# Le langage SQL : Le Langage de Manipulation des Données (LMD)

Sources du document :

- Livre bible Oracle 9i, J. Gabillaud, Editions Eni ;
- Support Oracle ;
- ISO Norme 2382:1999 parties 1 à 5, ISO Norme 9075 parties 1 à 14 corrigées en 2005.

Sources du document :

- Site sql.developpez.com ;
- Manuel SQL PostgreSQL (existe en plusieurs versions HTML, PDF…).



Olivier Mondet
http://unidentified-one.net

## A. Introduction

Le langage SQL est composé de différents sous-ensembles :

- **LMD :** Langage de Manipulation des données (DML, Data Manipulation Language)

Permet la manipulation et la mise à jour des tables, composé de quatre ordres fondamentaux : SELECT, UPDATE, INSERT, DELETE.

- **LDD :** Langage de Définition des Données (DDL, Data Definition Language)

Permet la définition et la mise à jour du schéma relationnel de la base de données (mode administration). Composé des ordres suivants : CREATE TABLE, CREATE INDEX, CREATE VIEW, DROP TABLE, DROP INDEX, DROP VIEW, ALTER TABLE.

- **LCD :** Langage de Contrôle des Données (DCL, Data Control Language)

Permet de définir les contraintes d'intégrité, de gérer les accès et les autorisations (administration). Composé des ordres : GRANT, REVOKE, LOCK. Recouvre les déclencheurs (triggers), procédures cataloguées.

Le langage SQL a été normé par l'ANSI (American National Standards Institut) en 1986, repris par l'ISO (Organisation Internationale de Normalisation) en 1987, puis modifié en 1989. On nomme alors les différentes versions : SQL-86, SQL-87 et le SQL-89. En 1992 le SQL prend un virage avec des évolutions majeures : on l'appelle le SQL-92 ou SQL-2. En 1999 est né le SQL-99 ou SQL-3 (qui intégrera désormais les triggers ou déclencheurs). Le SQL a été adossé au XML en 2003 (SQL-2003) L'objet de ce cours n'étant pas de retracer l'historique des ordres nous ne développeront donc pas plus cette partie.

- Extensions de SQL :

SQL+ (ORACLE), langage procédural (PL/SQL), Middleware (ODBC/JDBC), architecture Client/Serveur.

- Utilitaires :

Chargement et déchargement de tables à partir de fichiers.
Déchargement et rechargement de la base.
Générateurs d'états, gestionnaire d'écrans, QBE, interfaces.

# B. L'ordre SELECT

## B.1. Ordre SELECT (version SQL89)

Permet de consulter les tables et les vues d'une BDD (extraire les données). Les clauses SELECT et FROM sont obligatoires.

SELECT [ALL|DISTINCT] <critères de projection>
FROM <tables>
[WHERE <critères de sélection et/ou de jointure>]
[ORDER BY <critère de classement> [ASC|DESC]]
[GROUP BY <critère de regroupement>]
[HAVING <condition de regroupement>]
[FETCH FIRST n ROWS ONLY]

Les clauses sont exécutées dans cet ordre :
FROM → WHERE → GROUP BY / HAVING → SELECT → ORDER BY

Il est possible de donner un autre libellé aux colonnes projetées avec la clause AS : SELECT nomcli AS nom du client…. On peut également écrire : SELECT nomcli "nom du client"… (admis sous Oracle).

Exemple : affichage de toutes les colonnes d'une table.
SELECT *
FROM client;

## B.2. Ordre SELECT (version SQL92)

Dans sa version 92 le SQL impose une séparation distincte entre le critère de sélection et le critère de jointure (placé dans la clause WHERE initialement). Avec cette distinction on obtient une plus grande souplesse dans l'utilisation des critères de jointure. On peut d'ailleurs remplacer l'union (clause UNION), l'intersection (clause INTERSECT) ainsi que toutes les autres opérations sur les ensembles (clause MINUS) par la seule clause JOIN.

