTO nom_table (nom_colonne1, nom_colonne2, …)
VALUES ('valeur 1', 'valeur 2', …)
```

**A noter** : il est possible de ne pas renseigner toutes les colonnes. De plus, l'ordre des colonnes n'est pas important.

### Insertion de plusieurs lignes à la fois

Il est possible d'ajouter plusieurs lignes à un tableau avec une seule requête. Pour ce faire, il convient d'utiliser la syntaxe suivante :

```
INSERT INTO client (prenom, nom, age)
VALUES
('Marie', 'VIERS', 54),
('Vanessa', 'CALMEL', 32),
('Roger', 'BRUEL', 52)
```

**A noter** : lorsque le champ à remplir est de type VARCHAR ou TEXT il faut indiquer le texte entre guillemet simple. En revanche, lorsque la colonne est un numérique tel que INT ou BIGINT il n'y a pas besoin d'utiliser de guillemet, il suffit juste d'indiquer le nombre.

## SQL UPDATE

La commande UPDATE permet d'effectuer des modifications sur des lignes existantes. Très souvent cette commande est utilisée avec WHERE pour spécifier sur quelles lignes doivent porter la ou les modifications.

### Syntaxe

La syntaxe basique d'une requête utilisant UPDATE est la suivante :

```
UPDATE nom_table
SET colonne1 = 'nouvelle valeur1', colonne2 = 'nouvelle valeur2'
WHERE condition
```

Cette syntaxe permet d'attribuer une nouvelle valeur à la colonne nom_colonne pour les lignes qui respectent la condition stipulée avec WHERE.

## SQL DELETE

La commande DELETE en SQL permet de supprimer des lignes dans une table. En utilisant cette commande associée à WHERE il est possible de sélectionner les lignes concernées qui seront supprimées.

### Syntaxe

La syntaxe pour supprimer des lignes est la suivante :

```
DELETE FOM nom_table
WHERE condition
```

**Attention** : s'il n'y a pas de condition WHERE alors toutes les lignes seront supprimées et la table sera alors vide.