Sql

et

DB2

Document en cours de relecture

Eléments de cours donnés depuis 2011

Maurice Grossel

# Présentation

**SQL** (Structured Query Language, en français Langage de Requête Structurée) est un langage qui a des normes, mais certaines requêtes ne font pas partie du corpus des normes et sont propres à certains éditeurs.

L'ampleur des données à gérer a amené à créer un nouveau concept : la base de données.

Alors qu'un fichier est une suite de données, une base de données est une collection organisée d'informations structurées, en général contrôlée par un Système de Gestion de Base de Données (**SGBD**) et souvent accessible non pas directement mais à l'aide d'un logiciel spécifique.

Une base de données relationnelle est un type de base de données où les données sont liées à d'autres informations de cette base de données. Elles sont organisées en tables à deux dimensions et les tables peuvent être reliées entre elles. Le SGBD devient un **SGBDR** (Système de Gestion de Base de Données Relationnelle).

Un langage spécifique et normalié permet d'interroger et de modifier les données : **SQL** (Structured Query Language, en français Langage de Requête Structurée).

Le principe d'une base de données relationnelle est d'avoir :

- une seule localisation pour une donnée, donc un seul nom, une seule description et une absence de redondance,

- une absence de donnée calculée, autrement dit, ne pas stocker d'information déductible,

- uniquement des informations élémentaires. Donc pas de zone de groupe contrairement à Cobol,

- une organisation en tables à deux dimensions, les colonnes et les lignes, avec d'éventuelles relations entre les tables.

Par exemple, dans une table consacrée aux articles, on aura :

- une colonne "identifiant", une autre "libellé", une autre "prix", ce qui fait trois colonnes,

- un ensemble de lignes. Si l'entreprise vend 100 articles différents, elle aura donc 100 lignes dans sa table,

- une éventuelle clé primaire portant une information discriminante. Souvent c'est un numéro.

- d'éventuelles clés étrangères pour marquer ou non la relation avec d'autres tables. Par exemple, si l'article appartient à une seule famille, on aura une table "famille" avec un identifiant et d'autres informations (libellé, etc) et dans la table article, on aura l'identifiant famille,

- une ou plusieurs éventuelles relations entre la table, dite fille, et des tables mères dont elle dépend (exemple la table "article" est fille de la table "famille") indiquant l'action que doit entreprendre le SGBD :

- ON DELETE CASCADE. Si on supprime une famille, tous ses articles sont supprimés.

- ON DELETE RESTRICT. On ne peut supprimer une famille que si elle n'a pas d'article.

- ON DELETE SET NULL. Si on supprime une famille, la valeur de cette famille dans la colonne "famille" de la table article passera à NULL (on explique plus bas la notion de NULL),

- une table peut avoir aucune, une ou plusieurs tables filles et/ou aucune, une ou plusieurs tables mères,

- une table fille peut être fille d'une table mère qui est elle-même table fille. Si une table est elle-même directement ou indirectement mère d'une table dont elle est fille, le modèle conceptuel est à revoir.

Autres particularités :

- Une colonne peut être NULL, c'est à dire qu'elle est vide, ce qui est différent de "blanc" contrairement à Cobol. Un indicateur signale si la colonne est vide ou pas.

- Une colonne peut avoir une longueur variable, contrairement à Cobol.

- Une donnée ne peut pas être répétitive, contrairement à Cobol et à son OCCURS.

**Remarques**

Pour des raisons de performance, on peut être amené à déroger à ces règles et à créer des redondances ou à stocker des données calculées.

# La conception

* 1. La création d'une base de données n'est pas évidente. Il faut recencer les données, éliminer les synonymes, les équivocités, les dépendances d'une donnée vis à vis d'une autre et trouver un identifiant. La méthodologie Merise est bien appropriée et l'application des formes normales est un outil précieux.

## Formes normales

Une normalisation très utilisée est connue sous la suite de "formes normales". Elle permet lors de la création de réduire voire d'éliminer les doublons, et lors de l'exploitation de minimiser les incohérences pendant les mises à jour et de simplifier les requêtes.

Une forme normale désigne un type de relation particulier entre les entités.

La forme normale vient après la simple validité d'un modèle relationnel, c'est-à-dire après avoir vérifié la dépendance fonctionnelle des attributs avec la clé primaire, autrement dit après vérification que la clé primaire est bien un déterminant.

Les formes normales s'emboitent les unes dans les autres, tant et si bien que le respect d'une forme normale d'un niveau implique le respect des formes normales précédentes. Des huit formes normales, on s'arrête le plus souvent aux trois premières :

La liste et leur codification en anglais :

* la première forme normale notée 1FN (1NF) ;
* la deuxième forme normale notée 2FN (2NF) ;
* la troisième forme normale notée 3FN (3NF) ;
* la forme normale de Boyce Codd notée FNBC (BCNF) ;
* la quatrième forme normale notée 4FN (4NF) ;
* la cinquième forme normale notée 5FN (5NF) ;
* la forme normale domaine clé notée FNDC (DKNF) ;
* la sixième forme normale notée 6FN (6NF)
  + rarement présentée*.*

### 1FN – Première forme normale

Il faut définir les données à stocker, leur attribuer le type de chacune et les placer chacune dans une table propre.

Les données, portées par les colonnes, :

* sont toutes atomiques,
* contiennent une valeur scalaire (les valeurs ne peuvent pas être divisées en plusieurs sous-valeurs dépendant également individuellement de la clé primaire),
* contiennent des valeurs non répétitives,
* sont constantes dans le temps.

*Exemple de valeur atomique :*

**Proposition erronée :**

|  |  |
| --- | --- |
| Produit (clé) | Fournisseur |
| Jean | Labellefringue, Toutpourlespantalons |

Dans ce cas les valeurs du fournisseur ont plusieurs valeurs et ne sont pas atomiques.

**Solution** : la première forme normale impose de décomposer les attributs de la colonne fournisseur ainsi :

|  |  |
| --- | --- |
| Produit (clé) | Fournisseur |
| Jean | Labellefringue |
| Jean | Toutpourlespantalons |

*Exemple de valeur atomique :*

**Proposition erronée :**

* tout attribut contient une valeur atomique.

|  |  |
| --- | --- |
| Fournisseur\_ID (clé) | … |
| Labellefringue Paris |  |
| Toutpourlespantalons Lille |  |

L'attibut Fournisseur\_ID est composé de 2 attibuts atomiques. Il faut dédoubler la colonne.

**Solution** :

|  |  |
| --- | --- |
| Fournisseur\_ID (clé) | … |
| Labellefringue | Paris |
| Toutpourlespantalons | Lille |

*Exemple d'attributs répétitifs :*

***Proposition erronée***

* tous les attributs sont non répétitifs

|  |  |
| --- | --- |
| Produit (clé) | Fournisseur |
| Jean | Labellefringue, Toutpourlespantalons |

L'attribut Fournisseur est une liste. Il faut ajouter autant de lignes que nécessaires pour la colonne ne contienne plus de liste.

*Exemple d'éléments non costants dans le temps :*

***Proposition erronée***

* tous les attributs sont constants dans le temps.

|  |  |  |
| --- | --- | --- |
| Fournisseur (clé) | NOM | Ancienneté |
| 1 | Labellefringue | 35 |

L'attibut Ancienneté n'est pas constant dans le temps. Il faut changer la colonne.

**Solution** :

|  |  |  |
| --- | --- | --- |
| Fournisseur (clé) | NOM | Année de référence |
| 1 | Labellefringue | 2018 |

### 2FN – Deuxième forme normale

Pour être de la deuxième forme normale, la relation doit respecter la première forme normale et le principe suivant : un attribut non clé ne dépend pas d'une partie de la clé.

Le non-respect de la 2FN entraîne une redondance des données qui encombrent alors inutilement la mémoire et l'espace disque.

Exemple :

***Proposition erronée***

|  |  |  |
| --- | --- | --- |
| Produit (clé) | Fournisseur (clé) | Adresse fournisseur |
| Jean | Labellefringue | 15 rue du mail 49000 Angers |
| Short | Labellefringue | 15 rue du mail 49000 Angers |
| Jean | Toutpourlespantalons | 1 place du pilori 49000 Angers |

Si la clé de cette table est une clé composite (produit + fournisseur), le changement d'adresse d'un fournisseur implique de modifier sur plusieurs lignes. En effet, la colonne adresse ne dépend que d'une partie de la clé : la colonne fournisseur, d'où une redondance dans la table.

**Solution** : en seconde forme normale : on en déduit la création de deux tables.

|  |  |
| --- | --- |
| Produit (clé) | Fournisseur |
| Jean | Labellefringue |
| Short | Labellefringue |
| Jean | Toutpourlespantalons |

|  |  |
| --- | --- |
| Fournisseur (clé) | Adresse fournisseur |
| Labellefringue | 15 rue du mail 49000 Angers |
| Toutpourlespantalons | 1 place dupilori 49000 Angers |

Le changement d'adresse d'un fournisseur se fera sur une seule ligne de la table des fournisseurs.

* tous les attributs non-clés sont totalement dépendants fonctionnellement de la totalité de la clé primaire.

|  |  |  |
| --- | --- | --- |
| Ligne\_commande (clé) | Article (clé) | Description\_article |
| 1 | 15 | Jean cintré taille 46 |

La Description\_article ne dépend que d'une partie de la clé.

### 3FN – Troisième forme normale

Dans la troisième forme normale, la relation doit respecter la seconde forme normale et le principe suivant  : un attribut non clé ne dépend pas d'un ou plusieurs attributs ne participant pas à la clé.

Exemple:

|  |  |  |
| --- | --- | --- |
| Fournisseur (clé) | Adresse fournisseur | Pays |
| Labellefringue | 15 rue du mail 49000 Angers | France |
| Toutpourlespantalons | 1 place dupilori 49000 Angers | France |
| La ropahermosa | Pla de la Seu, s/n, 08002 Barcelona | Espagne |

Le pays de l'adresse ne dépendant pas de la clé de la table qui est le fournisseur, mais de la ville de l'adresse. Il faut utiliser deux tables.

Solution normalisée :

|  |  |  |
| --- | --- | --- |
| Fournisseur (clé) | Adresse fournisseur | Ville |
| Labellefringue | 15 rue du mail 49000 | Angers |
| Toutpourlespantalons | 1 place dupilori 49000 | Angers |
| La ropahermosa | Pla de la Seu, s/n, 08002 | Barcelona |

|  |  |
| --- | --- |
| Ville (clé) | Pays |
| Angers | France |
| Barcelona | Espagne |

Ainsi la modification du nom d'un pays n'impacte qu'une seul ligne.

Exemple de changement de nom de pays : la Haute Volta est devenue le Burkina Faso.

* tout attribut n'appartenant pas à une clé ne dépend pas d'un attribut non clé

|  |  |  |
| --- | --- | --- |
| Entete\_commande (clé) | Fournisseur | Nom\_fournisseur |
| 1 | 15 | Labellefringue |

### FNBC – Forme normale de Boyce-Codd

La 3FN ne garantit pas l'absence de redondance des données d'où la FNBC.

Dans la forme normale de Boyce-Codd, la relation doit respecter la troisième forme normale et que tous les attributs non-clé ne sont pas source de dépendance fonctionnelle (DF) vers une partie de la clé.

Le non-respect de la 2FN, 3FN et la FNBC entraîne de la redondance, une même donnée étant répétée un nombre considérable de fois.

### 4FN - quatrième forme normale

Pour être en 4FN, il faut respecter la 3FN (et pas forcément en FNBC)

Pour toute relation de dimension *n* en forme normale de Boyce-Codd, les relations de dimension *n-1* construites sur sa collection doivent avoir un sens. Il ne doit pas être possible de reconstituer les occurrences de la relation de dimension *n* par jointure de deux relations de dimension *n-1*. Cette normalisation conduit parfois à décomposer une relation complexe en deux relations plus simples.

### 5FN - cinquième forme normale

Pour toute relation de dimension *n* (avec *n* supérieur à 2) en quatrième forme normale, il ne doit pas être possible de retrouver l’ensemble de ses occurrences par jointure sur les occurrences des relations partielles prises deux à deux. Cette normalisation conduit parfois à décomposer une relation complexe en plusieurs relations plus simples.

*Le non-respect de la 4FN et 5FN entraîne de la perte de données et les données manquent de précision.*

### FNDC - forme normale domaine clé

Une relation est en FNDC si et seulement si toutes les contraintes sont la conséquence logique des contraintes de domaines et des contraintes de clés qui s'appliquent à la relation.

Les caractéristiques des trois premières formes normales :

1FN = La clé. 2FN = Toute la clé. 3FN = Rien que la clé.