Cet ordre est très puissant et permet de nombreuses opérations ensemblistes qui nécessitaient auparavant beaucoup de lignes. Les jointures externes sont détaillées à nouveau au point B.12.4)

SELECT <critères de projection> [AS entete1],
FROM <tables> [LEFT|RIGHT|FULL] [INNER|NATURAL|OUTER|EXCEPTION|UNION] JOIN <tables>
    ON <critères de jointure>
WHERE <critères de sélection>
[ORDER BY <critère de classement> [ASC|DESC]]
[GROUP BY <critère de regroupement>]
[HAVING <condition de regroupement>]
[FETCH FIRST n ROWS ONLY]

La clause **JOIN** peut avoir les fonctions qui suivent :
- JOIN table2, table2 = uniquement les enregistrements en correspondance (comme les jointures classiques),
- table1 LEFT OUTER JOIN table2 = tous les enregistrements de table1,
- table1 EXCEPTION JOIN table2 = uniquement les enregistrements sans correspondance,
- table1 RIGHT OUTER JOIN table2 = tous les enregistrements de table2,
- table1 RIGHT EXCEPTION JOIN table2 = uniquement les enregistrements sans correspondance de table2.

**NB :**

- NATURAL JOIN permet d'éviter de préciser les colonnes concernées par la jointure (que l'on peut restreindre à certaines colonnes avec la clause USING).
- INNER JOIN précise une jointure interne, ce qui est le cas par défaut avec la simple utilisation de JOIN.
- UNION permet de joindre deux tables dont la structure est différente.

Exemple : affichage de tous les clients ayant passé au moins une commande.
```
SELECT numclient, nomclient
FROM client JOIN commande ON client.numclient=commande.numclient;
```

Exemple : affichage de tous les clients n'ayant pas passé de commande.
```
SELECT numclient, nomclient
FROM client EXCEPTION JOIN commande ON client.numclient=commande.numclient;
```

## B.3. Les prédicats (opérateurs) :

### B.3.1. BETWEEN

Permet de rechercher une valeur dans une tranche (AND).

Exemple : affichage de toutes les commandes comprises entre deux dates.
```
SELECT numcde, datecde
FROM commande
WHERE datecde BETWEEN '01/02/01' AND '30/03/01';
```

NB: Ou alors en notation Access:
```
SELECT numcde, datecde
FROM commande
WHERE datecde BETWEEN #01/02/01# AND #30/03/01#;
```

### B.3.2. IN

Permet de chercher une valeur dans une liste (OR).

Exemple : affichage de toutes les commandes de 10 ou 20 articles.
```
SELECT numcde, datecde, qtecde
FROM commande
WHERE qtecde IN (10, 20);
```

### B.3.3. NULL

Permet de tester les valeurs non renseignées dans une colonne.

Exemple : affichage de tous les clients n'ayant pas communiqué leur numéro de téléphone.
```
SELECT numclient, nomclient
FROM client
WHERE telclient IS NULL;
```

### B.3.4. LIKE

Permet de rechercher dans une colonne une chaîne de caractères avec les caractères jokers _ et %.

Exemple : affichage des clients dont le nom commence par 'PSO'.
```
SELECT numclient, nomclient
```

FROM client
WHERE nomclient LIKE ‘PSO%’;

Nota : Sous Access la formulation des jokers est différente (proche plus proche de VB d’ailleurs…) :
**LIKE** permet de chercher une occurrence dans les champs d’une table.

- * = n’importe quelle chaîne de caractères
- ? = n’importe quel caractère # = n’importe quel nombre
- [A-G] = plage de caractères
- [1-8] = plage de nombres
- [!A-G] = hors plage de caractères
- [!1-8] = hors plage de nombres
- En SQL non Access % = * et _ = ?

## B.4. Les prédicats (existentiel) : EXISTS

Renvoie le booléen “vrai” si le sous-select correspondant retourne au moins une ligne (opération de division). Cf. sous requêtes.

## B.5. Les opérateurs booléens et opérateurs de comparaison :

### B.5.1. NOT (ou !)

Permet de sélectionner des lignes qui ne répondent pas à des critères de recherche.

Exemple : affichage des employés qui ne sont pas cadres.
SELECT matricule, nomemployé, categ
FROM employé
WHERE categ NOT = ‘Cadre’;

Ou
SELECT matricule, nomemployé, categ
FROM employé
WHERE categ != ‘Cadre’;
Ou
SELECT matricule, nomemployé, categ
FROM employé
WHERE NOT (categ = ‘Cadre’);

### B.5.2. Autres opérateurs

AND, OR, =, <, >, <=, >=, <>, !=

NB : Sous Access la négation est notée NOT au lieu de !, donc != n’existe pas pour exprimer la différence uniquement <>.

