# Le SQL : initiation

Le SQL est un langage informatique destiné à piloter différentes bases de données et les manipuler. Ce langage normé est compris et interprété par l’essentiel des systèmes présents sur le marché. SQL est un langage de type " déclaratif " et ensembliste. Ainsi classiquement, on spécifie ce que l’on veut obtenir ou faire, la machine décide la manière de l’exécuter. La différence fondamentale d’approche en mode développement entre le SQL et le PL/SQL réside dans le fait qu’avec ce dernier, vous indiquez l’ensemble des instructions nécessaires pour traiter un problème mode dit navigationnel).

Puisque le SQL n’inclue pas (sauf exception) la manière dont la requête va être exécutée, les systèmes de base de données incluent le plus souvent des moteurs de résolution afin de trouver la façon optimale de répondre à la requête. Les moteurs se servent de différents outils logiques ou statistiques intégrées au dictionnaire des données.

La syntaxe SQL fait l’objet de la norme ISO 9075. Cette norme permet aux différents systèmes intégrant SQL la possibilité d’enrichir le langage avec des instructions spécifiques et non normalisées.

* SQL est un langage de définition de données : DDL

Les instructions DDL permettent de créer, modifier, supprimer un élément de base de données.

* SQL est un langage de manipulation de données : DML

C'est la partie du SQL qui s'occupe de traiter les données. Elles permettent d'insérer, de modifier de supprimer et d'extraire des données.

* SQL est un langage de protections d'accès : DCL

C'est la partie du SQL qui s'occupe de gérer les droits d'accès aux tables (attribuer ou révoquer les droits).

* SQL est un langage de gestion des transactions : TCL

C'est la partie du SQL chargé de contrôler la bonne exécution des transactions. Cette partie permet notamment de gérer les propriétés **ACID** des transactions (Atomicité des données, Cohérence des données, Isolation des transactions, Durabilité : persistance des données).

Résumé des différents périmètres du SQL

Une image contenant texte

Description générée automatiquement

## Structure générale d’une requête

SELECT attributs recherchés

FROM une ou plusieurs tables

WHERE condition sur les n-uplets des tables

GROUP BY critère de regroupement de la clause SELECT

HAVING condition sur les n-uplets du SELECT

ORDER BY ordre de tri

Exemple 1 : en utilisant Bières, quelles bières sont fabriquées par Anheuser-Busch ?

## Contenu de la clause SELECT

### La structure

Le SELECT (appelé aussi projection) indique la structure des données visualisées. Toutes les informations d’un champ de résultat doivent être de même type (attention pour les UNION). En effet, la structure d’une requête doit obéir aux mêmes règles que les tables (en termes d’entités relationnelles).

Avec une relation dans le FROM clause, \* dans le SELECT signifie que la structure du résultat correspond à la totalité des attributs de la source (la clause FROM)

Exemple 2 : avec Bières

### Les alias

Il est possible de renommer les attributs de la clause SELECT avec des alias à l’aide du mot clé AS.

Exemple 3 avec la table bières ?

Attention, selon la version du moteur SQL, l’alias (AS) peut être remplacé par un <espace>

Comment faire pour nommer une colonne avec un espace à l’intérieur et/ou des caractères accentués ?

### Les expressions

Toute expression calculée à partir des attributs de la clause FROM peut apparaitre ou être utilisée dans un calcul de la clause SELECT.

Exemple 4 : Quels sont les prix en yen (1 euro = 120 yen) des bières vendues dans chaque bar

### Les constantes

Dans l’exemple au-dessus, 120 ne varie pas en fonction de la ligne courante de la relation. Elle est donc une constante. Les constantes peuvent être utilisées au meme titre que les champs composant le FROM.

Exemple 5 : Qui aime la bière MGD ?

### La profondeur

Dans le schéma ensembliste, chaque information peut être un attribut ou un ensemble du moment qu’il satisfait au type de la relation (comparaison d’une valeur à une valeur, ou une valeur comprise dans un ensemble).

Exemple 6 : Quels sont les prix pratiqués pour la bière Coors par rapport à la moyenne des prix ?

