**Table des matières**

[***SQL de base :*** 2](#_Toc443549348)

[**Définir comment vos données sont stockées** 2](#_Toc443549349)

[**Manipuler vos données** 2](#_Toc443549350)

[**Syntaxe « SELECT »SQL** 2](#_Toc443549351)

[**MANIPULATION DE TABLE** 5](#_Toc443549352)

[**Create Table** 5](#_Toc443549353)

[**Constraint** 6](#_Toc443549354)

[**Clé primaire** 8](#_Toc443549355)

[**Clé étrangère** 9](#_Toc443549356)

[**Create View** 11](#_Toc443549357)

[**Create Index** 13](#_Toc443549358)

[**Alter Table (pour altérer la structure d’une table)** 14](#_Toc443549359)

[**Drop Table** 17](#_Toc443549360)

[**Truncate Table** 17](#_Toc443549361)

[**Insert Into** 18](#_Toc443549362)

[**Update ﻿** 19](#_Toc443549363)

[**Delete From** 20](#_Toc443549364)

[**COMMANDES SQL :** 22](#_Toc443549365)

[**Jointure :** 22](#_Toc443549366)

[**Jointure Externe :** 23](#_Toc443549367)

[**Fonctions ﻿:** 25](#_Toc443549368)

[**Concaténer Fonction :** 26](#_Toc443549369)

[**SQL AVANCÉ:** 28](#_Toc443549370)

[**Union ﻿** 28](#_Toc443549371)

[**Union All** 29](#_Toc443549372)

[**Intersect** 31](#_Toc443549373)

[**Minus ﻿** 32](#_Toc443549374)

[**RÉFÉRENCE :** 33](#_Toc443549375)

***SQL de base :***

## **Définir comment vos données sont stockées**

* **CREATE DATABASE** est utilisé pour créer une nouvelle base de données vide.
* **DROP DATABASE** est utilisé pour détruire complètement une base de données existante.
* **CREATE TABLE** est utilisé pour créer une nouvelle table dans laquelle vos données seront enregistrées.
* **ALTER TABLE** est utilisé pour modifier la définition d'une table existante.
* **DROP TABLE** est utilisé pour détruire complètement une table existante.

## **Manipuler vos données**

* **SELECT** est utilisé quand vous voulez lire (ou sélectionner) vos données.
* **INSERT** est utilisé quand vous voulez ajouter (ou insérer) de nouvelles données.
* **UPDATE** est utilisé quand vous voulez modifier (ou actualiser) des données existantes.
* **DELETE** est utilisé quand vous voulez enlever (ou supprimer) des données existantes.
* **REPLACE** est utilisé quand vous voulez ajouter ou modifier (ou remplacer) des données existantes ou non.
* **TRUNCATE** est utilisé quand vous voulez vider (ou supprimer) toutes les données d'un modèle.

## **Syntaxe « SELECT »SQL**

**Select**

SELECT "nom de colonne" FROM "nom de table";

**Distinct (permet d’enlever les duplicatas)**

SELECT DISTINCT "nom de colonne"

FROM "nom de table";

**Where**

SELECT "nom de colonne"

FROM "nom de table"

WHERE "condition";

**And/Or**

SELECT "nom de colonne"

FROM "nom de table"

WHERE "condition simples"

{[AND|OR] "condition simples"}+;

**In**

SELECT "nom de colonne"

FROM "nom de table"

WHERE "nom de colonne" IN ('valeur1', 'valeur2', ...);

**Between**

SELECT "nom de colonne"

FROM "nom de table"

WHERE "nom de colonne" BETWEEN 'valeur1' AND 'valeur2';

**Like**

SELECT "nom de colonne"

FROM "nom de table"

WHERE "nom de colonne" LIKE {modèle};

**Order By**

SELECT "nom de colonne"

FROM "nom de table"

[WHERE "condition"]

ORDER BY "nom de colonne" [ASC, DESC];

**Count**

SELECT COUNT("nom de colonne")

FROM "nom de table";

**Group By**

SELECT "nom de colonne 1", SUM("nom de colonne 2")

FROM "nom de table"

GROUP BY "nom de colonne 1";

**Having**

SELECT "nom de colonne 1", SUM("nom de colonne 2")

FROM "nom de table"

GROUP BY "nom de colonne 1"

HAVING (condition fonction);

**Create Table**

CREATE TABLE "nom de table"

("colonne 1" "type de données pour la colonne 1",

"colonne 2" "type de données pour la colonne 2",

... );

**Drop Table**

DROP TABLE "nom de table";

**Truncate Table**

TRUNCATE TABLE "nom de table";

**Insert Into**

INSERT INTO "nom de table" ("colonne 1", "colonne 2", ...)

VALUES ("valeur 1", "valeur 2", ...);

**Update**

UPDATE "nom de table"

SET "colonne 1" = [nouvelle valeur]

WHERE "condition";

**Delete From**

DELETE FROM "nom de table"

WHERE "condition";

## **MANIPULATION DE TABLE**

### **Create Table**

La syntaxe SQL de **CREATE TABLE** est :

**CREATE TABLE "nom de table"  
("colonne 1" "type de données pour la colonne 1",  
"colonne 2" "type de données pour la colonne 2",  
... );**

Pour créer la table de clients spécifiée ci-dessus, il faut saisir :

**CREATE TABLE Customer  
(First\_Name char(50),  
Last\_Name char(50),  
Address char(50),  
City char(50),  
Country char(25),  
Birth\_Date datetime);**

Il est aussi possible d’attribuer une valeur par défaut à chaque colonne. La valeur par défaut est utile lorsqu’aucune valeur de colonne n’a été spécifiée au moment de l’insertion des données dans la table. Pour spécifier une valeur par défaut, ajoutez "Default [value]" après la définition du type de données. Dans l’exemple ci-dessus, pour définir par défaut la colonne "Address" (Adresse) à "Unknown" (Inconnu) et City (Ville) à "Grenoble", il faut saisir :

**CREATE TABLE Customer  
(First\_Name char(50),  
Last\_Name char(50),  
Address char(50) default 'Unknown',  
City char(50) default 'Grenoble',  
Country char(25),  
Birth\_Date datetime);**

Vous pouvez également limiter le type d’informations que doit contenir une table ou une colonne. Pour ce faire, utilisez le mot-clé [**CONSTRAINT**](http://www.1keydata.com/fr/sql/sql-constraint.php), qui sera traité ultérieurement.