## B.6. Les opérateurs ensemblistes

Nous avons vu lors de l'examen de l'ordre SELECT version SQL-92 que l'adjonction de la clause JOIN pouvait résoudre bon nombre des opérations ensemblistes. Nous présenterons les ordres avec la version SQL-89 car l'objet de cette partie est de comprendre le principe des opérateurs.

### B.6.1. L'union (ou / or)

L'union consiste à combiner deux relations (compatibles) pour créer une troisième relation qui contient toutes les occurrences appartenant à l'une ou à l'autre des relations de départ.

Exemple : on souhaiterait obtenir la liste des candidats passant les épreuves d'anglais ou d'espagnol (l'union des deux tables) (sans doublons).

```
SELECT *
FROM ANGLAIS
UNION
SELECT *
FROM ESPAGN
```

On peut utiliser UNION [ALL] pour avoir toutes les lignes communes aux deux tables (y compris celles en double), sans cela les doublons sont éliminés.

### B.6.2. L'intersection (et / and)

L'intersection consiste à combiner deux relations (compatibles) pour créer une troisième relation qui contient toutes les occurrences appartenant à l'une et à l'autre des relations de départ.

Exemple : on souhaiterait obtenir souhaiterait obtenir la liste des candidats passant les épreuves d'anglais et d'espagnol (l'intersection des deux tables).

```
SELECT *
FROM ANGLAIS
INTERSECT
SELECT *
FROM ESPAGN
```

### B.6.3. La différence (non / not)

La différence consiste à combiner deux relations (compatibles) pour créer une troisième relation qui contient toutes les occurrences appartenant à l'une des relations et non contenues dans l'autre des relations de départ. Deux différences sont possibles.

Exemple : on souhaiterait obtenir souhaiterait obtenir la liste des candidats passant les épreuves d'anglais seulement.

```
SELECT *
FROM ANGLAIS
MINUS
SELECT *
FROM ESPAGN
```

## B.7. Autres prédicats

### B.7.1. ALL

Utilisé avec un opérateur de comparaison sert à tester si une expression est vérifiée dans tous les cas de figure. L'expression est juste si la comparaison est vérifiée pour toutes les valeurs renvoyées par la clause de « sous-sélection ».

### B.7.2. SOME, ANY

Contrairement à ALL, l'expression est juste si la comparaison est vérifiée dans la clause de « sous-sélection » pour au moins une valeur.

## B.8. ORDER

Permet le tri des résultats d'une sélection.

ORDER BY <NomChamp / N° colonne> [ASC|DESC]

Exemple : affichage de la liste des employés triée par catégorie et par niveau de salaire (du plus élevé au plus faible).

```
SELECT matricule, nomemployé, categ, salaire
FROM employé
ORDER BY categ ASC, salaire DESC;
```

## B.9. DISTINCT

Clause qui permet d'éviter d'avoir des lignes redondantes.

Exemple : affichage de la liste des qualifications des employés sans doublons.

```
SELECT DISTINCT qualif
FROM employé;
```

NB : la clause ALL prend toutes les valeurs, mêmes celles en double (ALL est l'option par défaut et est facultatif).

Nota : Sous Access la formulation est différente : on utilise DISTINCTROW.

## B.10. Fonction SQL standards

### B.10.1. SUM

Permet de calculer la somme d'une colonne ou d'une expression.

Exemple : affichage du total des salaires versés.

```
SELECT SUM(salaire) "Total"
FROM employé;
```

NB : Ou en notation pour Access :

```
SELECT SUM(salaire) AS "Total"
FROM employé;
```

### B.10.2. AVG

Permet de calculer ma moyenne d'une colonne ou d'une expression.

Exemple : affichage de la rémunération moyenne des représentant (salaires+commission).
```
SELECT AVG(salaire+comm)
FROM employé;
```

### B.10.3. COUNT

Permet de compter le nombre de lignes.

- COUNT(*) : compte le nombre de lignes ;
- COUNT(colonne) : compte le nombre de valeurs dans une colonne ;
- COUNT(DISTINCT colonne) : compte les valeurs distinctes dans une colonne en éliminant les valeurs nulles.