Exemple 7 : Comment obtenir le prix de la bière Coors par bar, par rapport à la moyenne des prix de la bière Coors pour tous les bars ?

## Contenu de la clause FROM

La clause FROM recense toutes les sources de la requête. Classiquement il s’agit des tables et des vues de la base de données. Mais il peut s’agit aussi de résultats de requêtes (comme l’exemple au-dessus). On parle ainsi de vues à la volée.

Exemple 8 : pour un résultat identique que ci-dessus, obtenir le prix de la bière Coors par bar, par rapport à la moyenne des prix de la bière Coors pour tous les bars

La résolution de cette requête par la clause FROM fait un produit cartésien, nous verrons plus bas dans ce cours les différentes utilisations de la clause FROM (jointures, etc.)

## Contenu de la clause WHERE

La clause WHERE recense les restrictions appliquées sur la source de données (clause FROM).

Exemple 9 : Quel est le prix de la Bud chez Joe’s ?

* Quelques remarques :

2 apostrophes pour représenter une apostrophe dans une chaîne de caractère

Les opérateurs de comparaison sont utilisables (AND, OR, NOT)

SQL n’est pas case sensitive (attention tout de même aux comparaisons de chaînes de caractères)

Les regex et les expressions régulières sont utilisables ( % avec l’opérateur LIKE)

Exemple 10 : trouver les clients dont le téléphone commence par 831

## Requête à plusieurs relations

Certaines requêtes utilisent plusieurs relations (appelées jointures). Les attributs de toutes ces relations peuvent faire partie du SELECT. Comme il est interdit que 2 attributs d’une même relation aient le même nom (contrainte ensembliste), les attributs auront à être renommer (par alias). A défaut, les principaux systèmes le font afin de ne pas rendre la requête en erreur.

La jointure correspond à la transcription de la relation au travers d’attributs communs. Sans jointure, le système opère un produit cartésien.

Exemple 11 : Quelles sont les bières qu’au moins un des clients du Joe’s apprécie ?

* Sens des requêtes :

Comme dans le cas des requêtes unaires, le sens est :

1 – Faire le produit des relations du FROM

2 – Sélectionner les n-uplets qui satisfont le WHERE

3 – Projeter les attributs sur le SELECT

En sémantique opérationnelle, les requêtes à plusieurs relations forment des boucles imbriquées avec élimination éventuelle des lignes qui ne satisfont pas aux conditions (ou système des sources maitresses). Les clauses de jointures sont donc des restrictions particulières.

Une source de données peut être utilisée plusieurs fois dans la même requête, elle est vue comme plusieurs sessions qui n’intéragissent pas hormis dans les clauses de jointures (une restriction dans une session n’a pas d’influence sur la population d’une autre session).

Afin d’identifier les différentes sessions, on utlise des alias pour chacune de ces sources (donc à introduire dans le FROM).

Les alias sont généralisés et utilisés très souvent même lorsqu’il n’y a pas de besoin technique.

Exemple 12 : Trouver les bières qui ont la même marque (biere\_1, biere\_2).

Exemple 13 : Le même, mais pas de couple avec un nom identique (comme Bud, Bud)

Exemple 14 : Pas 2 fois le même couple dans un ordre différent (comme Miller, Bud et Bud,Miller)

## Sous-requêtes

Le résultat d’une requête a la même structure qu’une table, ainsi il peut être manipulée lui-même au sien d’une requête de plus haut niveau. Cela équivaut au jeu des poupées russes. Une table elle-même n’est jamais manipulée directement en interrogation de données : un FROM table1 équivaut à un FROM (SELECT \* FROM table1).

Ces requêtes (appelées aussi vues instantanées ou à la volée) peuvent être utilisées dans les clauses SELECT, FROM, WHERE et HAVING. Elles sont entourées de parenthèses et nommées dans le FROM afin de pouvoir être utilisées dans les autres clauses.

2 types de sous-requêtes :

* Toujours une et une seule valeur comme résultat
* N résultats (considéré ainsi comme une liste)

La gestion correcte de ces 2 cas de figure dépend du contexte. Par exemple N résultats ne doit pas être utilisé dans un SELECT, et une valeur ne peut pas être comparée en test d’égalité avec une liste de valeurs.