### **Constraint**

Les contraintes permettent de limiter le type de données à insérer dans une table. Ces contraintes peuvent être spécifiées lors de la création de la table par l’intermédiaire de l’instruction [**CREATE TABLE**](http://www.1keydata.com/fr/sql/sql-create-table.php), ou après la création de la table au moyen de l’instruction[**ALTER TABLE**](http://www.1keydata.com/fr/sql/sql-alter-table.php).

Les contraintes les plus communes sont :

* [**NOT NULL**](http://www.1keydata.com/fr/sql/sql-constraint.php#not_null)
* [**UNIQUE**](http://www.1keydata.com/fr/sql/sql-constraint.php#unique)
* [**CHECK**](http://www.1keydata.com/fr/sql/sql-constraint.php#check)
* [**Clé primaire**](http://www.1keydata.com/fr/sql/sql-cle-primaire.php)
* [**Clé étrangère**](http://www.1keydata.com/fr/sql/sql-cle-etrangere.php)

Ci-dessous, vous trouverez une description détaillée de chaque contrainte.

**NOT NULL**

Une colonne accepte par défaut une valeur **NULL**. Pour rejeter une valeur **NULL** dans une colonne, vous devez définir une contrainte sur cette colonne en indiquant que celle-ci n’accepte pas de valeur **NULL**.

Par exemple, dans l’instruction suivante

**CREATE TABLE Customer   
(SID integer NOT NULL,   
Last\_Name varchar(30) NOT NULL,   
First\_Name varchar(30));**

Les colonnes "SID" et "Last\_Name" n’accepte pas de valeur **NULL**, alors que "First\_Name" peut en accepter.

**UNIQUE**

La contrainte **UNIQUE** garantit que toutes les valeurs d’une colonne ne peuvent comporter de doublons.

Par exemple, dans l’instruction suivante

**CREATE TABLE Customer   
(SID integer UNIQUE,   
Last\_Name varchar(30),   
First\_Name varchar(30));**

La colonne "SID" n’accepte pas de doublons, alors qu’une telle contrainte ne s’applique pas aux colonnes "Last\_Name" et "First\_Name".

Notez qu’une colonne qui est spécifiée en tant que clé primaire ne peut accepter de doublons. En même temps, une colonne qui n’accepte pas de doublons peut ou ne pas être une clé primaire.

**CHECK**

La contrainte **CHECK** permet que toutes les valeurs d’une colonne satisfassent des conditions données.

Par exemple, dans l’instruction suivante :

**CREATE TABLE Customer   
(SID integer CHECK (SID > 0),   
Last\_Name varchar(30),   
First\_Name varchar(30));**

La colonne "SID" ne peut accepter que des entiers supérieurs à 0.

Notez que la contrainte **CHECK** ne peut pas être exécutée par MySQL cette fois-ci.

[**Clé primaire**](http://www.1keydata.com/fr/sql/sql-cle-primaire.php) et [**clé étrangère**](http://www.1keydata.com/fr/sql/sql-cle-etrangere.php) seront traitées dans les deux prochaines sections.

### **Clé primaire**

La clé primaire concourt à identifier uniquement chaque ligne d’une table. Elle peut représenter une partie d’un enregistrement concret, ou être un champ artificiel (un champ qui n’a rien à voir avec l’enregistrement réel). La clé primaire peut représenter un ou plusieurs champs d’une table. Lorsque la clé primaire représente plusieurs champs, elle est appelée « clé composite ».

Il est possible de spécifier les clés primaires au moment de la création de la table (à l’aide de [**CREATE TABLE**](http://www.1keydata.com/fr/sql/sql-create-table.php)) ou de la modification de la structure de la table existante (par le biais de [**ALTER TABLE**](http://www.1keydata.com/fr/sql/sql-alter-table.php)).

Vous trouverez ci-dessous quelques exemples pour spécifier une clé primaire lors de la création d’une table :

**MySQL**:

**CREATE TABLE Customer   
(SID integer,   
Last\_Name varchar(30),   
First\_Name varchar(30),   
PRIMARY KEY (SID));**

**Oracle**:

**CREATE TABLE Customer   
(SID integer PRIMARY KEY,   
Last\_Name varchar(30),   
First\_Name varchar(30));**

**SQL Server**:

**CREATE TABLE Customer   
(SID integer PRIMARY KEY,   
Last\_Name varchar(30),   
First\_Name varchar(30));**

Vous trouverez ci-dessous quelques exemples pour spécifier une clé primaire lors de la modification d’une table :

**MySQL**:

**ALTER TABLE Customer ADD PRIMARY KEY (SID);**

**Oracle**:

**ALTER TABLE Customer ADD PRIMARY KEY (SID);**

**SQL Server**:

**ALTER TABLE Customer ADD PRIMARY KEY (SID);**

Remarque: avant d’utiliser la commande **ALTER TABLE** pour ajouter une clé primaire, vous devez vous assurer que le champ est défini en tant que '**NOT NULL**' ; en d’autres mots, ce champ n’accepte pas de valeur **NULL**.

### **Clé étrangère**

La clé étrangère représente un champ (ou des champs) qui pointe vers la clé primaire d’une autre table. L’objectif de la clé étrangère est d’assurer l’intégrité référentielle des données. En d’autres mots, seules les valeurs devant apparaître dans la base de données sont permises.