Exemple : Affichage du nombre total de salariés.
```
SELECT COUNT(*)
FROM employé;
```

Exemple : affichage du nombre de catégories différentes parmi les employés.
```
SELECT COUNT(DISTINCT categ)
FROM employé;
```

### B.10.4. MAX / MIN

Renvoie la valeur maximum ou maximum d'une colonne ou d'une expression.

### B.10.5. VAR, STDDEV

Retourne la variance ou l'écart type.

## B.11. Fonctions de groupage de valeurs

Les fonctions de groupe de valeurs permettent de traiter des informations récapitulatives.

### B.11.1. GROUP BY

Clause qui permet de grouper les lignes par type.

Exemple : affichage du salaire moyen de chaque catégorie d'employés.
```
SELECT categ, AVG(salaire)
FROM employé
GROUP BY categ;
```

Exemple : affichage du nombre de commandes passées par client.
```
SELECT numclient, nomclient COUNT(numcde)
FROM client, commande
WHERE client.numcde = facture.numcde
GROUP BY numclient;
```

### B.11.2. GROUP BY... HAVING

Clause qui permet de spécifier des critères de sélection sur les groupes de valeurs. Cette clause accompagne la clause GROUP BY.
HAVING s'applique à des groupes de lignes alors que WHERE s'applique à des lignes.

Exemple : affichage du nombre de commandes passées par client uniquement dans le cas où il y a plus de 10 commandes.
```
SELECT numclient, nomclient COUNT(numfact)
FROM client, facture
WHERE client.numclient = facture.numclient
GROUP BY numclient
HAVING COUNT(*) > 10;
```

## B.12. Jointures multiples

### B.12.1. Les alias

Il est possible d'utiliser des alias (pour simplifier) pour nommer une table différemment dans la requête.

Par exemple :
```
SELECT F.titre, C.libellé-catégorie
FROM FILM F, CATÉGORIE C
WHERE F.code-catégorie = C.code-catégorie;
```

## B.12.2. SELECT imbriqués (sous-requêtes / sous-sélections)

Il est possible de mettre des SELECT à l'intérieur d'autres SELECT. On peut ainsi faire des requêtes à partir de résultats d'autres requêtes (requêtes imbriquées). Il s'agit là d'une fonctionnalité très puissante du SQL. Les sous-requêtes sont utilisées avec les opérateurs : =, IN, ALL, ANY, EXISTS et SOME.

Un SELECT peut utiliser le résultat d'une autre requête de plusieurs manières.

### *B.12.2.1. Par l'intermédiaire des opérateurs de comparaison*

SELECT...
WHERE expr. op._de_comparaison (sous requete)

Remarque : La sous requête ne devra donner qu'une seule valeur.

Exemple : on veut le nom des élèves de même âge que DUPONT.

```
SELECT nom
FROM élève
WHERE age = (SELECT age
        FROM élève
        WHERE nom = 'DUPONT')
```

Exemple : Liste des articles ayant un prix inférieur à la moyenne de leur famille.

```
SELECT *
FROM tarif t
WHERE prix < (SELECT AVG(prix)
        FROM tariff
        WHERE famille = t.famille)
```

### *B.12.2.2. BETWEEN*

Exemple : Liste des articles ayant un prix variant d'au maximum +/- 10 % par rapport à la moyenne.

```
SELECT *
FROM tariff
WHERE prix BETWEEN (SELECT AVG(prix)*0,9 from tarif) AND (SELECT AVG(prix)*1,1 FROM tarif)
```

### *B.12.2.3. ALL / ANY*

SELECT...
WHERE *expr. op._de_comparaison* ALL | ANY (*sous_requête*)

**ALL** : si la sous-requête retourne plusieurs valeurs, la condition devra être vérifiée par toutes ces valeurs.
**ANY** : il suffira qu'une des valeurs retournées par la sous requête satisfasse la condition.

Exemple : on veut le nom de l'élève plus vieux que tous les élèves habitants Nancy.

```
SELECT nom
FROM élève
WHERE age > ALL (SELECT age
        FROM élève
        WHERE ville = 'NANCY');
```

#### *B.12.2.4. IN / NOT IN*

SELECT...
WHERE *expression* [NOT] IN (*sous_requête*)

**IN** : La valeur de l'expression doit se trouver dans la liste retournée par la sous-requête.
**NOT IN** : La valeur de l'expression ne doit pas se trouver dans la liste retournée par la sous-requête.