### Sous-requête produisant 1 seul résultat

On peut utiliser ce type de requête comme valeur (dans un SELECT, dans un test d’égalité du WHERE ou du HAVING). L’utilisation d’une clé unique ou d’un calcul permet d’être sûr de respecter cette contrainte. Dans le cas contraire, une erreur système est générée. Les valeurs comparées doivent être de même structure et type.

Exemple 15 : Trouver les Bars qui vendent de la Miller au prix ou Joe’s vend de la Bud.

### Sous-requête produisant un ensemble

Le test d’égalité est un non sens dans ce type de comparaison, puisque les 2 éléments de comparaison ne sont pas par nature de même type (une valeur par rapport à une liste). On peut vouloir savoir si une valeur est incluse dans une liste à l’aide du test IN, ou bien savoir si un ensemble renvoie ou non une liste de résultat à l’aide d’ EXISTS. IN et EXISTS peuvent être combinés à NOT. Ces tests renvoient tous un booléen (TRUE, FALSE).

* Exemple 16 avec IN : trouver toutes les marques des bières que Fred apprécie, sans doublon
* Exemple 17 avec EXISTS : Trouver les bières qui sont les seules produites par leur marque. Une réponse avec EXISTS et une avec COUNT(\*)

## Opérations ensemblistes et multi-ensembles

Une des forces du modèle relationnel repose sur le fait qu’il est fondé sur une base mathématique (théorie des ensembles). Le langage SQL programme les opérations binaires (entre deux tables) suivantes :

* Intersection par l’opérateur **INTERSECT** qui extrait des données présentes simultanément dans les deux tables
* Union par les opérateurs **UNION** et **UNION ALL** qui fusionnent des données des deux tables
* Différence par l’opérateur **MINUS** qui extrait des données présentes dans une table sans être présentes dans la deuxième table
* Produit cartésien par le fait de disposer de deux tables dans la clause FROM, ce qui permet de composer des combinaisons à partir des données des deux tables.

L’opérateur ensembliste se place entre 2 requêtes afin d’en faire la jointure ensembliste.

SELECT….

UNION

SELECT…

Les structures des requêtes doivent toutes être les mêmes. Par convention, les attributs portent le nom de la première requête.

Question : quel opérateur ensembliste n’est pas commutatif ?

### INTERSECT

Sauf que MySQL n’implémente pas intersert.. Comment résoudre tout de même la requête suivante ?

Exemple 18 : trouver les clients qui apprécient des bières et fréquentent les bars qui vendent ces mêmes bières

### UNION

Exemple 19 : faire la liste des clients/bars et clients/bières en ajoutant un champ pour connaitre la nature de la relation (frequenter et aimer)

### MINUS

Sauf que MySQL n’implémente pas intersert.. Comment résoudre tout de même la requête suivante ?

Exemple 20 : trouver les clients qui aiment des bières mais ne fréquentent pas de bars

## Jointures internes

Les jointures permettent d’extraire des données issues de plusieurs tables. Le processus de normalisation du modèle relationnel est basé sur la décomposition et a pour conséquence d’augmenter le nombre de tables d’un schéma. Ainsi, la majorité des requêtes utilisent des jointures nécessaires pour pouvoir extraire des données de tables distinctes.

Une jointure met en relation deux tables sur la base d’une clause de jointure (comparaison de colonnes). Généralement, cette comparaison fait intervenir une clé étrangère d’une table avec une clé primaire d’une autre table (le modèle relationnel est basé sur les valeurs).

Les jointures pouvant être réalisées d’innombrables manières, nous les examinerons dans le chapitre du SQL avancé. Pour l’instant, voyons les jointures dites « relationnelles internes », qui sont aussi les plus utilisées.

SELECT [alias1.]col1, [alias2.]col2...

FROM nomTable1[alias1], nomTable2[alias2]...