Par exemple, nous avons deux tables, l’une appelée ***CUSTOMER*** qui inclut toutes les données du client, et l’autre ***ORDERS*** qui comprend ses commandes. La contrainte ici est que toutes les commandes doivent être associées à un client qui se trouve déjà référencé dans la table ***CUSTOMER***. Dans ce cas, une clé étrangère devra être placée sur la table ***ORDERS*** et mise en relation avec la clé primaire de la table ***CUSTOMER***. De cette façon, il sera possible d’assurer que toutes les commandes de la table ***ORDERS***sont mises en relation avec un client de la table ***CUSTOMER***. En d’autres mots, la table***ORDERS*** ne peut contenir d’informations sur un client qui ne se trouve pas dans la table ***CUSTOMER***.

La structure de ces deux tables sera comme suit :

Table ***CUSTOMER***

|  |  |
| --- | --- |
| Nom de Colonne | Caractéristique |
| SID | Clé primaire |
| Last\_Name |  |
| First\_Name |  |

Table ***ORDERS***

|  |  |
| --- | --- |
| Nom de Colonne | Caractéristique |
| Order\_ID | Clé primaire |
| Order\_Date |  |
| Customer\_SID | Clé étrangère |
| Amount |  |

Dans l’exemple ci-dessus, la colonne Customer\_SID de la table ***ORDERS*** représente une clé étrangère pointant vers la colonne SID de la table ***CUSTOMER***.

Les exemples ci-dessous illustrent comment spécifier la clé étrangère lors de la création de la table ***ORDERS*** :

**MySQL**:

**CREATE TABLE ORDERS   
(Order\_ID integer,   
Order\_Date datetime,   
Customer\_SID integer,   
Amount double,   
PRIMARY KEY (Order\_ID),   
FOREIGN KEY (Customer\_SID) REFERENCES CUSTOMER (SID));**

**Oracle**:

**CREATE TABLE ORDERS   
(Order\_ID integer PRIMARY KEY,   
Order\_Date date,   
Customer\_SID integer REFERENCES CUSTOMER (SID),   
Amount double);**

**SQL Server**:

**CREATE TABLE ORDERS   
(Order\_ID integer PRIMARY KEY,   
Order\_Date datetime,   
Customer\_SID integer REFERENCES CUSTOMER (SID),   
Amount double);**

Vous trouverez ci-dessous quelques exemples pour spécifier une clé étrangère lors de la modification d’une table. Il est supposé que la table ***ORDERS*** a été créée, et que la clé étrangère n’a pas encore été introduite :

**MySQL**:

**ALTER TABLE ORDERS   
ADD FOREIGN KEY (Customer\_SID) REFERENCES CUSTOMER (SID);**

**Oracle**:

**ALTER TABLE ORDERS   
ADD (CONSTRAINT fk\_orders1) FOREIGN KEY (Customer\_SID) REFERENCES CUSTOMER (SID);**

**SQL Server**:

**ALTER TABLE ORDERS   
ADD FOREIGN KEY (Customer\_SID) REFERENCES CUSTOMER (SID);**

### **Create View**

La syntaxe pour la création d’une vue est comme suit :

**CREATE VIEW "nom de vue" AS "instruction SQL";**

"instruction SQL" peut être n’importe quelle instruction SQL que nous avons vue dans ce didacticiel.

À titre d’illustration, utilisons un exemple simple. Supposons que nous avons la table suivante :

Table ***Customer***

|  |  |
| --- | --- |
| Nom de Colonne | Type de Données |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Address | char(50) |
| City | char(50) |
| Country | char(25) |
| Birth\_Date | datetime |

et pour créer une vue appelée ***V\_Customer*** contenant seulement les colonnes First\_Name, Last\_Name et Country de cette table, il faut saisir :

**CREATE VIEW V\_Customer  
AS SELECT First\_Name, Last\_Name, Country  
FROM Customer;**

Nous avons à présent une vue appelée ***V\_Customer*** avec la structure suivante :

View ***V\_Customer***

|  |  |
| --- | --- |
| Nom de Colonne | Type de Données |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Country | char(25) |

Il est aussi possible d’utiliser une vue pour appliquer des jointures à deux tables. Dans ce cas, les utilisateurs ne verront qu’une vue au lieu de deux tables, disposant ainsi d’instructions SQL beaucoup plus simples. Supposons que nous avons les deux tables suivantes :

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| **Store\_Name** | **Sales** | **Txn\_Date** |
| Los Angeles | 1500 | 05-Jan-1999 |
| San Diego | 250 | 07-Jan-1999 |
| Los Angeles | 300 | 08-Jan-1999 |
| Boston | 700 | 08-Jan-1999 |

Table ***Geography***

|  |  |
| --- | --- |
| **Region\_Name** | **Store\_Name** |
| East | Boston |
| East | New York |
| West | Los Angeles |
| West | San Diego |

et pour créer une vue contenant des ventes par données de région, il faudra définir l’instruction SQL suivante :

**CREATE VIEW V\_REGION\_SALES  
AS SELECT A1.Region\_Name REGION, SUM(A2.Sales) SALES  
FROM Geography A1, Store\_Information A2  
WHERE A1.Store\_Name = A2.Store\_Name  
GROUP BY A1.Region\_Name;**

Nous avons la vue ***V\_REGION\_SALES***, définie pour stocker les ventes par enregistrements de région. Pour connaître le contenu de cette vue, il faut saisir :

**SELECT \* FROM V\_REGION\_SALES;**

Résultat:

|  |  |
| --- | --- |
| **REGION** | **SALES** |
| **East** | **700** |
| **West** | **2050** |

### **Create Index**

Les index nous permettent de retrouver rapidement les données contenues dans une table. Utilisons un exemple pour illustrer l’utilité des index : supposons que nous sommes intéressés à lire des informations sur la culture de piments dans un livre de jardinage. Au lieu de commencer la lecture dès le début jusqu’à parvenir à la section sur les piments, il est beaucoup plus rapide d’aller à l’index en fin de livre, repérer les pages contenant les informations au sujet des piments, puis aller directement à ces pages. Aller à l’index en premier lieu nous permet d’épargner du temps et c’est une méthode largement plus efficace pour repérer les informations dont nous avons besoin.