Exemple : on veut la liste des élèves qui n'ont jamais été collés (qui ne sont pas dans la table COLLES).

```
SELECT nom
FROM élève
WHERE nom NOT IN (SELECT NOM
        FROM colles);
```

#### *B.12.2.5. EXISTS / NOT EXISTS*

SELECT...
WHERE [NOT] EXISTS (*sous_requête*);

**EXISTS** : La sous-requête devra retourner au moins une valeur pour que la condition soit vraie.
**NOT EXISTS** : La sous-requête ne devra retourner aucune valeur pour que la condition soit vraie.

Exemple : on veut la liste des classes pour lesquelles il n'y a personne de collé.

```
SELECT DISTINCT classe
FROM élève
WHERE NOT EXISTS (SELECT *
        FROM colles
        WHERE élève.classe = colle.classe);
```

### B.12.3. Auto-jointure

Il est possible de définir plusieurs fois la même table dans un même SELECT.

Exemple : nombre de salariés dont le salaire est supérieur au salaire moyen.

```
SELECT COUNT(matricule)
FROM employé
WHERE salaire < (SELECT AVG(salaire)
        FROM employé);
```

Exemple : affichage de la liste des employés gagnant plus que le salarié nommé "GRALOPPY".

```
SELECT a.matricule, a.nomemployé, a.salaire
FROM employé a, employé b
WHERE a.salaire < b.salaire
AND b.nomemployé = "GRALOPPY";
ou
SELECT matricule, nomemployé, salaire
FROM employé
WHERE salaire < (SELECT salaire
        FROM employé
        WHERE nomemployé = "GRALOPPY");
```

### B.12.4. Jointures externes

Lorsque l'on exécute une requête, les lignes des tables qui ne vérifient pas la condition exprimée par le pivot de jointure, ne son pas affichées dans le résultat. Une jointure externe (ou demie jointure) favorise une table, appelée table dominante, par rapport à l'autre, appelée table subordonnée. Les lignes de la table dominante sont affichées même si la condition de jointure n'est pas réalisée.

L'opérateur (+) placé après le nom d'une table indique la table subordonnée (dans laquelle il manque des éléments).

Exemple : Une table contient différents films et une autre les genres. Chaque film possède un seul genre, mais par contre il se trouve que certains genres ne soient pas (encore) affectés à un genre. Une jointure habituelle ne permettrait d'afficher que les films de la base et les genres correspondants, sans afficher les autres genres. Sauf…

```
SELECT film.titrefilm, genre.genrefilm
FROM film, genre
WHERE genre.numgenre = film.numgenre (+);
```

**INNER JOIN** <Table> **ON** <Champ1>=<Champ2> : permet de sélectionner les enregistrements de deux tables jointes en n'affichant pour les deux tables que les enregistrements qui ont une correspondance pour leur champ commun (clé primaire et clé étrangère). Ce qui revient au WHERE table1.champ = table2.champ que nous connaissons.
**LEFT JOIN** <Table> **ON** <Champ1>=<Champ2> : permet de sélectionner les enregistrements de deux tables jointes en affichant pour la table de gauche (clé primaire) tous les enregistrements même s'ils n'ont pas de correspondance dans la table de droite (clé étrangère).
**RIGHT JOIN** <Table> **ON** <Champ1>=<Champ2> : permet de sélectionner les enregistrements de deux tables jointes en affichant pour la table de droite (clé étrangère) tous les enregistrements même s'ils n'ont pas de correspondance dans la table de gauche (clé primaire).

## C. Fonctions essentielles

| **Fonction(x)** | **Retourne ?** | **Exemple** |
|---|---|---|
| MAX(X,Y) | retourne la plus grande valeur de X ou de Y | MAX(prixHA, pritarif) * qte |
| MIN(X,Y) | retourne la plus petite valeur de X ou de Y | MIN(datce, datliv) |
| ABSVAL(x) | la valeur absolue de x | ABSVAL(prix) * qte |
| CEIL(x) | Retourne l'entier immédiatement supérieur à X | CEIL(2,42) = 3<br>CEIL(2,56) = 3 |
| RAND() | Retourne un nombre aléatoire | |
| ROUND(x,y) | Retourne l'arrondi comptable à la précision y | ROUND(2,42 , 1) = 2,40<br>ROUND(2,56 , 1) = 2,60 |
| SIGN(x) | Retourne -1 si x est négatif, 1 s'il est positif, 0 s'il est null | Where SIGN(x) = -1 |
| TRUNCATE(x,y) | Retourne le chiffre immédiatement inférieur à X | TRUNCATE(2,42 , 1) = 2,40<br>TRUNCATE(2,56 , 1) = 2,50 |