* Si une entité ou une relation en troisième forme normale a une clé composée, aucune des propriétés élémentaires de cette clé ne doit être en dépendance fonctionnelle d’une autre propriété.

|  |  |  |
| --- | --- | --- |
| Commercial\_ID (clé) | Secteur\_ID (clé) | Ville\_ID (clé) |
| DUVAL | Nord | Lille |
| DUPONT | Ouest | Nantes |

Si un des commerciaux part, on perd la relation Secteur et Ville.

**FNDC - forme normale domaine clé**

* Une relation est en FNDC si et seulement si toutes les contraintes sont la conséquence logique des contraintes de domaines et des contraintes de clés qui s'appliquent à la relation.

Soit la relation VEHICULE, avec les attributs suivants :

|  |  |  |  |
| --- | --- | --- | --- |
| CONSTRUCTEUR\* | MODELE\* | TYPE | PTAC (KG) |
| Renault | Estafette | VL | 2500 |
| Iveco | Eurostar 440 | PL | 19000 |
| Berliet | GDM 1934 | PL | 15000 |
| VolksWagen 2 900 | combi | VL | 2 900 |

*On remarque que le type VL (véhicule léger) ou PL (poids lourd) est déterminé par la valeur du PTAC. Ainsi, au-dessus de 3,5 tonnes le véhicule est un PL. En dessous c'est un VL… Il y a redondance des données de type qui peut être déduite de la lecture de la valeur du PTAC. En cas de changement de la réglementation (barre des 3,5 tonnes qui pourrait être amenée à changer) alors il faut mettre à jour plusieurs n-uplets !* - Pour résoudre cette anomalie de mise à jour, il faut décomposer la relation en deux comme suit :

**1° VEHICULE, avec les attributs suivants :**

|  |  |  |
| --- | --- | --- |
| CONSTRUCTEUR\* | MODELE\* | PTAC (KG) |
| Renault | Estafette | 2500 |
| Iveco | Eurostar 440 | 19000 |
| Berliet | GDM 1934 | 15000 |
| VolksWagen 2 900 | combi | 2 900 |

*Le type de véhicule ne figure plus. Il sera déduit de la valeur du PTAC : au-dessus de 3,5 tonnes le véhicule est un PL. En dessous c'est un VL.*

**2° TYPE VEHICULE, avec les attributs suivants :**

|  |  |
| --- | --- |
| TYPE\* | PTAC (KG) |
| VL | 0 |
| PL | 3500 |

*Une inéqui-jointure sera nécessaire à reconstituer la relation originale.*

# Gestion du contenant

Une table est le contenant ordonné des données. C'est un ensemble de colonnes et de lignes, avec éventuellement un ou plusieurs index, et éventuellement un ou plusieurs liens vers d'autres tables.

Une colonne est un sous-ensemble d'une table pour une donnée en particulier, par exemple le nom d'un employé, ou autre exemple de colonne pour le nom de son service.

Pour définir une **colonne (column en langage SQL)**, on indique :

* son nom,
* son type (numérique, caractère, date, etc),
* sa longueur,
* si sa longueur est fixe ou variable,
* si elle doit toujours être renseignée ou non (dans ce cas on dit NULL),
* si, lors de l'insertion d'une ligne, la colonne est créée avec une valeur par défaut lorsque aucune valeur n'est spécifiée.

Une **ligne** **(row en langage SQL)** est l'ensemble des colonnes d'une table pour une instance. Par exemple le nom, le service, etc pour l'employé "Dupont", et une autre ligne pour chacun des employés.

Un **index** **(index en langage SQL)** est une structure de données créée sur une table à partir d'une ou de plusieurs de ses colonnes pour accélérer le traitements.

Une **requête** **(request en langage SQL)** est une instruction normalisée qui permet de lire des informations d'une base de données, ou de les mettre à jour, ou de les supprimer.

Une **vue** **(view en langage SQL)** est un ensemble virtuel de colonnes d'une ou de plusieurs tables destiné à faciliter l'écriture des requêtes. Elle est créée en définissant la requête sur les colonnes des tables physiques. Elle n'est pas plus performante que cette requête.

Un **trigger** est un script qui se déclenche dans certaines conditions et qui va exécuter une action. Par exemple, on peut créer un trigger qui va se déclencher lors de la modification de la table article et qui va mettre à jour la colonne "date de dernière mise à jour". L'évènement peut être porté par une table et modifier une autre table.

## Commandes SQL

Les commandes SQL permettent de créer, de modifier et de supprimer les tables et leur environnement :

* **CREATE** : pour créer une entité (TABLE, INDEX, VIEW)
* **DROP** : pour supprimer (TABLE, INDEX, VIEW)
* **ALTER** : pour modifier la structure d'une table, une colonne, pour ajouter un index, ...
* **GRANT** et **REVOKE** : pour donner ou annuler les droits d'accès d'une table à d'autres utilisateurs que son créateur.

Malgré la normalisation de SQL, il existe des variantes d'un SGBD à l'autre sur l'écriture de ces mises à jour et sur leurs étendues (possibilité ou nom de modifié une colonne, etc). Il existe aussi des objets présents chez un SGBD et absents chez un autre.

## ****Commande CREATE****

### **Commande CREATE DATABASE**

La création d’une base de données en SQL est possible en ligne de commande. Même si les systèmes de gestion de base de données (SGBD) sont souvent utilisés pour créer une base, il convient de connaître la commande à utiliser, qui est très simple.

**Syntaxe**

Pour créer une base de données qui sera appelé « ma\_base » il suffit d’utiliser la requête suivante qui est très simple:

**CREATE DATABASE** ma\_base

**Base du même nom qui existe déjà**

Avec MySQL, si une base de données porte déjà ce nom, la requête retournera une erreur. Pour éviter d’avoir cette erreur, il convient d’utiliser la requête suivante pour MySQL:

**CREATE DATABASE IF NOT EXISTS** ma\_base

L’option IF NOT EXISTS permet juste de ne pas retourner d’erreur si une base du même nom existe déjà. La base de données ne sera pas écrasée.

**Options**

Dans le standard SQL la commande CREATE DATABASE n’existe pas. Il faut donc vérifier la documentation des différents SGBD pour vérifier les syntaxes possibles et les options disponibles : jeux de caractères, propriétaire de la base, les limites de connexion ...

### Commande CREATE TABLE

La commande CREATE TABLE permet de créer une table en SQL. Une table est une entité qui est contenu dans une base de données pour stocker des données ordonnées dans des colonnes. La création d’une table sert à définir les colonnes et le type de données qui seront contenus dans chacun des colonnes (entier, chaîne de caractères, date, valeur binaire …).

**Syntaxe**

La syntaxe générale pour créer une table est la suivante :

**CREATE TABLE** nom\_de\_la\_table

(

colonne\_1 type\_donnees,

colonne\_2 type\_donnees,

colonne\_3 type\_donnees,

colonne4 type\_donnees

)

Dans cette requête, 4 colonnes ont été définies. Le mot-clé «type\_donnees» sera remplacé par un mot-clé pour définir le type de données (INT, DATE, TEXT …). Pour chaque colonne, il est également possible d'ajouter des options telles que (liste non-exhaustive):

* **NOT NULL :** empêche d’enregistrer une valeur nulle pour une colonne.
* **DEFAULT :** attribue une valeur par défaut si aucune donnée n’est indiquée pour cette colonne lors de l’ajout d’une ligne dans la table.
* **PRIMARY KEY :** indique si cette colonne est considérée comme clé primaire pour un index.

**Exemple**

Imaginons que l’ont souhaite créer une table utilisateur, dans laquelle chaque ligne correspond à un acheteur inscrit sur un site web. La requête pour créer cette table peut ressembler à ceci:

**CREATE TABLE** acheteur

(

id **INT PRIMARY KEY NOT NULL**,

nom **VARCHAR**(100),

prenom **VARCHAR**(100),

email **VARCHAR**(255),

date\_naissance **DATE**,

pays **VARCHAR**(255),

ville **VARCHAR**(255),

code\_postal **VARCHAR**(5),

nombre\_achat **INT**

)

Voici des explications sur les colonnes créées :

* **id :** identifiant unique. C'est un entier non nul utilisé comme clé primaire
* **nom :** nom de l’acheteur dans une colonne de type VARCHAR. Le nombre entre parenthèse indique le maximum, qui est ici de 100 caractères
* **prenom :** idem mais pour le prénom
* **email :** adresse email. Même principe : 255 caractères au maximum. Il n'existe pas de propriété spécifique qui vérifie la présence d'un @. Il faut des contrôles spécifiques.
* **date\_naissance :** date de naissance enregistré au format AAAA-MM-JJ (exemple : 1973-11-17)
* **pays :** nom du pays de l’utilisateur sous 255 caractères au maximum
* **ville :** idem pour la ville
* **code\_postal :** 5 caractères du code postal
* **nombre\_achat :** nombre d’achats de cet utilisateur sur le site

### Commande CREATE INDEX

En SQL, la commande CREATE INDEX permet de créer un index.

Index SQL

L’index est utile pour accélérer l’exécution d’une requête SQL qui lit des données et ainsi pour améliorer les performances d’une application utilisant une base de données.

### Analogie pour comprendre les index en SQL

Un index, dans le domaine bibliographique, permet de lister les mots-clés importants abordés dans un ouvrage et d’indiquer les pages où le mot est mentionné. Ainsi, un lecteur qui recherche une thématique spécifique peut se baser sur cet index pour trouver les pages qui abordent le sujet. De même un index est une ressource non indispensable, mais c’est un un gain de temps énorme pour l’utilisateur qui accède facilement à l’information recherchée.

#### Index en SQL

Un index, dans une base de données se base sur le même principe qu’un index dans un livre. Avec un index placé sur une ou plusieurs colonnes le système d’une base de données peut rechercher les données d’abord sur l’index et s’il trouve ce qu’il cherche il saura plus rapidement où se trouve les enregistrements concernés.

Ces petites ressources ont toutefois leurs inconvénients car cela occupe de l’espace supplémentaire dans la base de données. Par ailleurs, l’insertion de données est plus long car les index sont mis à jour à chaque fois que des données sont insérées.

Généralement un index pourra être utilisé dans les requêtes utilisant les clauses WHERE, GROUP BY ou ORDER BY. Lorsqu’une base de données possède un grand nombre d’enregistrements (exemple: plusieurs milliers ou plusieurs millions de lignes) un index permet de gagner un temps précieux pour la lecture de données.

#### Créer un index ordinaire

La syntaxe basique pour créer un index est la suivante :

**CREATE INDEX** `index\_nom` **ON** `table`;

Il est également possible de créer un index sur une seule colonne en précisant la colonne sur laquelle doit s’appliquer l’index :

**CREATE INDEX** `index\_nom` **ON** `table` (`colonne\_1`);

L’exemple ci-dessus va donc insérer l’index intitulé «index\_nom» sur la table nommée «table» uniquement sur la colonne "colonne\_1″. Pour insérer un index sur plusieurs colonnes il est possible d’utiliser la syntaxe suivante:

**CREATE INDEX** `index\_nom`

**ON** `table` (`colonne\_1`, `colonne\_2`);

L’exemple ci-dessus permet d’insérer un index les 2 colonnes : colonne\_1 et colonne\_2.

#### Créer un index unique

Un index unique permet de spécifier qu’une ou plusieurs colonnes doivent contenir des valeurs uniques à chaque enregistrement. Le système de base de données retournera une erreur si une requête tente d’insérer des données qui feront doublons sur la clé d’unicité. Pour insérer un tel index il suffit d’exécuter une requête SQL respectant la syntaxe suivante :

**CREATE UNIQUE INDEX** `index\_nom`

**ON** `table` (`colonne\_1`);

Dans cet exemple un index unique sera créé sur la colonne nommée **colonne\_1**. Cela signifie qu’il ne peut pas y avoir plusieurs fois la même valeur sur 2 enregistrements distincts contenus dans cette table.