Le principe est le même quant à retrouver des données d’une table de bases de données. Sans index, le système de base de données balaie la table entière (ce processus est appelé 'table scan' (balayage de table)) pour trouver les informations recherchées. Avec l’index créé, le système de la base de données peut d’abord repérer dans l’index l’emplacement des données, puis aller directement à ces emplacements afin d’obtenir les données requises. C’est beaucoup plus rapide.

Par conséquent, il est plus que souhaitable de créer des index pour les tables. Un index peut couvrir une ou plusieurs colonnes. La syntaxe générale pour la création d’un index est :

**CREATE INDEX "nom d'index" ON "nom de table" (nom de colonne);**

Supposons que nous avons la table suivante :

Table ***Customer***

|  |  |
| --- | --- |
| Nom de Colonne | Type de Données |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Address | char(50) |
| City | char(50) |
| Country | char(25) |
| Birth\_Date | datetime |

Et pour ***créer un index dans la colonne Last\_Name***, il faut saisir

**CREATE INDEX IDX\_CUSTOMER\_LAST\_NAME  
ON Customer (Last\_Name);**

Pour ***créer un index dans City et Country***, il faut saisir

**CREATE INDEX IDX\_CUSTOMER\_LOCATION  
ON Customer (City, Country);**

Il n’y a aucune règle stricte sur les conditions de nommer un index. La méthode généralement acceptée consiste à placer un préfixe, tel que "IDX\_", devant un nom d’index pour éviter toute confusion avec d’autres objets de la base de données. Une bonne idée aussi est de fournir des informations sur les tables et colonnes indexées.

Notez que la syntaxe exacte de **CREATE INDEX** peut différer en fonction de la base de données utilisée. Consultez votre manuel de référence de la base de données pour connaître la syntaxe précise.

### **Alter Table (pour altérer la structure d’une table)**

Vous pouvez être amené à modifier à n’importe quel moment la structure d’une table créée dans la base de données. Les raisons classiques de modification d’une table sont les suivantes :

* Ajouter une colonne
* Supprimer une colonne
* Changer un nom de colonne
* Changer les types de données d’une colonne

Noter que la liste ci-dessus n’est pas exhaustive. La commande **ALTER TABLE** permet aussi de modifier la structure d’une table (changer les spécifications de la clé primaire ou ajouter une contrainte unique à une colonne).

La syntaxe SQL de **ALTER TABLE** est :

**ALTER TABLE "nom de table"  
[alter spécifications];**

[alter spécifications] dépend du type de modification à effectuer. Pour les utilisations mentionnées ci-dessus, les instructions [alter specification] sont :

* Ajouter une colonne : ADD "colonne 1" "type de données pour la colonne 1"
* Supprimer une colonne : DROP "colonne 1"
* Changer un nom de colonne : CHANGE "vieux nom de colonne" "nouvelle nom de colonne" "type de données pour le nouveau nom de colonne"
* Changer le type de données d’une colonne : MODIFY "colonne 1" "nouvelle type de données"

En utilisant la table "customer" créée dans la section [**CREATE TABLE**](http://www.1keydata.com/fr/sql/sql-create-table.php), il convient de se reporter aux exemples mentionnés ci-dessus.

Table ***Customer***

|  |  |
| --- | --- |
| Nom de Colonne | Type de Données |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Address | char(50) |
| City | char(50) |
| Country | char(25) |
| Birth\_Date | datetime |

Pour ajouter une colonne appelée "Gender" (Sexe) à cette table, il faut saisir :

**ALTER TABLE Customer ADD Gender char(1);**

Structure de table résultante:

Table ***Customer***

|  |  |
| --- | --- |
| Nom de Colonne | Type de Données |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Address | char(50) |
| City | char(50) |
| Country | char(25) |
| Birth\_Date | datetime |
| Gender | char(1) |

Ensuite, pour renommer "Address" comme "Addr", il suffit de saisir :

**ALTER TABLE Customer CHANGE Address Addr char(50);**

Structure de table résultante:

Table ***Customer***

|  |  |
| --- | --- |
| Nom de Colonne | Type de Données |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Addr | char(50) |
| City | char(50) |
| Country | char(25) |
| Birth\_Date | datetime |
| Gender | char(1) |

Puis, pour changer le type de données pour "Addr" à char(30), il suffit de saisir :

**ALTER TABLE Customer MODIFY Addr char(30);**

Structure de table résultante:

Table ***Customer***

|  |  |
| --- | --- |
| Nom de Colonne | Type de Données |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Addr | char(30) |
| City | char(50) |
| Country | char(25) |
| Birth\_Date | datetime |
| Gender | char(1) |

Finalement, pour supprimer la colonne "Gender", il suffit de saisir :

**ALTER TABLE Customer DROP Gender;**

Structure de table résultante :

Table ***Customer***

|  |  |
| --- | --- |
| Nom de Colonne | Type de Données |
| First\_Name | char(50) |
| Last\_Name | char(50) |
| Addr | char(30) |
| City | char(50) |
| Country | char(25) |
| Birth\_Date | datetime |

### **Drop Table**

Nous sommes, parfois, amenés à supprimer une table d’une base de données pour une raison quelconque. La commande **DROP TABLE** de SQL est donc idéale pour la suppression de tables d’une base de données. La syntaxe pour **DROP TABLE** est :

**DROP TABLE "nom de table";**

Ainsi, pour supprimer la table « Customer » créée dans la section [**CREATE TABLE**](http://www.1keydata.com/fr/sql/sql-create-table.php), il suffit de saisir :

**DROP TABLE Customer;**

### **Truncate Table**

Cette commande est très utile pour supprimer toutes les données d’une table. La commande **DROP TABLE**, que nous avons vue dans la [**last section**](http://www.1keydata.com/fr/sql/sql-drop-table.php) permet aussi d’exécuter cette opération. Mais comment devons-nous procéder pour supprimer les données tout en conservant la table ? Pour ce faire, il convient d’utiliser la commande**TRUNCATE TABLE**. La syntaxe pour **TRUNCATE TABLE** est :

**TRUNCATE TABLE "nom de table";**

Pour supprimer la table « Customer » créée dans [**SQL CREATE TABLE**](http://www.1keydata.com/fr/sql/sql-create-table.php), il suffit de saisir :

**TRUNCATE TABLE Customer;**

### **Insert Into**

Dans les sections précédentes, nous avons appris comment retrouver des informations dans les tables. Mais comment pouvons-nous ajouter ces lignes de données dans ces tables en première position ? Ce thème et l’instruction **INSERT** seront décrits dans cette section, tandis que l’instruction [**UPDATE**](http://www.1keydata.com/fr/sql/sql-update.php) sera décrite dans la section suivante.