WHERE (conditionsDeJointure);

Cette forme est la plus utilisée car elle est la plus simple à écrire. Un autre avantage de ce type de jointure est qu’elle laisse le soin au SGBD d’établir la meilleure stratégie d’accès (choix du premier index à utiliser, puis du deuxième, etc.) pour optimiser les performances. Afin d’éviter les ambiguïtés concernant le nom des colonnes, on utilise en général des alias de tables pour suffixer les tables dans la clause FROM et préfixer les colonnes dans les clauses SELECT et WHERE.

.

Exemple 21 : trouver les buveurs qui aiment des bières et qui fréquentent des bars (le contraire de l’exemple minus plus haut)

Le DISTINCT permet de supprimer les doublons. L’une des différence entre les opérateurs ensemblistes et les jointures est que par nature, les opérateurs ensemblistes suppriment les doublons afin de respecter les notions d’ensembles avec lignes uniques (d’où l’utilisation de ALL si besoin).

## Opérateurs d’agrégation

Les opérateurs d’agrégation (appelés également opérateurs de groupe) permettent d’effectuer un calcul au sein de la projection.

Classiquement pour compter le nombre d’éléments COUNT(\*)

Exemple : connaitre le nombre de bars

SELECT COUNT(\*) FROM bars

Quelques fonctions d’agrégation :

AVG([DISTINCT | ALL] expr) Moyenne de expr(nombre).

COUNT({\* | [DISTINCT | ALL] expr}) Nombre de lignes (\* toutes les lignes, expr pour les colonnes non nulles).

MAX([DISTINCT | ALL] expr) Maximum de expr (nombre, date, chaîne).

MIN([DISTINCT | ALL] expr) Minimum de expr (nombre, date, chaîne).

STDDEV([DISTINCT | ALL] expr) Écart type de expr (nombre).

SUM([DISTINCT | ALL] expr) Somme de expr (nombre)

## GROUP BY

La clause GROUP BY permet de réaliser l’opération par sous-ensemble (analogue à des sous-totaux).

Exemple 22 : trouver le prix moyen de chaque bière

Exemple 23 : trouver pour chaque client, le prix moyen de Bud dans les bars qu’il fréquente

## Insert

INSERT INTO nom\_de\_table VALUES (liste des valeurs)

Exemple : ajouter à aimer que Sally apprécie la Bud :

INSERT INTO aimer VALUES(‘Sally’, ‘Bud’);

On peut aussi préciser les attributs pour insertion :

INSERT INTO nom\_de\_table(liste des attributs) VALUES (liste des valeurs)

Cette précision peut être utile pour :

* Ne pas dépendre de l’ordre de la structure de la table
* Toutes les valeurs n’ont pas forcément de valeur. Dans ce cas le système mettra NULL ou la valeur par défaut définie.

Exemple pour le même insert ci-desssus :

INSERT INTO aimer(biere, client)VALUES(‘Bud’, ‘Sally’);

Il est aussi possible d’insérer plusieurs lignes en une seule commande INSERT, en incluant le résultat d’une requête :

INSERT INTO nom\_de\_table (SELECT …. FROM ….WHERE…)

Exemple : Avec frequenter(client, bar), ajouter à une nouvelle relation amis\_sally(nom)tous les amis potentiels de Sally, ceux qui fréquentent un bar que fréquente aussi Sally :

Sinon, comment créer la table et l’alimenter directement avec cette requête ?

## Delete

DELETE FROM nom\_de\_table [WHERE….]

Une instruction DELETE sans clause WHERE supprime toutes les données de la table (cf TRUNCATE).

Le DELETE peut aussi se baser sur le résultat d’une requête :

Exemple : Suppression dans bieres de toutes les bieres dont le producteur fabrique une autre biere

(on annulera ensuite cette ordre SQL).

Quel pourrait être un algorithme navigationnel ? Est-il le même qu’avec une requête ?

Indice : Si je supprime une bière, au tour d’après la condition ne sera plus vérifiée…

## Update

L’objectif est de modifier la valeur de certains n-uplets.

UPDATE nom\_de\_table

SET <liste des attributs=nouvelle valeur>

[WHERE …]

Exemple : changement du numéro de téléphone de Fred qui devient 555-1212

Exemple d’un UPDATE global : ramener le prix de toutes les bières à 4 euros si le prix est plus élevé

(On annulera ensuite l’ordre SQL)