Il est également possible de créer un index d’unicité sur 2 colonnes, en respectant la syntaxe suivante:

**CREATE UNIQUE INDEX** `index\_nom`

**ON** `table` (`colonne\_1`, `colonne\_2`);

#### Convention de nommage

Le nommage des noms d'index est libre, mais voici une liste de suggestions de préfixes souvent rencontrés :

* Préfixe « PK\_ » pour **P**rimary **K**ey (traduction : clé primaire)
* Préfixe « FK\_ » pour **F**oreign **K**ey (traduction : clé étrangère)
* Préfixe « UK\_ » pour **U**nique **K**ey (traduction : clé unique)
* Préfixe « UX\_ » pour **U**nique **I**ndex (traduction : index unique)
* Préfixe « IX\_ » pour chaque autre **I**nde**X**

**Remarque :**

**Si la table existe, il faut la supprimer par DROP.**

**Les lignes ou le reste des lignes commençant par – sont des commentaires.**

**-- DROP TABLE API16.FAM ;**

**-- COMMIT ;**

**CREATE TABLE API16.FAM (**

**FAMCOD CHAR(2) PRIMARY KEY NOT NULL,**

**FAMLIB CHAR(20) DEFAULT '?',**

**FAMCRE TIMESTAMP,**

**FAMMAJ TIMESTAMP**

**)**

**-- IN DBMATE2.TSLEMA02**

**;**

**CREATE UNIQUE INDEX**

**UI\_API16\_FAM**

**ON API16.FAM (FAMCOD)**

**;**

**GRANT SELECT ON API16.FAM TO PUBLIC**

**;**

### Commande CREATE VIEW

Une **vue** (VIEW en SQL) est une synthèse d'une requête d'interrogation de la base. On peut la voir comme une table virtuelle, définie par une requête.

Les avantages des vues sont :

* d'éviter de taper une requête très longue : la vue sert à donner un nom à la requête pour l'utiliser souvent,
* de masquer certaines données à certains utilisateurs. En SQL, les protections d'une vue ne sont pas forcément les mêmes que celles des tables sous-jacentes.

**CREATE** **VIEW** V\_PERSMAIS **AS**

**SELECT** CODEP, CODERESP

**FROM** PERSONNE P, MAISON M

**WHERE** P.CODEP = M.CODERESP

;

**Les vues se créent avec la commande CREATE VIEW. Un exemple, où les employés d'une entreprise sont dans une table, les départements de l'entreprise dans une autre et où on doit faire une jointure pour afficher le nom du département à côté de celui de l'employé :**

1. CREATE TABLE Employes (id SERIAL, nom TEXT, departement INTEGER);
2. CREATE TABLE Departements (id SERIAL, nom TEXT);
3. SELECT e.nom as Employe, d.nom as Departement
4. FROM Employes e,Departements d WHERE e.departement = d.id;

Si, par contre, on crée une vue :

1. CREATE VIEW ToutLeMonde AS
2. SELECT e.nom as Employe, d.nom as Departement
3. FROM Employes e,Departements d
4. WHERE e.departement = d.id;

On pourra alors écrire la requête SELECT ci-dessus bien plus simplement, la jointure ne sera plus visible :

1. SELECT \* FROM ToutLeMonde ;

Les **vues** s'utilisent pratiquement comme des tables (elles peuvent être dans une clause FROM d'un SELECT, dans un UPDATE, etc) avec quelques restrictions, qui dépendent du SGBD.

## ****Commande DROP****

### **Commande DROP DATABASE**

En SQL, la commande DROP DATABASE permet de supprimer totalement une base de données et tout ce qu’elle contient. Cette commande est à utiliser avec beaucoup d’attention car elle permet de supprimer tout ce qui est inclus dans une base: les tables, les données, les index …

**Syntaxe**

Pour supprimer la base de données «ma\_base», la requête est la suivante :

**DROP DATABASE** ma\_base

**Attention :** cela va supprimer toutes les tables et toutes les données de cette base. Si vous n’êtes pas sûr de ce que vous faites, n’hésitez pas à effectuer une sauvegarde de la base avant de supprimer.

**Ne pas afficher d’erreur si la base n’existe pas**

Par défaut, si le nom de base utilisé n’existe pas, la requête retournera une erreur. Pour éviter d’obtenir cette erreur si vous n’êtes pas sûr du nom, il est possible d’utiliser l’option IF EXISTS. La syntaxe sera alors la suivante:

**DROP DATABASE IF EXISTS** ma\_base

### **Commande DROP TABLE**

La commande DROP TABLE en SQL permet de supprimer définitivement une table d’une base de données. Cela supprime en même temps les éventuels index, trigger, contraintes et permissions associées à cette table.

**Attention :** il faut utiliser cette commande avec attention car une fois supprimée, les données sont perdues. Avant de l’utiliser sur une base importante il peut être judicieux d’effectuer un backup (une sauvegarde) pour éviter les mauvaises surprises.

**Syntaxe**

Pour supprimer une table « nom\_table » il suffit simplement d’utiliser la syntaxe suivante :

**DROP TABLE** nom\_table ;

**A savoir :** s’il y a une dépendance avec une autre table, il est recommandé de les supprimer avant de supprimer la table. C’est le cas par exemple s’il y a des clés étrangères.

**Intérêts**

Il arrive qu’une table soit créée temporairement pour stoquer des données qui n’ont pas vocation à être ré-utiliser. La suppression d’une table non utilisée est avantageux sur plusieurs aspects :

* **Libérer de la mémoire** et alléger le poids des backups
* **Éviter des erreurs** dans le futur si une table porte un nom similaire ou qui porte à confusion
* Lorsqu’un développeur ou administrateur de base de données découvre une application, il est **plus rapide de comprendre le système** s’il n’y a que les tables utilisées qui sont présentes.

**Exemple de requête**

Imaginons qu’une base de données possède une table "client\_2009" qui ne sera plus jamais utilisé et qui existe déjà dans un ancien backup. Pour supprimer cette table, il suffit d’effectuer la requête suivante :

**DROP TABLE** client\_2009 ;

L'éxécution de cette requête va permettre de supprimer la table.

**DROP TABLE** VENTES ;

## Commande GRANT

**La syntaxe est GRANT (sur quelle instruction) ON (table ou vue) TO (quel utilisateur) :**

****GRANT** SELECT **ON** TABLE MATA01.PERSONNE **TO** **PUBLIC** ;**

****GRANT** ALL **ON** TABLE MATA01.ARTICLE **TO** **PUBLIC** ;**

****GRANT** SELECT, INSERT **ON** TABLE COMMANDE**

****TO** **USER** AZER,USER QSDF;**

**Remarque : on ne peut pas donner de privilège sur les vues avec ALTER, INDEX, et REFERENCE.**

## Commande Revoke

**La syntaxe est REVOKE (sur quelle instruction) ON (table ou vue) FROM (quel utilisateur) (avec l'option GRANT)**

****REVOKE** SELECT **ON** TABLE PERSONNE **FROM** **PUBLIC** ;**

****REVOKE** ALL **ON** TABLE PERSONNE **FROM** AZERT, QSDF ;**

## ****Commande ALTER TABLE****

La commande ALTER TABLE en SQL permet de modifier une table existante. Il est ainsi possible d’ajouter une colonne, d’en supprimer une ou de modifier une colonne existante, par exemple pour changer le type.

Certains SGBD n'acceptent pas toute les possibilités ou exigent une syntaxe différente.

**Syntaxe de base**

D’une manière générale, la commande s’utilise de la manière suivante:

**ALTER TABLE** nom\_table instruction

Le mot-clé « instruction » ici sert à désigner une commande supplémentaire, qui sera détaillée ci-dessous selon l’action que l’ont souhaite effectuer : ajouter, supprimer ou modifier une colonne.

* + 1. **Ajouter une colonne**

L’ajout d’une colonne dans une table est relativement simple et peut s’effectuer à l’aide d’une requête ressemblant à ceci:

**ALTER TABLE** nom\_table

**ADD** nom\_colonne type\_donnees

Pour ajouter une colonne qui correspond à une rue sur une table utilisateur, il est possible d’utiliser la requête suivante:

**ALTER TABLE** utilisateur

**ADD** adresse\_rue VARCHAR(255)

* + 1. **Supprimer une colonne**

Une syntaxe permet également de supprimer une colonne pour une table. Il y a 2 manières totalement équivalente pour supprimer une colonne:

**ALTER TABLE** nom\_table **DROP** nom\_colonne

Ou (le résultat sera le même)

**ALTER TABLE** nom\_table

**DROP COLUMN** nom\_colonne

* + 1. **Modifier une colonne**

Pour modifier une colonne, comme par exemple changer le type d’une colonne, il y a différentes syntaxes selon le SGBD.

**ALTER TABLE** PERSONNE

**ADD COLUMN** NOM **CHAR(30)**;

**ALTER COLUMN** TESLIB

**SET DATA TYPE VARCHAR**(30);

**ALTER TABLE** FAMILLE

**ADD PRIMARY KEY** (FAMCOD);

**ALTER TABLE** ARTICLE

**ADD FOREIGN KEY**(FAMCOD) **REFERENCES** FAMILLE

**ON DELETE RESTRICT ;**

ou

**ON DELETE CASCADE** ;

ou

**ON DELETE SET NULL** ;

**ALTER TABLE** FAMILLE

**RENAME COLUMN** DATEMAJ **TO** MAJDATE;

Ici, le mot-clé « type\_donnees » est à remplacer par un type de données tel que INT, VARCHAR, TEXT, DATE …

* + 1. **Renommer une colonne**

Pour renommer une colonne, il convient d’indiquer l’ancien nom de la colonne et le nouveau nom de celle-ci.

**DB2**

Pour DB2, il faut également indiquer le type de la colonne.

**ALTER TABLE** nom\_table

**RENAME COLUMN** colonne\_ancien\_nom

**TO** colonne\_nouveau\_nom type\_donnees

## Format des colonnes

Les valeurs maximales et les caractéristiques des types de données en SQL peuvent varier légèrement selon le système de gestion de base de données (SGBD) utilisé, comme MySQL, PostgreSQL, SQL Server, etc.

En Cobol les nombres sont packés : PIC S9(n)V9(m) comp-3.

Cependant, voici une vue d'ensemble générale des types de données que vous avez mentionnés :

**INT (Integer)** mot, binaire, virgule fixe :

**Taille** : Généralement 4 octets.

**Valeur maximale** :

231−1231−1 soit 2.147.483.647 pour les entiers signés.

232−1232−1 soit 4.294.967.295 pour les entiers non signés .

**SMALLINT** demi-mot, binaire, virgule fixe :

**Taille** : Généralement 2 octets.

**Valeur maximale** :

215−1215−1 soit 32.767 pour les entiers signés

216−1216−1 soit 65.535 pour les entiers non signés.

**FLOAT** binaire, virgule flottante :

**Taille** : Généralement 4 octets pour une précision simple.

**Valeur maximale approximative** :

3,40282347×10383,40282347×1038 avec une précision d'environ 7 chiffres.

**REAL**  binaire, virgule flottante :

**Taille** : Généralement équivalent à FLOAT, soit 4 octets.

**Valeur maximale approximative** :

Similaire à FLOAT, 3,40282347×10383,40282347×1038.

**DECIMAL et NUMERIC** packé, S9(n-d)V9(d) COMP-3 :

**Taille** : Variable, définie par l'utilisateur avec une précision et une échelle spécifiques.

**Valeur maximale** : Dépend de la précision définie. Par exemple, DECIMAL(10,2) peut stocker des nombres jusqu'à 99.999.999,99.

**Attention** : contrairement à Cobol, le premier nombre indique le nombre total de chiffres, y compris les décimales :

DECIMAL(3,2) correspond à PIC 9,99.

**CHAR(n)** caractère, fixe PIC X(-) :

**Taille** : n octets, où n est le nombre de caractères spécifié.

**Valeur maximale** : Peut stocker jusqu'à 255 caractères dans la plupart des SGBD.

**VARCHAR(n)** caractère, PIC X(longueur maximum) :

**Taille** : Variable, jusqu'à n octets, où n est le nombre maximum de caractères spécifié.

**Valeur maximale** : Peut stocker jusqu'à un très grand nombre de caractères, souvent limité à 65,535 caractères dans de nombreux SGBD, mais cela peut varier.

Ces valeurs sont des indications générales et peuvent différer selon le SGBD spécifique que vous utilisez. Il est toujours recommandé de consulter la documentation de votre SGBD pour obtenir des informations précises.

Exemple de la colonne LUXLIB décrite en Cobol

LUXLIB VARCHAR(20) DEFAULT '?',

10 LUXLIB.

49 LUXLIB-LEN PIC S9(4) USAGE COMP.

49 LUXLIB-TEXT PIC X(20).

**DATE** date avec le siècle, 10 octets. '2017-03-28'

**TIME** heure, avec minute et seconde, 8 octets,

**TIMESTAMP** date avec heure, minute, seconde et 10 puissance -6 seconde, 26 octets

Exemple : le 14 juillet 2024 à 9 h 25mn 5s :

'2024-07-14-09.25.05:000000000'

Pour connaitre la date et l'heure, il faut une table avec une seule occurrence. IBM met à disposition la table sysibm.sysdummy1.