Sous SQL, il existe deux manières de base pour **INSÉRER** des données dans une table : l’une consiste à insérer des données une ligne à la fois, et l’autre plusieurs à la fois. Pour **INSÉRER** des données une ligne à la fois :

La syntaxe pour l’insertion de données dans une table une ligne à la fois est comme suit :

**INSERT INTO "nom de table" ("colonne 1", "colonne 2", ...)  
VALUES ("valeur 1", "valeur 2", ...);**

Supposons que nous avons une table qui a la structure suivante,

Table ***Store\_Information***

|  |  |
| --- | --- |
| Nom de Colonne | Type de Données |
| Store\_Name | char(50) |
| Sales | float |
| Txn\_Date | datetime |

et pour insérer une ligne supplémentaire dans la table représentant les données de ventes pour Los Angeles le 10 janvier 1999, dont les ventes de ce magasin, à ce jour, s’élevaient à 900 €, il faudra utiliser le script SQL suivant :

**INSERT INTO Store\_Information (Store\_Name, Sales, Txn\_Date)  
VALUES ('Los Angeles', 900, '10-Jan-1999');**

Le deuxième type de la commande **INSERT INTO** permet d’ajouter plusieurs lignes à une table. Contrairement à l’exemple précédent, où nous avons inséré une seule ligne en spécifiant ses valeurs pour toutes les colonnes, nous devons maintenant utiliser une instruction [**SELECT**](http://www.1keydata.com/fr/sql/sql-select.php) pour spécifier les données à ajouter à la table. Nous sommes donc en train d’insérer des données provenant d’une autre table. La syntaxe est comme suit :

**INSERT INTO "table1" ("column1", "column2", ...)  
SELECT "column3", "column4", ...  
FROM "table2";**

Notez que c’est la méthode la plus simple. L’instruction entière peut facilement contenir les clauses [**WHERE**](http://www.1keydata.com/fr/sql/sql-where.php), [**GROUP BY**](http://www.1keydata.com/fr/sql/sql-group-by.php), et [**HAVING**](http://www.1keydata.com/fr/sql/sql-having.php), ainsi que les jointures et alias de table.

Par exemple, pour avoir une table ***Store\_Information***, regroupant les données de ventes de l’année 1998, et que vous savez d’avance que les données d’origine sont contenues dans la table ***Sales\_Information***, il faut saisir :

**INSERT INTO Store\_Information (Store\_Name, Sales, Txn\_Date)  
SELECT Store\_Name, Sales, Txn\_Date  
FROM Sales\_Information  
WHERE Year (Txn\_Date) = 1998;**

C’est la syntaxe SQL Server qui a été utilisée ici pour extraire les informations annuelles périmées. Chaque base de données relationnelle a sa propre syntaxe. Par exemple, sous Oracle, vous utiliserez **TO\_CHAR (Txn\_Date, 'yyyy') = 1998**.

### **Update ﻿**

Nous pouvons, parfois, être amenés à modifier les données contenues dans une table. Pour ce faire, il convient d’utiliser la commande **UPDATE**. La syntaxe de cette commande est :

**UPDATE "nom de table"  
SET "colonne 1" = [nouvelle valeur]  
WHERE "condition";**

Dans le cas d’une table comme suit :

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| **Store\_Name** | **Sales** | **Txn\_Date** |
| Los Angeles | 1500 | 05-Jan-1999 |
| San Diego | 250 | 07-Jan-1999 |
| Los Angeles | 300 | 08-Jan-1999 |
| Boston | 700 | 08-Jan-1999 |

Et nous nous rendons compte que les ventes pour Los Angeles du 08-Jan-1999 sont en réalité de 500 € au lieu de 300 €, et que cette entrée particulière doit être corrigée. Pour ce faire, nous utiliserons la requête SQL suivante :

**UPDATE Store\_Information  
SET Sales = 500  
WHERE Store\_Name = 'Los Angeles'  
AND Txn\_Date = '08-Jan-1999';**

La table résultante ressemblerait à

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| **Store\_Name** | **Sales** | **Txn\_Date** |
| Los Angeles | 1500 | 05-Jan-1999 |
| San Diego | 250 | 07-Jan-1999 |
| Los Angeles | 500 | 08-Jan-1999 |
| Boston | 700 | 08-Jan-1999 |

Dans ce cas, il n’y a qu’une ligne qui satisfait la condition de la clause [**WHERE**](http://www.1keydata.com/fr/sql/sql-where.php). Toutes les lignes qui satisfont la condition seront modifiées.