| | | |
|---|---|---|
| DEC(x,l,d) | x au format numérique packé avec la lg et la précision demandée. | DEC(zonebinaire)<br>DEC(avg(prix), 9, 2) |
| DIGITS(x) | x en tant que chaîne de caractères | DIGITS(datnum) |
| CHAR(x) | x en tant que chaîne de car. (x étant une date) | CHAR(current date) |
| FLOAT(x) | x au format "virgule flottante" | FLOAT(qte) |
| INT(x) | x au format binaire | INT(codart) |
| STRIP(x) ou TRIM(x)<br>RTRIM(x)<br>LTRIM(x) | supprime les blancs au deux extrémités de x.<br>• les blancs de droite<br>• les blancs de gauche | TRIM(raisoc) |
| LENGTH(x) ou<br>OCTET_LENGTH(x) | la longueur de x | LENGTH(nom)<br>LENGTH(TRIM(nom)) |
| CONCAT(x,y) | concatene X et Y (aussi x CONCAT y ou X !! Y) | CONCAT(nom, prenom) |
| SUBSTR(x,d,l) | extrait une partie de x depuis D sur L octets | SUBSTR(nom, 1, 10)<br>SUBSTR(nom, length(nom), 1) |
| LEFT(x,l) | extrait une partie de x depuis 1 sur L octets | LEFT(nom, 10) |
| RIGHT(x,l) | extrait les L derniers octets de x | RIGHT(nom, 5) |
| SPACE(n) | retourne n blancs | nom concat space(5) concat prenom |
| REPEAT(x,n) | retourne n fois x | repeat('*', 15) |
| MOD(x,y) | le reste de la division de x par y | MOD(annee, 4) |
| RRN(fichier) | N° de rang | RRN(clientp1) |
| TRANSLATE(x)<br>UPPER(x)<br>UCASE(x) | X en majuscule | WHERE UCASE(RAISOC) LIKE 'VO%' |
| LOWER(x)<br>LCASE(x) | x en minuscule | WHERE LCASE(ville) LIKE 'nan%' |
| DECODE(expr,search1,result1,...[,default]) | Effectue une recherche dans une expression et retourne le résultat. | |
| NVL(expr,replace) | Remplace une valeur par une autre | |

# D. Mise à jour des tables

## D.1. Insertion de nouvelles lignes

INSERT INTO <nom de la table>
VALUES (<valeur1> [, <valeur2>, …])

L'instruction doit être suivi de l'instruction COMMIT pour valider.

Exemple : on veut ajouter un nouvel élève dans la table classe.
INSERT INTO classe
VALUES ('Dugenoux', 'Antony', '20/12/85');

COMMIT;

## D.2. Mise-à-jour de valeurs

UPDATE <nom de la table>
SET <colonne1> = nouvelle valeur / NULL / calculs
[<colonne2> = nouvelle valeur / NULL / calculs, …]
[WHERE <condition>]

Exemple : on veut changer le nom de la ville de ceux qui ont le département n° 91000 pour l'ensemble de nos clients.
UPDATE client
SET ville = 'EVRY'
WHERE cp = 91000;

COMMIT;

## D.3. Suppression de lignes

DELETE FROM <nom de la table>
[WHERE <condition>]

L'instruction doit être suivi de l'instruction COMMIT pour valider.

Exemple : On veut supprimer tous les clients qui n'ont pas passé de commande.
DELETE FROM client
WHERE cde<1;

COMMIT;

Exemple : Soit un fichier article ayant une colonne "unité_stockage" et un fichier stock, il s'agit de supprimer les articles dans le fichier stock si la zone "unité_stockage" est à blanc dans le fichier article.

```
DELETE FROM stock S
WHERE exists (SELECT *
        FROM articles
        WHERE codart = S.codart
        AND "unité_stockage" = ' ')
```