**On peut extraire**

|  |  |  |
| --- | --- | --- |
| Function | Description | Example |
| CHAR | Returns a string representation of its first argument. | CHAR(cust\_hiredate,USA) |
| DAYS | Returns an integer representation of its argument. | DAYS(‘2008-01-01’) |
| YEAR | Returns the year part of its argument | YEAR(cust\_hiredate) |
| MONTH | Returns the month part of its argument | MONTH(cust\_hiredate) |
| DAY | Returns the day part of its argument | DAY(cust\_hiredate) |
| HOUR | Returns the hour part of its argument | HOUR(CURRENT TIME) |
| MINUTE | Returns the minute part of its argument | MINUTE(CURRENT TIME) |
| SECOND | Returns the minute part of its argument | SECOND(CURRENT TIME) |
| MICROSECOND | Returns the microsecond part of its argument | MICROSECOND(CURRENT TIMESTAMP) |
| DATE | Returns the date derived from its argument | DATE(‘2008-01-01’) |
| TIME | Returns the time derived from its argument | TIME(’13:00:00’) |
| TIMESTAMP | Returns the timestamp derived from its argument | TIMESTAMP(CURRENT DATE) |

SELECT current date FROM sysibm.sysdummy1;

SELECT current time FROM sysibm.sysdummy1;

SELECT current timestamp FROM sysibm.sysdummy1;

SELECT YEAR (current timestamp) FROM sysibm.sysdummy1;

SELECT MONTH (current timestamp) FROM sysibm.sysdummy1;

SELECT DATE (current timestamp) FROM sysibm.sysdummy1;

SELECT TIME (current timestamp) FROM sysibm.sysdummy1;

SELECT DAY (current timestamp) FROM sysibm.sysdummy1;

SELECT HOUR (current timestamp) FROM sysibm.sysdummy1;

SELECT MINUTE (current timestamp) FROM sysibm.sysdummy1;

SELECT SECOND (current timestamp) FROM sysibm.sysdummy1;

SELECT MICROSECOND (current timestamp) FROM sysibm.sysdummy1;

**Manipulation des colonnes**

**LENGTH**(nom\_de\_la\_colonne) donne la longueur d'une colonne

SELECT LENGTH(DECCOLORIS)

FROM API16.DEC ;

**SUBSTR** (nom\_de\_la\_colonne,p **,l**g )

renvoie une partie de la colonne nom\_de\_la\_colonne depuis la positionp sur une longueur de **l**g

P commence à 1

Lg est facultatif = maximum

SELECT DECCOLORIS,

SUBSTR(DECCOLORIS, 2, 3 )

FROM API16.DEC ;

Table DECCOLORIS :

Contenu résultat

BLEU LEU

ROUGE OUG

# Gestion du contenu

## Commandes SQL

Les commandes SQL permettent de lire et de mettre à jour les tables :

INSERT : insérer des lignes

SELECT : lire des colonnes

UPDATE : mettre à jour des colonnes

DELETE : supprimer des lignes

Des fonctions propres à certains SGBD viennent enrichir ce panel basique.

Les commandes et fonctions peuvent être complétées de CLAUSE (exemple : WHERE) et les clauses d'OPERATEUR (exemples : =, IN).

Il existe aussi des MOTS-CLES, comme DISTINCT, SUM, etc.

**A noter :** comme en Cobol, lorsque le colonne 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 faut pas utiliser de guillemet, il suffit juste d’indiquer le nombre.

## Commande 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.

### 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 renseignant seulement une partie des colonnes.

### Insérer une ligne en spécifiant toutes les colonnes

La syntaxe pour remplir une ligne, sans donner le nom des colonnes, avec cette méthode est la suivante :

**INSERT INTO** 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 resté 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** table

(colonne\_1, colonne\_2, …)

**VALUES** ('valeur 1', 'valeur 2', ...)

**A noter :** il est possible de ne pas renseigner toutes les colonnes. De plus, l’ordre des colonnes dans INSERT peut être différent de l'ordre dans la table.

### Insertion de plusieurs lignes à la fois

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

**INSERT INTO** client (prenom, nom, ville, age)

**VALUES**

('Rébecca', 'Armand', 'Saint-Didier', 24),

('Aimée', 'Hebert', 'Marigny-le-Châtel', 36),

('Charles', 'Ribeiro', 'Maillères', 27),

('Hilaire', 'Savary', 'Conie-Molitard', 58);

Remarque : certain SGBD, comme DB2, n'accepte pas cet insertion multiple.

### **Instruction ON DUPLICATE KEY UPDATE**

L’instruction ON DUPLICATE KEY UPDATE est une fonctionnalité de MySQL qui permet de mettre à jour des données lorsqu’un enregistrement existe déjà dans une table. Cela permet d’avoir qu’une seule requête SQL pour effectuer selon la convenance un INSERT ou un UPDATE.

**Syntaxe**

Cette commande s’effectue au sein de la requête INSERT INTO avec la syntaxe suivante, avec "a" comme index primaire :

**INSERT INTO** table (a, b, c)

**VALUES** (1, 20, 68)

**ON DUPLICATE KEY UPDATE** a=a+1

**A noter :** cette requête se traduit comme suit :

* insérer les données **a, b et c** avec les données respectives de **1, 20 et 68**
* Si la clé primaire existe déjà pour ces valeurs alors seulement faire une insertion avec a = a+1

**Exemple avec la clause WHERE**

**Avec DB2 datemaj = CURRENT DATE**

Grâce à la commande « ON DUPLICATE KEY » Il est possible d’enregistrer la date à laquelle la données est insérée pour la première fois et la date de dernière mise à jour, comme le montre la commande ci-dessous:

**INSERT INTO** table (a, b, c, date\_insert)

**VALUES** (1, 20, 1, NOW())

**ON DUPLICATE KEY**

**UPDATE** date\_update=NOW

**WHERE** c=1

**A noter :** cette requête se traduit comme suit :

1. insérer les données a, b, c et date\_insert, avec les données respectives de 1, 20, 1 ainsi que la date et l’heure actuelle
2. Si la clé primaire existe déjà pour ces valeurs alors mettre a jour la date et l’heure du colonne « date\_update »
3. Effectuer la mise à jour uniquement sur les colonnes où c = 1

Pour n’utiliser qu’une seule ligne qui permet d’ajouter des votes dans cette table, sans se préoccuper de savoir s’il faut faire un INSERT ou un UPDATE, il est possible d’utiliser la requête SQL suivante:

**INSERT INTO** vote

(produit\_id, vote\_count, vote\_first\_date

, vote\_last\_date)

**VALUES** (50, 1, NOW(), NOW())

**ON DUPLICATE KEY**

**UPDATE** vote\_count = vote\_count+1, vote\_last\_date = NOW()

Dans cette requête la date et l’heure est générée automatiquement avec la fonction NOW().

Résultat après la première exécution de la requête:

| id | produit\_id | vote\_count | vote\_first\_date | vote\_last\_date |
| --- | --- | --- | --- | --- |
| 1 | 46 | 2 | 2012-04-25 17:45:24 | 2013-02-16 09:47:02 |
| 2 | 39 | 4 | 2012-04-28 16:54:44 | 2013-02-14 21:04:35 |
| 3 | 49 | 1 | 2012-04-25 19:11:09 | 2013-01-06 20:32:57 |
| 4 | 55 | 1 | 2013-04-02 15:06:34 | 2013-04-02 15:06:34 |

Ce résultat montre bien l’ajout d’une ligne en fin de table, donc la requête a été utilisé sous la forme d’un INSERT.

Après une deuxième exécution de cette même requête le lendemain, les données seront celles-ci:

| id | produit\_id | vote\_count | vote\_first\_date | vote\_last\_date |
| --- | --- | --- | --- | --- |
| 1 | 46 | 2 | 2012-04-25 17:45:24 | 2013-02-16 09:47:02 |
| 2 | 39 | 4 | 2012-04-28 16:54:44 | 2013-02-14 21:04:35 |
| 3 | 49 | 1 | 2012-04-25 19:11:09 | 2013-01-06 20:32:57 |
| 4 | 55 | 2 | 2013-04-02 15:06:34 | 2013-04-03 08:14:57 |

Ces résultats montre bien qu’il y a eu un vote supplémentaire et que la date du dernier vote a été mis à jour.

### Insérer une ligne ou ne rien faire

Dans certains cas il est intéressant d’utiliser un INSERT mais de ne rien faire si la commande a déjà été insérée précédemment. Malheureusement, si la clé primaire existe déjà la requête retournera une erreur. Et s’il n’y a rien à mettre à jour, la commande ON DUPLICATE KEY UPDATE (ODKU) ne semble pas convenir. Toutefois il y a une astuce qui consiste à utiliser une requête de ce type:

**INSERT INTO** table (a, b, c)

**VALUES** (1, 45, 6)

**ON DUPLICATE KEY**

**UPDATE** id = id

Cette requête insert les données et ne produit aucune erreur si l’enregistrement existait déjà dans la table.

**A savoir :** théoriquement il aurait été possible d’utiliser INSERT IGNORE mais malheureusement cela empêche de retourner des erreurs telles que des erreurs de conversions de données.

### Compatibilité

Pour le moment cette fonctionnalité n’est possible qu’avec MySQL depuis la version 4.1 (date de 2003).

## ****Commande 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** table

**SET** nom\_colonne\_1 **=** 'nouvelle valeur'

**WHERE** **condition**

Cette syntaxe permet d’attribuer une nouvelle valeur à la colonne nom\_colonne\_1 pour les lignes qui respectent la condition stipulé avec WHERE. Il est aussi possible d’attribuer la même valeur à la colonne nom\_colonne\_1 pour toutes les lignes d’une table si la condition WHERE n’était pas utilisée.

A noter, pour spécifier en une seule fois plusieurs modification, il faut séparer les attributions de valeur par des virgules. Ainsi la syntaxe deviendrait la suivante :

**UPDATE** table

**SET** colonne\_1 = 'valeur 1'

, colonne\_2 = 'valeur 2'

, colonne\_3 = 'valeur 3'

**WHERE** **condition;**

**UPDATE** client

**SET** rue = '49 Rue Ameline'

, ville = 'Saint-Eustache'

, code\_postal = '76210'

**WHERE** id = 2 ;

Cette requête sert à définir la colonne rue à « 49 Rue Ameline », la ville à « Saint-Eustache » et le code postal à « 76210 » uniquement pour ligne où l’identifiant est égal à 2.

### Modifier toutes les lignes

Il est possible d’effectuer une modification sur toutes les lignes en omettant d’utiliser une clause conditionnelle. Il est par exemple possible de mettre la valeur « FRANCE » dans la colonne « pays » pour toutes les lignes de la table, grâce à la requête SQL ci-dessous.

**UPDATE** client

**SET** pays **=** 'FRANCE' ;

**Si la clause WHERE est absente, toutes les lignes de la table sont impactées.**

1. ****UPDATE** API16.ART**
2. ****SET** ARTLIB = 'NOUVEAU LIBELLE'**
3. **, ARTCRE = CURRENT TIMESTAMP**
4. ****WHERE** ARTIDE = 8 ;**

## ****Commandes DELETE****

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

Avant d’essayer de supprimer des lignes, il est recommandé d’effectuer une **sauvegarde de la base de données**, ou tout du moins de la table concernée par la suppression. Ainsi, s’il y a une mauvaise manipulation il est toujours possible de restaurer les données.

**A noter**  : voir aussi la commande TRUNCATE

**Syntaxe**

La syntaxe pour supprimer des lignes est la suivante :

**DELETE** **FROM** `table`

**WHERE** condition

### Supprimer toutes les données

Pour supprimer toutes les lignes d’une table il convient d’utiliser la commande DELETE sans utiliser de clause conditionnelle.

**DELETE FROM** `utilisateur` ;

1. **DELETE** **FROM** API16.MAR
2. **WHERE** MARCOD > 'AO' ;

Conseil : Lors des tests, avant de supprimer une ligne, afficher ou sauvegarder le contenu de la table. Si la table a des tables dépendantes, afficher ou sauvegarder le contenu de ces tables.