Il est également possible de **UPDATE** plusieurs colonnes à la fois. La syntaxe dans ce cas-ci ressemblerait à ce qui suit :

**UPDATE "nom de table"  
SET colonne 1 = [valeur 1], colonne 2 = [valeur 2]  
WHERE "condition";**

### **Delete From**

Nous pouvons, parfois, être amenés à supprimer des enregistrements d’une table. Pour ce faire, il est possible d’utiliser la commande **DELETE FROM**. La syntaxe de cette commande est :

**DELETE FROM "nom de table"  
WHERE "condition";**

Il est plus facile de comprendre en utilisant un exemple. Supposons que nous avons actuellement une table comme suit :

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| **Store\_Name** | **Sales** | **Txn\_Date** |
| Los Angeles | 1500 | 05-Jan-1999 |
| San Diego | 250 | 07-Jan-1999 |
| Los Angeles | 300 | 08-Jan-1999 |
| Boston | 700 | 08-Jan-1999 |

Et nous décidons de ne conserver aucune information de Los Angeles dans cette table. Pour ce faire, nous saisissons la requête SQL suivante :

**DELETE FROM Store\_Information  
WHERE Store\_Name = 'Los Angeles';**

Le contenu de la table devrait paraître à

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| **Store\_Name** | **Sales** | **Txn\_Date** |
| San Diego | 250 | 07-Jan-1999 |
| Boston | 700 | 08-Jan-1999 |

## **COMMANDES SQL :**

### **Jointure :**

Nous souhaitons maintenant analyser les jointures. Pour définir correctement des jointures sous SQL, il faudra se reporter à la plupart des éléments que nous avons exposés jusqu’à présent. Supposons que nous avons les deux tables suivantes :

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| **Store\_Name** | **Sales** | **Txn\_Date** |
| Los Angeles | 1500 | 05-Jan-1999 |
| San Diego | 250 | 07-Jan-1999 |
| Los Angeles | 300 | 08-Jan-1999 |
| Boston | 700 | 08-Jan-1999 |

Table ***Geography***

|  |  |
| --- | --- |
| **Region\_Name** | **Store\_Name** |
| East | Boston |
| East | New York |
| West | Los Angeles |
| West | San Diego |

et nous souhaitons connaître les ventes par région. La table ***Geography*** comprend des informations sur des régions et magasins, et la table ***Store\_Information*** contient des données relatives aux ventes de chaque magasin. Afin d’obtenir les informations sur les ventes par région, il faut combiner les informations des deux tables. En examinant les deux tables, nous constatons qu’elles sont mises en relation par le champ commun « Store\_Name ». Nous allons d’abord décrire l’instruction SQL, puis expliquer l’utilisation de chaque segment :

**SELECT A1.Region\_Name REGION, SUM(A2.Sales) SALES  
FROM Geography A1, Store\_Information A2  
WHERE A1.Store\_Name = A2.Store\_Name  
GROUP BY A1.Region\_Name;**

Résultat:

|  |  |
| --- | --- |
| **REGION** | **SALES** |
| **East** | **700** |
| **West** | **2050** |

Les deux premières lignes permettent à SQL de sélectionner deux champs : le premier est le champ "Region\_Name" de la table ***Geography*** (dont l’alias est REGION), et le second est la somme du champ "Sales" de la table ***Store\_Information*** (dont l’alias est SALES). Notez comment les alias de tables sont utilisés ici : A1 est l’alias de Geography et A2 de Store\_Information. Sans l’alias, la première ligne deviendrait :

**SELECT Geography.Region\_Name REGION, SUM(Store\_Information.Sales) SALES**

Ce qui n’est pas très concis. Par essence, les alias de table sont destinés à faciliter la compréhension des instructions SQL, surtout en présence de plusieurs tables.

Nous passons maintenant à la ligne 3 (à l’instruction [**WHERE**](http://www.1keydata.com/fr/sql/sql-where.php)). C’est là où se spécifie la condition de la jointure. Dans ce cas, pour nous assurer que le contenu dans « store\_name » de la table Geography corresponde à celui de la table ***Store\_Information***, il faut les définir comme valeur égale. L’instruction **WHERE** est essentielle pour obtenir une requête correcte. Sans l’instruction **WHERE** appropriée, nous obtiendrons une jointure cartésienne. Les jointures cartésiennes retournent toute combinaison possible des deux tables (ou tout nombre de tables indiqué dans l’instruction **FROM**). Une jointure cartésienne retournerait donc un résultat d’un total de 4 x 4 = 16 lignes.

### **Jointure Externe :**

Nous avons précédemment analysé la jointure gauche ou jointure intérieure, où nous avons sélectionné des lignes communes aux tables désignées pour une jointure. Comment devons-nous procéder dans les cas où nous serions intéressés à sélectionner des éléments d’une table indépendamment de s’ils sont contenus dans la deuxième table ? Il est nécessaire d’utiliser maintenant la commande **OUTER JOIN** de SQL.

La syntaxe pour exécuter une jointure externe sous SQL dépend de la base de données utilisée. Par exemple, dans Oracle, il faut placer un "(+)" dans la clause [**WHERE**](http://www.1keydata.com/fr/sql/sql-where.php) de l’autre côté de la table dans laquelle nous souhaitons inclure toutes les lignes.

Supposons que nous avons les deux tables suivantes :

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| **Store\_Name** | **Sales** | **Txn\_Date** |
| Los Angeles | 1500 | 05-Jan-1999 |
| San Diego | 250 | 07-Jan-1999 |
| Los Angeles | 300 | 08-Jan-1999 |
| Boston | 700 | 08-Jan-1999 |

Table ***Geography***

|  |  |
| --- | --- |
| **Region\_Name** | **Store\_Name** |
| East | Boston |
| East | New York |
| West | Los Angeles |
| West | San Diego |

Et nous souhaitons trouver le montant des ventes de tous les magasins. Dans le cas d’une jointure régulière, il n’est pas possible d’obtenir le critère de recherche souhaité étant donné que "New York" n’apparaîtra pas dans la table ***Store\_Information***. Par conséquent, il est nécessaire d’exécuter une jointure externe sur les deux tables ci-dessus :

**SELECT A1.Store\_Name, SUM(A2.Sales) SALES  
FROM Geography A1, Store\_Information A2  
WHERE A1.Store\_Name = A2.Store\_Name (+)  
GROUP BY A1.Store\_Name;**

Notez que dans ce cas de figure, c’est la syntaxe Oracle qui est utilisée pour la jointure externe.

Résultat:

|  |  |
| --- | --- |
| **Store\_Name** | **SALES** |
| **Boston** | **700** |
| **New York** |  |
| **Los Angeles** | **1800** |
| **San Diego** | **250** |

Remarque : à défaut d’une correspondance dans la deuxième table, la valeur NULL sera retournée. Dans ce cas, "New York" n’apparaîtra pas dans la table ***Store\_Information***, sa colonne "SALES" correspondante contient une valeur NULL.