Exemple : Table FAM mère avec DELETE RESTRICT sur la table ART

1. SELECT \*
2. FROM API16.FAM F
3. WHERE FAMCOD ='FA'
4. ORDER BY 1 , 2 ;

Réponse :

1. FAMCOD FAMLIB
2. FA FANTAISIE
3. *NUMBER OF ROWS DISPLAYED IS 1*

Table fille avec DELETE RESTRICT

1. SELECT ARTIDE, ARTFAM
2. FROM API16.ART M
3. WHERE ARTFAM ='FA'
4. ORDER BY ARTFAM, ARTIDE ;

Réponse :

1. ARTIDE ARTFAM
2. 1 FA
3. 2 FA
4. *NUMBER OF ROWS DISPLAYED IS 2*
5. DELETE
6. FROM API16.ART M
7. WHERE ARTFAM ='FA' ;

Réponse :

*---------+---------+---------+---------+---------*

*DSNT408I SQLCODE =* ***-532****, ERROR: THE RELATIONSHIP FKDEC RESTRICTS THE DELETION OF ROW*

*La tentative de suppression de l'occurrence de la table mère a été un échec à cause de la relation "DELETE RESTRICT".*

======================================

Exemple : Table mère avec DELETE CASCADE

SELECT MARCOD , MARLIB

FROM API16.MAR M

WHERE MARCOD = 'AG'

ORDER BY 1 ;

Réponse :

MARCOD MARLIB

AG MARQUE AG

*NUMBER OF ROWS DISPLAYED IS 1*

Autre requête

SELECT ARTIDE, ARTMAR

FROM API16.ART A

WHERE ARTMAR = 'AG'

ORDER BY 1, 2 ;

Réponse :

ARTIDE ARTMAR

11 AG

*NUMBER OF ROWS DISPLAYED IS 1*

Autre requête

DELETE FROM API16.MAR

WHERE MARCOD = 'AG' ;

Réponse :

DSNE615I NUMBER OF ROWS AFFECTED IS 1

Vérification :

SELECT MARCOD , MARLIB

FROM API16.MAR M

WHERE MARCOD = 'AG'

ORDER BY 1 ;

Réponse :

*MARCOD MARLIB*

*DSNE610I NUMBER OF ROWS DISPLAYED IS 0*

Vérification :

SELECT ARTIDE, ARTMAR

FROM API16.ART A

WHERE ARTMAR = 'AG'

ORDER BY 1, 2 ;

Réponse :

ARTIDE ARTMAR

*NUMBER OF ROWS DISPLAYED IS 0*

Explication : la suppression de la ligne de la table mère avec un DELETE CASCADE a entrainé la suppression des lignes dans la table fille.

## ****Commande MERGE****

Dans le langage SQL, la commande MERGE insert ou met à jour des données dans une table. Cette commande permet d’éviter d’effectuer plusieurs requêtes pour savoir si une donnée est déjà dans la base de données et ainsi adapter l’utilisation d’une requête pour ajouter ou une autre pour modifier la donnée existante. Cette commande peut aussi s’appeler « **upsert** ».

**Attention :** bien que l’instruction a été ajoutée dans le standard SQL:2003, les différentes SGBD n’utilisent pas toutes les mêmes méthodes pour effectuer un upsert.

**Syntaxe**

La syntaxe standard pour effectuer un merge consiste à utiliser une requête SQL semblable à celle ci-dessous :

**MERGE INTO** table\_1

**USING** table\_reference **ON** (conditions)

**WHEN MATCHED THEN**

**UPDATE SET** table\_1.colonne\_1 = valeur\_1

, table\_1.colonne\_2 = valeur\_2

**DELETE WHERE** conditions2

**WHEN NOT MATCHED THEN**

**INSERT** (colonnes1, colonne\_3)

**VALUES** (valeur\_1, valeur\_3)

Voici les explications détaillées de cette requête :

1. MERGE INTO permet de sélectionner la table à modifier
2. USING et ON permet de lister les données sources et la condition de correspondance
3. WHEN MATCHED permet de définir la condition de mise à jour lorsque la condition est vérifiée
4. WHEN NOT MATCHED permet de définir la condition d’insertion lorsque la condition n’est pas vérifiée

### Compatibilité

Les systèmes de gestion de bases de données peuvent implémenter cette fonctionnalité soit de façon standard, en utilisant une commande synonyme ou en utilisant une syntaxe non standard.

* **Syntaxe standard :**
  + SQL Server, Oracle, DB2, Teradata et EXASOL
* **Utilisation du terme UPSERT :**
  + Microsoft SQL Azure et MongoDB
* **Utilisation non standard :**
  + MySQL, SQLite, Firebird, IBM DB2 et Microsoft SQL

## ****Commande TRUNCATE TABLE****

En SQL, la commande TRUNCATE permet de supprimer toutes les données d’une table sans supprimer la table en elle-même. En d’autres mots, cela permet de purger la table. Cette instruction diffère de la commande DROP qui a pour but de supprimer les données ainsi que la table qui les contient.

**A noter :** l’instruction TRUNCATE est semblable à l’instruction DELETE sans utilisation de WHERE. Parmi les petite différences TRUNCATE est toutefois plus rapide et utilise moins de ressource. Ces gains en performance se justifie notamment parce que la requête n’indiquera pas le nombre d’enregistrement supprimés et qu’il n’y aura pas d’enregistrement des modifications dans le journal.

**Syntaxe**

Cette instruction s’utilise dans une requête SQL semblable à celle-ci :

**TRUNCATE TABLE** `table\_a\_purger`

Dans cet exemple, les données de la table « table\_a\_purger » seront perdues une fois cette requête exécutée.

Une fois la requête exécutée, la table ne contiendra plus aucun enregistrement. En d’autres mots, toutes les lignes du table présenté ci-dessus auront été supprimées.

### Différence entre TRUNCATE et DELETE

La commande TRUNCATE s’avère être similaire à la commande DELETE, lorsqu’elle est utilisée de la façon suivante :

**DELETE FROM** `fourniture`

Il y a toutefois une différence notable : la commande TRUNCATE va **ré-initialiser la valeur de l’auto-incrément**, s’il y en a un.

Dès lors, il faut potentiellement noter la valeur de l’auto-incrément avant de faire un TRUNCATE, surtout s’il faut ré-indiquer l’ancienne valeur après avoir écrasé toutes les données.

## Commande COMMIT

La commande COMMIT confirme les modifications effectuées depuis le précédent COMMIT, ou à défaut depuis le début des requêtes SQL. La base de données est alors réellement mise à jour.

Si la commande n’est pas explicite, elle est réalisée à la fin du programme.

La commande inverse est ROLLBACK.

**COMMIT** ;

## Commande ROLLBACK

La commande ROLLBACK, en un seul mot, annule les modifications demandées depuis le dernier COMMIT, ou à défaut depuis le début des requêtes SQL. La base de données n’est alors pas mise à jour.

Attention. Le temps d’exécution d’un ROLLBACK peut être beaucoup plus long que le temps d’exécution des mises à jour, c’est pourquoi il est conseillé de rendre fréquents les COMMIT.

Explication : les mises à jour sont enregistrées dans un journal pour permettre de revenir en arrière. Le COMMIT supprime le contenu du journal.

La commande inverse est COMMIT.

**ROLLBACK** ;

## Commande SELECT

La commande SELECT retourne le contenu d'une ou de plusieurs colonnes d’une ou de plusieurs tables.

### Commande pour une table

La commande SELECT permet de retourner une ou plusieurs colonnes dans l'ordre voulu qui peut être différent de l'ordre de création de la table.

Sélection d'une colonne d'une seule table

**SELECT** nom\_de\_colonne **FROM** nom\_de\_la\_table

Sélection de plusieurs colonnes d'une seule table

Cette requête SQL va **sélectionner** (SELECT) la colonne «nom\_de\_colonne » **provenant** (FROM) de la table appelé «nom\_de\_la\_table ».

**SELECT** nom\_de\_colonne\_1, nom\_de\_colonne\_2, ... , nom\_de\_colonneN

**FROM** nom\_de\_la\_table

Avantages et inconvénients du sélecteur étoile

**SELECT** \* **FROM** nom\_de\_la\_table

Dans une requête SQL, le caractère étoile permet de lire tous les colonnes d’une base de données. Très pratique pour connaître facilement et rapidement toutes les colonnes d’une table sans avoir à lister soi-même le nom des colonnes désirés, il est déconseillé dans les applications.

Le caractère étoile peut avoir de mauvaises conséquences sur les performances d’une application. Pour optimiser les performances, il convient de lire seulement les données utilisées par l’application.

Il existe un risque d'erreur si la base de données est modifiée (modification d’une colonne, modification de l’ordre des colonnes …).

Le caractère \* doit par conséquent être réservé à un usage ponctuel lorsqu’un développeur souhaite lire rapidement le contenu d’une base.

SELECT \* FROM API16.MAR ;

*réponse :*

*---------+---------+---------+---------+---------+----*

*MARCOD MARLIB MARCRE MARMAJ*

*---------+---------+---------+---------+---------+----*

*AH MARQUE AH 2024-06-17 ----------*

*AI MARQUE AI 2024-06-17 ----------*

*AJ MARQUE AJ 2024-06-17 ----------*

*AK MARQUE AK 2024-06-17 ----------*

*AL MARQUE AL 2024-06-17 ----------*

*AM MARQUE AM 2024-06-17 ----------*

*AN MARQUE AN 2024-06-17 ----------*

*AO MARQUE AO 2024-06-17 ----------*

*DSNE610I NUMBER OF ROWS DISPLAYED IS 8*

Remarque : quand une colonne est NULL, son contenu est affiché par une suite de tirets "-".

# Les clauses

## Clause WHERE

La clause WHERE dans une requête SQL permet de restreindre les lignes d’une base de données à celles qui respectent une condition.

1. **SELECT** nom\_colonne (ou \*, SUM...)
2. **FROM** nom\_table
3. **WHERE** condition
4. **SELECT** \* **FROM** client
5. **WHERE** ville = 'Pau'

### 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  NOT (condition) |
| > | Supérieur à |
| < | Inférieur à |
| >= | Supérieur ou égale à |
| <= | Inférieur ou égale à |
| IN | Appartient à une liste de plusieurs valeurs possibles |
| BETWEEN | Est compris dans un intervalle donné, bornes incluses (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 |

**Attention**: il y a quelques opérateurs qui n’existe pas dans des vieilles versions de système de gestion de bases de données (SGBD). De plus, il y a de nouveaux opérateurs non indiqués ici qui sont disponibles avec certains SGBD. N’hésitez pas à consulter la documentation de MySQL, PostgreSQL ou autre pour voir ce qu’il vous est possible de faire.

### Opérateurs AND et OR

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

Syntaxe d’utilisation des opérateurs AND et OR

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

L’opérateur AND permet de s’assurer que la condition\_1 ET la condition\_2 sont vraies :

**SELECT** nom\_colonnes

**FROM** nom\_table

**WHERE** condition\_1 **AND** condition\_2

L’opérateur OR vérifie que la condition\_1 OU la condition\_2 est vraie :

**SELECT** nom\_colonnes

**FROM** nom\_table

**WHERE** condition\_1 **OR** condition\_2

Ces opérateurs peuvent être combinés à l’infini et mélangés. L’exemple ci-dessous filtre les résultats de la table « nom\_table » si condition\_1 ET condition\_2 OU condition\_3 est vrai :

**SELECT** nom\_colonnes

**FROM** nom\_table

**WHERE** condition\_1 **AND** (condition\_2 **OR** condition\_3)

Il vaut mieux utiliser des parenthèses qui permettent d’éviter les erreurs et améliorent la lecture par un humain.

### Opérateur IN

L’opérateur logique IN  s’utilise avec la clause WHERE pour vérifier si une colonne est égale à une des valeurs comprises dans un ensemble de valeurs déterminé. C’est une méthode simple pour vérifier si une colonne est égale à une valeur OU à une autre valeur OU à une autre valeur et ainsi de suite, sans avoir à utiliser de multiple fois l’likeopérateur OR.

**Syntaxe**

**SELECT** nom\_colonne **FROM** table

**WHERE** nom\_colonne

**IN** ( valeur\_1, valeur\_2, valeur\_3,...)

Cette syntaxe peut être associée à l’opérateur NOT pour recherche toutes les lignes différentes des valeurs stipulées.

### Opérateur BETWEEN

L’opérateur BETWEEN sélectionne un intervalle de données dans une requête utilisant WHERE. L’intervalle peut être constitué de chaînes de caractères, de nombres ou de dates. L’exemple le plus concret consiste par exemple à récupérer uniquement les enregistrements entre 2 dates définies.

Syntaxe

**SELECT** \*

**FROM** table

**WHERE** nom\_colonne

**BETWEEN** 'valeur\_1' **AND** 'valeur\_2'

### Opérateur LIKE

L’opérateur LIKE est utilisé dans la clause WHERE pour 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.

**SELECT** \* **FROM** table

**WHERE** colonne **LIKE** modele

* 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 enregistrement 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 « paon », « 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 ou plusieurs caractères . Ainsi, ce modèle permet de retourner les lignes « aac », « abc » ou même « azc » mais pas « ac ».

### Opérateur IS NULL ou IS NOT NULL

L’opérateur IS permet de filtrer les résultats qui contiennent ou non la valeur NULL.

**SELECT** \*

**FROM** `table`

**WHERE** nom\_colonne **IS NULL**

**SELECT** \*

**FROM** `table`

**WHERE** nom\_colonne **IS NOT NULL**

L’opérateur IS retourne en réalité un booléen, c’est à dire une valeur TRUE si la condition est vrai 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.