### **Fonctions ﻿:**

Puisque nous avons commencé à utiliser des chiffres, la question qui nous vient à l’esprit naturellement est de nous demander s’il est possible d’effectuer des calculs arithmétiques avec ces chiffres, tels que l’addition ou le calcul de la moyenne. La réponse est oui ! SQL dispose des fonctions arithmétiques suivantes :

* **AVG**
* **COUNT**
* **MAX**
* **MIN**
* **SUM**

La syntaxe pour utiliser ces fonctions est :

**SELECT "nom de fonction"("nom de colonne")  
FROM "nom de table";**

Par exemple, pour obtenir la somme de toutes les ventes à partir de la table suivante :

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| **Store\_Name** | **Sales** | **Txn\_Date** |
| Los Angeles | 1500 | 05-Jan-1999 |
| San Diego | 250 | 07-Jan-1999 |
| Los Angeles | 300 | 08-Jan-1999 |
| Boston | 700 | 08-Jan-1999 |

il faut saisir :

**SELECT SUM(Sales) FROM Store\_Information;**

Résultat :

|  |
| --- |
| **SUM(Sales)** |
| **2750** |

***2.750*** représente la somme de toutes les entrées Sales : 1.500 + 250 + 300 + 700.

En plus d’utiliser des fonctions, avec SQL vous pouvez aussi effectuer de simples tâches telles que l’addition (+) et la soustraction (-). Pour les données à type de caractères, plusieurs fonctions de chaînes sont disponibles, telles que les fonctions [**de concaténation**](http://www.1keydata.com/fr/sql/sql-concatener.php), [**de Trim**](http://www.1keydata.com/fr/sql/sql-trim.php) et [**de sous-chaîne**](http://www.1keydata.com/fr/sql/sql-substring.php). Chaque fournisseur SGBDR dispose de sa propre implémentation de fonctions de chaîne, et nous vous recommandons de consulter les références de votre SGBDR pour en savoir plus sur l’utilisation de ces fonctions.

### **Concaténer Fonction :**

Il est parfois nécessaire de combiner ensemble (concaténer) les résultats de plusieurs champs différents. Chaque base de données a sa propre méthode de concaténation:

* MySQL: CONCAT( )
* Oracle: CONCAT( ), ||
* SQL Server: +

La syntaxe pour **CONCAT( )** est comme suit :

**CONCAT (str1, str2, str3, ...)**

Concaténez ensemble str1, str2, str3, et toute autre chaîne. Notez que la fonction**CONCAT( )** d’Oracle ne permet que deux arguments : cette fonction ne permet de concaténer ensemble que deux chaînes à la fois. En utilisant '||', il est toutefois possible de concaténer plus de deux chaînes à la fois dans Oracle.

Analysons quelques exemples. Supposons que nous avons la table suivante :

Table ***Geography***

|  |  |
| --- | --- |
| **Region\_Name** | **Store\_Name** |
| East | Boston |
| East | New York |
| West | Los Angeles |
| West | San Diego |

Exemple 1

**MySQL/Oracle**:

**SELECT CONCAT (Region\_Name, Store\_Name) FROM Geography   
WHERE Store\_Name = 'Boston';**

Résultat:

**'EastBoston'**

Exemple 2

**Oracle**:

**SELECT Region\_Name || ' ' || Store\_Name FROM Geography   
WHERE Store\_Name = 'Boston';**

Résultat:

**'East Boston'**

Exemple 3

**SQL Server**:

**SELECT Region\_Name + ' ' + Store\_Name FROM Geography   
WHERE Store\_Name = 'Boston';**

Résultat:

**'East Boston'**

## **SQL AVANCÉ:**

### **Union ﻿**

L’objectif de la commande **UNION** de SQL est de combiner ensemble les résultats de deux requêtes. La commande **UNION**  est un peu similaire à [**Jointure**](http://www.1keydata.com/fr/sql/sql-jointure.php) du fait qu’elles sont utilisées pour mettre en liaison des informations provenant de plusieurs tables. Une restriction de **UNION** est que toutes les colonnes correspondantes doivent inclure le même type de données. Aussi, lors de l’utilisation de **UNION**, seules les valeurs distinctes sont sélectionnées (similaire à **SELECT**[**DISTINCT**](http://www.1keydata.com/fr/sql/sql-distinct.php)).

La syntaxe est comme suit :

**[instructions SQL 1]  
UNION  
[instructions SQL 2];**

Supposons que nous avons les deux tables suivantes :

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| **Store\_Name** | **Sales** | **Txn\_Date** |
| Los Angeles | 1500 | 05-Jan-1999 |
| San Diego | 250 | 07-Jan-1999 |
| Los Angeles | 300 | 08-Jan-1999 |
| Boston | 700 | 08-Jan-1999 |

Table ***Internet\_Sales***

|  |  |
| --- | --- |
| **Txn\_Date** | **Sales** |
| 07-Jan-1999 | 250 |
| 10-Jan-1999 | 535 |
| 11-Jan-1999 | 320 |
| 12-Jan-1999 | 750 |

et pour retrouver toutes les dates où il y a des ventes, il faudra utiliser l’instruction SQL suivante :

**SELECT Txn\_Date FROM Store\_Information  
UNION  
SELECT Txn\_Date FROM Internet\_Sales;**

Résultat:

|  |
| --- |
| **Txn\_Date** |
| **05-Jan-1999** |
| **07-Jan-1999** |
| **08-Jan-1999** |
| **10-Jan-1999** |
| **11-Jan-1999** |
| **12-Jan-1999** |

Notez qu’en saisissant « **SELECT DISTINCT Txn\_Date**» pour les deux instructions SQL, nous obtiendrons le même jeu de résultats.