**SELECT** \* **FROM** API16.MAR

**WHERE** MARCOD **NOT IN**

(SELECT ARTMAR

FROM API16.ART

)

;

Explication : sélectionner les occurrences de la table marque MAR dont les codes de marque (MARCOD est l'index primaire de la table MAR) sont absents de la table article ART (la colonne ARTMAR de la table ART est la clé étrangère sur MARCOD)

Réponse :

MARCOD MARLIB MARCRE MARMAJ

*AH MARQUE AH 2024-06-17 ----------*

*AI MARQUE AI 2024-06-17 ----------*

*AJ MARQUE AJ 2024-06-17 ----------*

*AK MARQUE AK 2024-06-17 ----------*

*AL MARQUE AL 2024-06-17 ----------*

*AM MARQUE AM 2024-06-17 ----------*

*AN MARQUE AN 2024-06-17 ----------*

*AO MARQUE AO 2024-06-17 ----------*

*DSNE610I NUMBER OF ROWS DISPLAYED IS 8*

# Mot-clé, clause

### Mot-clé DISTINCT ou UNIQUE

Pour éviter de retourner les doublons, il faut ajouter DISTINCT après le mot SELECT.

L’utilisation basique de cette commande consiste alors à effectuer la requête suivante:

**SELECT** **DISTINCT** ma\_colonne

**FROM** nom\_de\_la\_table

Requête pour Oracle

Pour le Système de Gestion de Bases de Données (SGBD) Oracle, cette requête est remplacée par la commande « UNIQUE »:

**SELECT** **UNIQUE** ma\_colonne

**FROM** nom\_de\_la\_table

L’utilisation de la commande DISTINCT est très pratique pour éviter les résultats en double. Cependant, pour optimiser les performances il est préférable d’utiliser la clause GROUP BY lorsque c’est possible.

### Clause GROUP BY

La clause GROUP BY groupe plusieurs résultats et permet d'utiliser une fonction de totaux sur un groupe de résultat. Sur une table qui contient toutes les ventes d’un magasin, il est par exemple possible de lister en regroupant les ventes par clients et d’obtenir le coût total des achats pour chaque client.

**SELECT** colonne\_1, **fonction**(colonne\_2)

**FROM** table

**GROUP BY** colonne\_1

**A noter :** cette commande doit toujours s’utiliser après l'éventuelle clause WHERE et avant l'éventuelle commande HAVING.

**A noter :** SELECT et GROUP BY doivent être suivies du même nombre de colonnes et des mêmes colonne :

**SELECT** colonne\_1, colonne\_2 **fonction**(colonne\_3)

**FROM** table

**GROUP BY** colonne\_1, colonne\_2

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

**SELECT** client, **SUM**(montant\_achat)

**FROM** achat

**GROUP BY** client

La manière simple de comprendre le GROUP BY c’est tout simplement de penser 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.

... ????$$

### Fonctions de statistiques

Il existe plusieurs fonctions qui peuvent être utilisées pour manipuler plusieurs lignes. Les principales fonctions d’agrégations statistiques sont les suivantes. Il faut mettre entre () soit une étoile (\*) soit un nom de colonne :

* **AVG()** pour calculer la moyenne d’un ensemble de valeurs. Permet de connaître le prix du panier moyen pour 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 premier achat 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

Ces fonctions se révèlent rapidement indispensable pour travailler sur des données.

### Condition HAVING

La condition HAVING est presque similaire à WHERE à la différence que HAVING est utlisée avec des fonctions telles que SUM(), COUNT(), AVG(), MIN() ou MAX().

**SELECT** colonne\_1, **SUM**(colonne\_2)

**FROM** nom\_table

**GROUP BY** colonne\_1

**HAVING** fonction(colonne\_2) **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 « colonne\_1″ et que la condition de HAVING soit respectée.

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

Exemple : récupérer la liste des clients qui ont commandé pour plus de 4000€, toute commandes confondues :

**SELECT** client, **SUM**(montant\_achat)

**FROM** achat

**GROUP BY** client

**HAVING SUM**(montant\_achat) > 4000

### Commande 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.

**SELECT** colonne\_1, colonne\_2, colonne\_3

**FROM** table

**ORDER BY** colonne\_1

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.

**SELECT** colonne\_1, colonne\_2, colonne\_3

**FROM** table

**ORDER BY** colonne\_1 **DESC**, colonne\_2 **ASC**

### **Mot-clé AS (alias)**

On peut utiliser des **alias** pour renommer temporairement une colonne ou une table dans une requête. Cette astuce est particulièrement utile pour faciliter la lecture des requêtes.

**Alias sur une colonne**

C’est pratique pour avoir un nom facilement identifiable dans une application qui doit ensuite exploiter les résultats d’une recherche.

**SELECT** colonne\_1 **AS** c1, colonne\_2

**FROM `**table**`**

Cas concrets d’utilisations :

* Une colonne qui s’appelle normalement **code**\_3166**** peut être renommée "code\_pays", ce qui est plus compréhensible pour un développeur.
* Une requête qui utilise la commande UNION sur des colonnes aux noms différents peut être ambigue pour un développeur. En renommant les colonnes avec un même nom il est plus simple de traiter les résultats.
* Lorsqu’une fonction est utilisé, le nom d’une colonne peut-être un peu complexe. Il est ainsi possible de renommer la colonne sur laquelle il y a une fonction.

Exemple :

1. SELECT COUNT(\*) AS nombre\_de\_resultats
2. FROM `table\_1`.

* Lorsque plusieurs colonnes sont combinées il est plus simple de renommer la nouvelle colonne qui est une concaténation de plusieurs colonnes.

**Alias sur une table**

Il permet d’attribuer un autre nom à une table dans une requête SQL. Cela peut aider à avoir des noms plus court, plus simple et plus facilement compréhensible. Ceci est particulièrement vrai lorsqu’il y a des jointures.

**SELECT** \* **FROM** `nom\_table` **AS** t1

Le AS étant facultatif, on trouve souvent :

**SELECT** \* **FROM** `nom\_table` t1

L'alias est visiblement utile quand la requête porte sur plusieurs tables. Exemple :

**SELECT** CL.ID\_CLIENT, ID\_COMMANDE

**FROM** CLIENT CL, COMMANDE CO

**WHERE** CL.ID\_CLIENT = CO.ID\_CLIENT ;

### **Clauses LIMIT, OFFSET, TOP, FETCH NEXT, ROWNUM, FETCH FIRST n ROWS ONLY**

Ne fonctionne pas avec DB2.

**SELECT** colonnes

**FROM** table

**LIMIT** nombre\_de\_lignes;

* colonnes : Les colonnes que vous souhaitez sélectionner.
* table : La table à partir de laquelle vous souhaitez récupérer les données.
* nombre\_de\_lignes : Le nombre maximum de lignes que vous souhaitez retourner.

On souhaite récupérer seulement les 5 premiers employés :

**SELECT** \*

**FROM** employes

**LIMIT** 5;

### Utilisation avec OFFSET

La clause LIMIT est souvent utilisée avec OFFSET pour paginer les résultats. OFFSET spécifie le nombre de lignes à ignorer avant de commencer à retourner des lignes.

**SELECT** colonnes

**FROM** table

**LIMIT** nombre\_de\_lignes **OFFSET** décalage;

Décalage: Le nombre de lignes à ignorer avant de commencer à retourner des lignes.

Pour récupérer les 5 employés suivants après les 5 premiers, on peut utiliser :

**SELECT** \*

**FROM** employes

**LIMIT** 5 **OFFSET** 5;

Cela ignorera les 5 premiers employés et retournera les 5 suivants.

Remarques :

* La clause LIMIT est prise en charge par la plupart des SGBD populaires comme MySQL, PostgreSQL, et SQLite.
* Dans certains SGBD comme SQL Server et Oracle, la syntaxe peut être différente. Par exemple, SQL Server utilise TOP ou FETCH NEXT, et Oracle utilise ROWNUM ou FETCH FIRST n ROWS ONLY.

La clause LIMIT est un outil puissant pour optimiser les performances des requêtes et gérer les grands ensembles de données de manière efficace.

### **Expression CASE à compléter depuis le livre**

Ne fonctionne pas sur DB2.

En SQL, le mot-clé CASE est utilisé pour fournir une logique conditionnelle dans les requêtes. Il fonctionne de manière similaire à la structure if-then-else dans d'autres langages de programmation. La clause CASE permet de retourner différentes valeurs en fonction de conditions spécifiques.

### Syntaxe de CASE

Il existe deux formes principales de la syntaxe CASE :

CASE **simple** : Compare une expression à un ensemble de valeurs

SELECT  
 CASE expression  
 WHEN value1 THEN result1  
 WHEN value2 THEN result2  
 ...  
 ELSE default\_result  
 END AS column\_name  
FROM table\_name;

CASE **recherché** : Évalue une série de conditions booléennes.

Exemple de CASE simple

Supposons que vous avez une table produits avec une colonne categorie\_id, et vous voulez afficher le nom de la catégorie au lieu de l'ID :

SELECT  
 nom\_produit,  
 CASE categorie\_id  
 WHEN 1 THEN 'Électronique'  
 WHEN 2 THEN 'Vêtements'  
 WHEN 3 THEN 'Alimentation'  
 ELSE 'Autre'  
 END AS categorie  
FROM produits;

Exemple de CASE recherché

Supposons que vous avez une table etudiants avec une colonne note, et vous voulez classer les étudiants en fonction de leur note :

SELECT nom,  
 CASE  
 WHEN note >= 90 THEN 'Excellent'  
 WHEN note >= 80 THEN 'Très bien'  
 WHEN note >= 70 THEN 'Bien'  
 WHEN note >= 60 THEN 'Assez bien'  
 ELSE 'Insuffisant'  
 END AS performance  
FROM etudiants;

Utilisations courantes :

* **Transformation de données** : Convertir des valeurs codées en valeurs lisibles.
* **Création de nouvelles colonnes** : Ajouter des colonnes calculées basées sur des conditions.
* **Filtrage conditionnel** : Utiliser CASE dans une clause WHERE pour des conditions complexes.

La clause CASE est un outil puissant pour ajouter de la logique conditionnelle à vos requêtes SQL, rendant vos résultats plus informatifs et personnalisés.

## ****Jointure 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.

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 jointure d’inégalité. Ces cas étant assez particulier et pas si simple à comprendre, ils ne seront pas élaboré sur cette page.

### Types de jointures

* + 1. Il y a plusieurs méthodes pour associer 2 ou plusieurs tables ensemble. Voici la liste des différentes techniques qui sont utilisées.

### Commande UNION

La commande UNION de SQL concatène les enregistrements de plusieurs requêtes SELECT en incluant les doublons.

Il faut que le nombre de colonnes soit identique dans chaque SELECT, avec le même type de données et le même ordre.

Les lignes résultantes identiques ne sont pas répétées.

Exemple :

Table commandes :

DUPOND 31/07/2025

DURAND 30/06/2024

DUBOIS 30/06/2025

Table clients :

DUPOND EUGENE

DURAND PATRICK

DUSSON FRANCOIS

**SELECT** nom\_client **FROM** commandes

UNION

**SELECT** nom\_client **FROM** clients ;

Résultat :

nom\_client

DUPOND

DURAND

DUBOIS

DUSSON

### Commande UNION ALL

La commande UNION ALL de SQL est très similaire à la commande UNION. Elle inclue les doublons. Ainsi, si un même résultat est présent dans les résultats des 2 requêtes concaténées, alors l’union des 2 avec UNION ALL retournera 2 fois ce même résultat.

**A savoir :** tout comme la commande UNION, il convient que les 2 requêtes retournent exactement le même nombre de colonnes, avec les mêmes types de données et dans le même ordre.

La syntaxe de la requête SQL pour unir les résultats des 2 tables est la suivante:

**SELECT** \* **FROM** table\_1  
**UNION ALL**  
**SELECT** \* **FROM** table\_2

Exemple :

**SELECT** nom\_client **FROM** commandes

**UNION** **ALL**

**SELECT** nom\_client **FROM** clients ;

Résultat :

nom\_client

DUBOIS

DUPOND

DUPOND

DURAND

DURAND

DUSSON

## SQL Sous-requête

Une sous-requête, ou « requête imbriquée », ou « requête en cascade », est 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 constante.

**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 une exemple typique d’une sous-requête qui retourne un seul résultat à la requête principale.

**SELECT** \* **FROM** `table\_1`

**WHERE** `nom\_colonne` =

(

**SELECT** `valeur`

**FROM** `table\_2`

**LIMIT** 1

)

Cet exemple montre une requête interne (celle sur "table\_2") qui renvoie une seule valeur. La requête externe quant à elle, va chercher les résultat de « table\_1 » 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 <>.