### **Union All**

L’objectif de la commande **UNION ALL** de SQL est aussi de combiner ensemble les résultats de deux requêtes. La différence entre **UNION ALL** et **UNION** est la suivante : [**UNION**](http://www.1keydata.com/fr/sql/sql-union.php) sélectionne seulement des valeurs distinctes et **UNION ALL** sélectionne toutes les valeurs.

La syntaxe pour **UNION ALL** est comme suit :

**[instructions SQL 1]  
UNION ALL  
[instructions SQL 2];**

Utilisons l’exemple de la section précédente pour illustrer la différence. Supposons que nous avons les deux tables suivantes:

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| **Store\_Name** | **Sales** | **Txn\_Date** |
| Los Angeles | 1500 | 05-Jan-1999 |
| San Diego | 250 | 07-Jan-1999 |
| Los Angeles | 300 | 08-Jan-1999 |
| Boston | 700 | 08-Jan-1999 |

Table ***Internet\_Sales***

|  |  |
| --- | --- |
| **Txn\_Date** | **Sales** |
| 07-Jan-1999 | 250 |
| 10-Jan-1999 | 535 |
| 11-Jan-1999 | 320 |
| 12-Jan-1999 | 750 |

et pour retrouver toutes les dates où il y a des ventes au magasin et aussi sur Internet, il faudra utiliser l’instruction SQL suivante:

**SELECT Txn\_Date FROM Store\_Information  
UNION ALL  
SELECT Txn\_Date FROM Internet\_Sales;**

Résultat:

|  |
| --- |
| **Txn\_Date** |
| **05-Jan-1999** |
| **07-Jan-1999** |
| **08-Jan-1999** |
| **08-Jan-1999** |
| **07-Jan-1999** |
| **10-Jan-1999** |
| **11-Jan-1999** |
| **12-Jan-1999** |

### **Intersect**

À l’instar de la commande [**UNION**](http://www.1keydata.com/fr/sql/sql-union.php), **INTERSECT** opère aussi sur deux instructions SQL. La différence entre les deux commandes est la suivante : **UNION** agit essentiellement comme un opérateur **OR (OU)** operator (la valeur est sélectionnée si elle apparaît dans la première ou la deuxième instruction), la commande **INTERSECT** agit comme un opérateur **AND (ET)** operator (la valeur est sélectionnée seulement si elle apparaît dans les deux instructions).

La syntaxe est comme suit :

**[instructions SQL 1]  
INTERSECT  
[instructions SQL 2];**

Supposons que nous avons les deux tables suivantes :

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| **Store\_Name** | **Sales** | **Txn\_Date** |
| Los Angeles | 1500 | 05-Jan-1999 |
| San Diego | 250 | 07-Jan-1999 |
| Los Angeles | 300 | 08-Jan-1999 |
| Boston | 700 | 08-Jan-1999 |

Table ***Internet\_Sales***

|  |  |
| --- | --- |
| **Txn\_Date** | **Sales** |
| 07-Jan-1999 | 250 |
| 10-Jan-1999 | 535 |
| 11-Jan-1999 | 320 |
| 12-Jan-1999 | 750 |

et pour retrouver toutes les dates où il y a les deux types de ventes : au magasin et sur Internet, il faudra utiliser l’instruction SQL suivante :

**SELECT Txn\_Date FROM Store\_Information  
INTERSECT  
SELECT Txn\_Date FROM Internet\_Sales;**

Résultat :

|  |
| --- |
| **Txn\_Date** |
| **07-Jan-1999** |

Notez que la commande **INTERSECT** ne retournera que des valeurs distinctes.

### **Minus ﻿**

**MINUS** opère sur deux instructions SQL. Elle prend tous les résultats de la première instruction SQL, puis soustrait ceux de la deuxième instruction SQL pour obtenir la réponse finale. Si la deuxième instruction SQL comprend des résultats qui ne sont pas inclus dans la première instruction SQL, ils seront ignorés.

La syntaxe est comme suit :

**[instructions SQL 1]  
MINUS  
[instructions SQL 2];**

Poursuivons avec le même exemple :

Table ***Store\_Information***

|  |  |  |
| --- | --- | --- |
| **Store\_Name** | **Sales** | **Txn\_Date** |
| Los Angeles | 1500 | 05-Jan-1999 |
| San Diego | 250 | 07-Jan-1999 |
| Los Angeles | 300 | 08-Jan-1999 |
| Boston | 700 | 08-Jan-1999 |

Table ***Internet\_Sales***

|  |  |
| --- | --- |
| **Txn\_Date** | **Sales** |
| 07-Jan-1999 | 250 |
| 10-Jan-1999 | 535 |
| 11-Jan-1999 | 320 |
| 12-Jan-1999 | 750 |

et pour trouver toutes les dates où il y a des ventes au magasin, mais pas sur Internet, il faudra utiliser l’instruction SQL suivante :

**SELECT Txn\_Date FROM Store\_Information  
MINUS  
SELECT Txn\_Date FROM Internet\_Sales;**

Résultat:

|  |
| --- |
| **Txn\_Date** |
| **05-Jan-1999** |
| **08-Jan-1999** |

'05-Jan-1999', '07-Jan-1999',et '08-Jan-1999' sont les valeurs distinctes retournées à partir de **SELECT Txn\_Date FROM Store\_Information**. '07-Jan-1999' est également retourné à partir de la deuxième instruction SQL, **SELECT Txn\_Date FROM Internet\_Sales**, il sera donc exclu du jeu de résultats final.

Notez que la commande **MINUS** ne retournera que des valeurs distinctes.

Certaines bases de données peuvent utiliser **EXCEPT** au lieu de **MINUS**. Pour une utilisation correcte, consultez la documentation de votre base de données.

# **RÉFÉRENCE :**

* + <http://www.1keydata.com/fr/sql/sql-create-table.php>
  + <http://sql.sh/cours/select>
  + <http://www.1keydata.com/fr/sql/sql-select.php>
  + <http://www.1keydata.com/fr/sql/>
  + <http://www.1keydata.com/fr/sql/sql-avance.php>
  + <http://docs.postgresql.fr/8.3/sql-commands.html>