**Exemple**

Avec une telle application, il est peut-être utile de connaître la question liée à la dernière réponse ajoutée sur l’application. Cela peut être effectué via la requête SQL suivante:

**SELECT** \* **FROM** `question`

**WHERE**q\_id **=**

(

**SELECT** r\_fk\_question\_id

**FROM** `reponse`

**ORDER BY** r\_date\_ajout **DESC**

**LIMIT** 1

)

Une telle requête va retourner la ligne suivante:

| q\_id | q\_date\_ajout | q\_titre | q\_contenu |
| --- | --- | --- | --- |
| 3 | 2013-04-18 20:09:56 | Que faire si un appareil est cassé? | Est-il préférable de réparer les appareils électriques ou d'en acheter de nouveaux? |

Ce résultat démontre que la question liée à la dernière réponse sur le forum est bien trouvée à partir de ce résultat.

### Requête imbriquée qui retourne plusieurs valeurs

Imaginons maintenant que l’ont souhaite obtenir les questions liées à toutes les réponses comprises entre 2 dates. Ces questions peuvent être récupérée par la requête SQL suivante:

**SELECT** \* **FROM** `question`

**WHERE** q\_id

IN (

**SELECT** r\_fk\_question\_id

**FROM** `reponse`

**WHERE** r\_date\_ajout

**BETWEEN** '2013-01-01'

**AND** '2013-12-31'

)

Dans notre cas, cette requête retournera les résultats suivants :

| q\_id | q\_date\_ajout | q\_titre | q\_contenu |
| --- | --- | --- | --- |
| 1 | 2013-03-24 12:54:32 | Comment réparer un ordinateur? | Bonjour, j'ai mon ordinateur de cassé, comment puis-je procéder pour le réparer? |
| 2 | 2013-03-26 19:27:41 | Comment changer un pneu? | Quel est la meilleur méthode pour changer un pneu facilement ? |
| 3 | 2013-04-18 20:09:56 | Que faire si un appareil est cassé? | Est-il préférable de réparer les appareils électriques ou d'en acheter de nouveaux? |

Une telle requête permet donc de récupérer les questions qui ont eu des réponses entre 2 dates. C’est pratique dans notre cas pour éviter d’obtenir des réponses qui n’ont pas eu de réponses du tout ou pas de nouvelles réponses depuis longtemps.

## SQL EXISTS

Dans le langage SQL, la commande EXISTS s’utilise dans une clause conditionnelle pour savoir s’il y a une présence ou non de lignes lors de l’utilisation d’une sous-requête.

**A noter :** cette commande ne doit pas être confondue avec la clause IN. La commande EXISTS vérifie si la sous-requête retourne un résultat ou non, tandis que IN vérifie la concordance d’une à plusieurs valeurs.

**Syntaxe**

L’utilisation basique de la commande EXISTS consiste à vérifier si une sous-requête retourne un résultat ou non, en utilisant EXISTS dans la clause conditionnelle. La requête externe s’exécutera uniquement si la requête interne retourne au moins un résultat.

**SELECT** nom\_colonne\_1

**FROM** `table\_1`

**WHERE** **EXISTS** **(**

**SELECT** nom\_colonne\_2

**FROM** `table\_2`

**WHERE** nom\_colonne\_3 = 10

**)**

Dans l’exemple ci-dessus, s’il y a au moins une ligne dans **table\_2** dont **nom\_colonne\_3** contient la valeur 10, alors la sous-requête retournera au moins un résultat. Dès lors, la condition sera vérifiée et la requête principale retournera les résultats de la colonne **nom\_colonne\_1** de **table\_1**.

**Exemple**

Soit un système composé d’une table qui contient des commandes et d’une table contenant des produits.

**Table commande :**

| c\_id | c\_date\_achat | c\_produit\_id | c\_quantite\_produit |
| --- | --- | --- | --- |
| 1 | 2014-01-08 | 2 | 1 |
| 2 | 2014-01-24 | 3 | 2 |
| 3 | 2014-02-14 | 8 | 1 |
| 4 | 2014-03-23 | 10 | 1 |

**Table produit :**

| p\_id | p\_nom | p\_date\_ajout | p\_prix |
| --- | --- | --- | --- |
| 2 | Ordinateur | 2013-11-17 | 799.9 |
| 3 | Clavier | 2013-11-27 | 49.9 |
| 4 | Souris | 2013-12-04 | 15 |
| 5 | Ecran | 2013-12-15 | 250 |

Il est possible d’effectuer une requête SQL qui affiche les commandes pour lesquels il y a effectivement un produit.

**SELECT** \* **FROM** commande

**WHERE** **EXISTS** **(**

**SELECT** \*

**FROM** produit

**WHERE c**\_produit\_id = p\_id

**)**

**Résultat :**

| c\_id | c\_date\_achat | c\_produit\_id | c\_quantite\_produit |
| --- | --- | --- | --- |
| 1 | 2014-01-08 | 2 | 1 |
| 2 | 2014-01-24 | 3 | 2 |

Le résultat démontre bien que seul les commandes n°1 et n°2 ont un produit qui se trouve dans la table **produit** (cf. la condition c\_produit\_id = p\_id). Cette requête est intéressante sachant qu’elle n’influence pas le résultat de la requête principale, contrairement à l’utilisation d’une jointure qui va concaténer les colonnes des 2 tables jointes.

## SQL ALL

La commande ALL permet de comparer une valeur dans l’ensemble de valeurs d’une sous-requête. Elle permet de s’assurer qu’une condition est vraie pour **tous** les résultats retournés par une sous-requête.

**Syntaxe**

Cette commande s’utilise dans une clause conditionnelle entre l’opérateur de condition et la sous-requête.

**SELECT** \* **FROM** table\_1

**WHERE** condition >

**ALL** **(**

**SELECT** \*

**FROM** table\_2

**WHERE** condition₂ **)**

**A savoir :** les opérateur conditionnels peuvent être les suivants : **=**, **<**, **>**, **<>**, **!=**, **<=**, **>=**, **!>** ou **!<**.

**Exemple**

Imaginons une requête similaire à la syntaxe de base présentée précédemment :

**SELECT** **colonne\_1** **FROM** table\_1

**WHERE** colonne\_1 >

**ALL (**

**SELECT** colonne\_1

**FROM** table\_2

**)**

Avec cette requête, si dans **table\_1** il y a un résultat avec la valeur **10**, voici les différents résultats de la conditions selon le contenu de **table\_2** :

* La condition est vrai si table\_2 contient {-5,0,+5} car toutes les valeurs sont inférieures à 10
* La condition est fausse si table\_2 contient {12,6,NULL,-100} car au moins une valeur est inférieure à 10
* La condition est non connue si table\_2 est vide

## SQL ANY / SOME

Cette commande s’utilise dans une clause conditionnelle juste après un opérateur conditionnel et juste avant une sous-requête. La commande ANY (ou SOME) compare une valeur avec le résultat d’une sous-requête. Il est ainsi possible de vérifier si une valeur est « égale », « différente », « supérieur », « supérieur ou égale », « inférieur » ou « inférieur ou égale » pour **au moins une des valeurs** de la sous-requête.

**A noter :** le mot-clé SOME est un alias de ANY, l’un et l’autre des termes peut être utilisé.

**Syntaxe**

L’exemple ci-dessous démontre une utilisation basique de ANY dans une requête SQL :

**SELECT** \* **FROM** table\_1

**WHERE** condition >

**ANY (**

**SELECT** \*

**FROM** table\_2

**WHERE** condition\_2

**)**

Cette requête peut se traduire de la façon suivante : sélectionner toutes les colonnes de table\_1, où la condition est supérieure à n’importe quel résultat de la sous-requête.

**A savoir :** les opérateur conditionnels peuvent être les suivants : **=**, **<**, **>**, **<>**, **!=**,**<=**, **>=**, **!>** ou **!<**.

**Exemple**

En se basant sur l’exemple relativement simple présenté ci-dessus, il est possible d’effectuer une requête concrète qui utilise la commande ANY :

**SELECT** colonne\_1 **FROM** table\_1

**WHERE** colonne\_1 >

**ANY (**

**SELECT** colonne\_1

**FROM** table\_2

**)**

Supposons que la **table\_1** possède un seul résultat dans lequel colonne\_1 est égal à 10.

* La condition est vrai si table\_2 contient {21, 14, 7} car il y a au moins une valeur inférieure à 10
* La condition est fausse si table\_2 contient {20, 10} car aucune valeur est strictement inférieure à 10
* La condition est non connue si table\_2 est vide

### Astuce

La commande IN est équivalent à l’opérateur **=** suivi de ANY.

**Exemples**

**Liste de la table MAR quelle que soit la table ART**

**SELECT \* FROM API16.MAR ;**

**---------+---------+---------+---------+---------**

**MARCOD MARLIB MARCRE MARMAJ**

**---------+---------+---------+---------+---------**

**AB MARQUE AB 2024-06-17 ----------**

**AC MARQUE AC 2023-06-27 ----------**

**AD MARQUE AD 2024-05-27 ----------**

**AE MARQUE AE 2025-06-26 ----------**

**AF MARQUE AF 2024-06-17 ----------**

**AG MARQUE AG 2024-06-17 ----------**

**AH MARQUE AH 2024-06-17 ----------**

**AI MARQUE AI 2024-06-17 ----------**

**AJ MARQUE AJ 2024-06-17 ----------**

**AK MARQUE AK 2024-06-17 ----------**

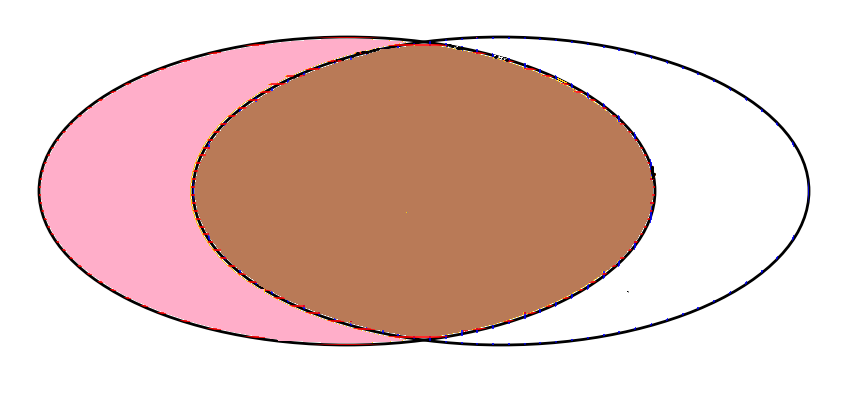
**AL MARQUE AL 2024-06-17 ----------**

**AM MARQUE AM 2024-06-17 ----------**

**AN MARQUE AN 2024-06-17 ----------**

**AO MARQUE AO 2024-06-17 ----------**

**DSNE610I NUMBER OF ROWS DISPLAYED IS 14**



**Liste de la table MAR ayant un code dans la table ART**

****SELECT** \* **FROM** API16.MAR**

****WHERE** MARCOD **IN****

****(SELECT** ARTMAR**

****FROM** API16.ART**

****)** ;**

**---------+---------+---------+---------+---------**

**MARCOD MARLIB MARCRE MARMAJ**

**---------+---------+---------+---------+---------**

**AB MARQUE AB 2024-06-17 ----------**

**AC MARQUE AC 2023-06-27 ----------**

**AD MARQUE AD 2024-05-27 ----------**

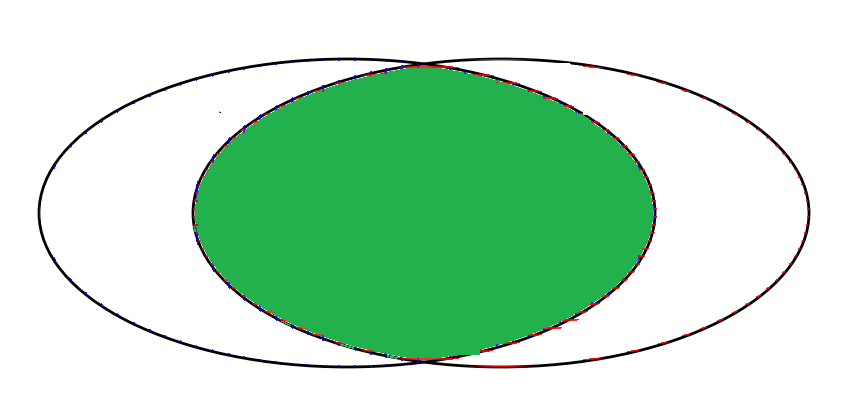
**AE MARQUE AE 2025-06-26 ----------**

**AF MARQUE AF 2024-06-17 ----------**

**AG MARQUE AG 2024-06-17 ----------**

**DSNE610I NUMBER OF ROWS DISPLAYED IS 6**

**DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100**



**Liste de la table MAR n'ayant aucun code dans la table ART**

****SELECT** \* **FROM** API16.MAR**

****WHERE** MARCOD **NOT IN****

****(SELECT** ARTMAR**

****FROM** API16.ART**

****)** ;**

**---------+---------+---------+---------+---------**

**MARCOD MARLIB MARCRE MARMAJ**

**---------+---------+---------+---------+---------**

**AH MARQUE AH 2024-06-17 ----------**

**AI MARQUE AI 2024-06-17 ----------**

**AJ MARQUE AJ 2024-06-17 ----------**

**AK MARQUE AK 2024-06-17 ----------**

**AL MARQUE AL 2024-06-17 ----------**

**MARCOD MARLIB MARCRE MARMAJ**

**---------+---------+---------+---------+---------**

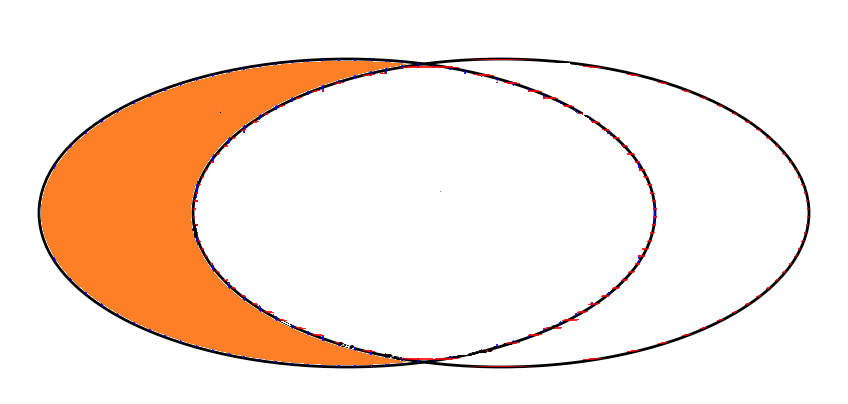
**AM MARQUE AM 2024-06-17 ----------**

**AN MARQUE AN 2024-06-17 ----------**

**AO MARQUE AO 2024-06-17 ----------**

***DSNE610I NUMBER OF ROWS DISPLAYED IS 8***

***DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100***



# SQL EXPLAIN

L’instruction EXPLAIN doit être utilisée juste avant un SELECT et permet d’afficher le plan d’exécution d’une requête SQL. Cela permet de savoir de quelle manière le Système de Gestion de Base de Données (SGBD) va exécuter la requête et s’il va utiliser des index et lesquels.

Un plan d’exécution est une représentation des étapes que SQL effectue pour exécuter une requête.

En utilisant cette commande la requête ne renverra pas les résultats du SELECT mais l'analyse de cette requête.

**A noter :** le résultat de cette instruction est différent selon les SGBD, tel que MySQL ou PostgreSQL. Par ailleurs, le nom de cette instruction diffère pour certains SGBD.

# Commentaires en SQL

Il peut être intéressant d’insérer des commentaires dans les requêtes SQL pour mieux s’y retrouver lorsqu’il y a de grosses requêtes complexes. Il y a plusieurs façon de faire des commentaires dans le langage SQL, qui dépendent notamment du Système de Gestion de Base de Données utilisées (SGBD) et de sa version.

**Commentaire double tiret : —**

Le double tiret permet de faire un commentaire jusqu’à la fin de la ligne.

**Exemple :**

1. **SELECT** \*   **--** tout sélectionner
2. **FROM** table\_1 **--** dans la table "table\_1"
3. **--**
4. **SELECT \* FROM API16.MAR**
5. **;**
6. **-- COMMENTAIRE**

**Commentaire dièse : #**

**Ne fonctionne pas avec DB2.**

**Commentaire multi-ligne : /\* et \*/**

1. **Ne fonctionne pas avec DB2.**

# Procédure stockée

Une procédure stockée ("stored procedure" en anglais) est un ensemble d'instructions SQL précompilées, stockées dans une base de données et exécutées sur demande par le SGBD qui manipule la base de données.

Les procédures stockées peuvent être lancées par un utilisateur, un DBA ou automatiquement par un événement déclencheur ("trigger" en anglais).

Elles sont optimisées par rapport aux requêtes qui sont envoyées au serveur SQL, puis font l'objet d'une "analyse syntaxique", et enfin d'une interprétation avant d'être exécutées.

Pour une procédure stockée, la requête n'est envoyée qu'une fois sur le réseau puis analysée, interprétée et stockée sur le serveur sous forme exécutable. Pour l'exécuter, on envoie une requête comportant le nom de la procédure stockée.

On peut passer des paramètres à une procédure stockée lors de son appel, et on reçoit le résultat comme pour toute requête SQL.

# Fonctions d’agrégation SQL

Les fonctions d’agrégation dans le langage SQL permettent d’effectuer des opérations statistiques sur un ensemble d’enregistrements. Étant donné que ces fonctions s’appliquent à plusieurs lignes en même temps, elle permettent des opérations qui servent à récupérer l’enregistrement le plus petit, le plus grand ou bien encore de déterminer la valeur moyenne sur plusieurs enregistrement.

Les fonctions d’agrégation sont des fonctions idéales pour effectuer quelques statistiques de bases sur des tables. Les principales fonctions sont les suivantes.

## Fonction AVG() calcul

La fonction d’agrégation AVG() dans le langage SQL permet de calculer une valeur moyenne sur un ensemble d’enregistrement de type numérique et non nul. pour calculer la moyenne sur un ensemble d’enregistrements

**Syntaxe**

La syntaxe pour utiliser cette fonction de statistique est simple :

1. **SELECT AVG**(nom\_colonne) **FROM** nom\_table

Cette requête permet de calculer la note moyenne de la colonne « nom\_colonne » sur tous les enregistrements de la table « nom\_table ». Il est possible de filtrer les enregistrements concernés à l’aide de la clause WHERE. Il est aussi possible d’utiliser la commande GROUP BY pour regrouper les données appartenant à la même entité.

**A savoir :** la syntaxe est conforme avec la norme SQL et fonctionne correctement avec tous les Systèmes de Gestion de Base de Données (SGBD), incluant : MySQL, PostgreSQL, Oracle et SQL Server.

**Exemple**

Pour connaitre le montant moyen effectué par chaque client, il est possible d’utiliser une requête qui va utiliser :

* GROUP BY pour regrouper les ventes des mêmes clients
* La fonction AVG() pour calculer la moyenne des enregistrements

La requête sera donc la suivante :

1. **SELECT \***
2. **FROM API16.ART**
3. **ORDER BY ARTMAR, ARTIDE**
4. **;**

**---------+---------+---------+---------+---------**

**ARTIDE ARTLIB ARTFAM ARTMAR**

**---------+---------+---------+---------+---------**

**1 ? FA AB**

**3 NOUVEAU LIBELLE AB AB**

**2 NOUVEAU LIBELLE FA AC**

**4 ? AC AD**

**7 ? AD AD**

**5 ? AC AE**

**8 NOUVEAU LIBELLE AE AE**

**6 ? AC AF**

**9 ? AE AF**

**10 ? AF AF**

**11 ? AF AG**

**DSNE610I NUMBER OF ROWS DISPLAYED IS 11**

1. **SELECT ARTMAR**
2. **FROM API16.ART**
3. **GROUP BY ARTMAR**
4. **;**

**---------+---------+---------+---------+**

**ARTMAR**

**---------+---------+---------+---------+**

**AB**

**AC**

**AD**

**AE**

**AF**

**AG**

***DSNE610I NUMBER OF ROWS DISPLAYED IS 6***

1. **SELECT ARTMAR , COUNT(\*)**
2. **FROM API16.ART**
3. **GROUP BY ARTMAR**
4. **;**

**---------+---------+---------+---------**

**ARTMAR**

**---------+---------+---------+---------**

**AB 2**

**AC 1**

**AD 2**

**AE 2**

**AF 3**

**AG 1**

***DSNE610I NUMBER OF ROWS DISPLAYED IS 6***

1. **SELECT ARTIDE, DECIDE, DECQUANTITE**
2. **FROM API16.DEC**
3. **ORDER BY**
4. **ARTIDE, DECIDE ;**

**---------+---------+---------+---------**

**ARTIDE DECIDE DECQUANTITE**

**---------+---------+---------+---------**

**1 1. 1.**

**1 2. 10.**

**2 1. 100.**

**2 2. 200.**

**2 3. 300.**

**3 1. 333.**

**4 1. 444.**

***DSNE610I NUMBER OF ROWS DISPLAYED IS 7***

1. ****SELECT** ARTIDE**
2. **, **AVG**(DECQUANTITE) **AS** AVG**
3. **, **SUM**(DECQUANTITE) **AS** SUM**
4. **, **MIN**(DECQUANTITE) **AS** MIN**
5. **, **MAX**(DECQUANTITE) **AS** MAX**
6. ****FROM** API16.DEC**
7. ****GROUP** **BY** ARTIDE**
8. ****ORDER** **BY** ARTIDE ;**

**------+---------+---------+---------+---------**

**ARTIDE AVG SUM MIN MAX**

**------+---------+---------+---------+---------**

**1 5.500000000000 11. 1. 10.**

**2 200.000000000000 600. 100. 300.**

**3 333.000000000000 333. 333. 333.**

**4 444.000000000000 444. 444. 444.**

***NUMBER OF ROWS DISPLAYED IS 4***

## Fonction CONCAT() agrégation

CONCAT() concatène des chaînes de caractères.

1. ****SELECT** \***
2. ****FROM** API16.FAM**
3. **;**

**---------+---------+-------**

**FAMCOD FAMLIB**

**---------+---------+-------**

**FA FANTAISIE**

**CU CUISINE**

**AB FAMILLE AB**

**AC FAMILLE AC**

**AD FAMILLE AD**

1. ****SELECT** FAMCOD, CONCAT(FAMLIB**
2. **, FAMLIB) **AS** CONCATENE**
3. ****FROM** API16.FAM**
4. **;**

**---------+---------+---------+---------+-------**

**FAMCOD CONCATENE**

**---------+---------+---------+---------+-------**

**FA FANTAISIE FANTAISIE**

**CU CUISINE CUISINE**

**AB FAMILLE AB FAMILLE AB**

**AC FAMILLE AC FAMILLE AC**

**AD FAMILLE AD FAMILLE AD**

## Fonction COUNT calcul

La fonction d’agrégation COUNT() permet de compter le nombre d’enregistrements dans une table

L’utilisation la plus générale consiste à utiliser la syntaxe suivante :

1. **SELECT** **COUNT**(colonne) **FROM** table

La requête admet toutes les complexités, avec notamment WHERE.

Pour compter le nombre total de lignes d’une table, il convient d’utiliser l’étoile « \* » qui signifie que l’on veut compter le nombre de lignes sur toutes les colonnes. La syntaxe est alors la suivante :

1. **SELECT COUNT(**\***) FROM** table

Il est aussi possible de connaitre le nombre de lignes sur une colonne en particulier. Les lignes qui possèdent la valeur NULL ne seront pas comptabilisées. La syntaxe pour compter les lignes sur la colonne « nom\_colonne » est la suivante :

1. **SELECT COUNT(**nom\_colonne**) FROM** table

Enfin, il est également possible de compter le nombre d’enregistrements distincts pour une colonne. La fonction ne comptabilisera pas les doublons pour une colonne choisie.

1. **SELECT COUNT(DISTINCT** nom\_colonne**)**
2. **FROM** table

**Exemple** de la table Article, avec les colonnes ARTIDE, identifiant, ARTLIB, libellé, ARTAM, famille, ARTMAT, marque.

1. **SELECT** **\***
2. **FROM** API16.ART
3. **ORDER** **BY** ARTMAR, ARTIDE ;

ARTIDE ARTLIB ARTFAM ARTMAR

------+---------+---------+---------+---------+

1 ? FA AB

3 NOUVEAU LIBELLE AB AB

2 NOUVEAU LIBELLE FA AC

4 ? AC AD

7 ? AD AD

5 ? AC AE

8 NOUVEAU LIBELLE AE AE

6 ? AC AF

9 ? AE AF

10 ? AF AF

11 ? AF AG

1. **SELECT** **COUNT**(ARTMAR)
2. **FROM** API16.ART ;

Réponse : 11

1. **SELECT** **COUNT**(**DISTINCT** ARTMAR)
2. **FROM** API16.ART ;

Réponse : 6

1. **SELECT COUNT(ARTMAR)**
2. **FROM API16.ART**
3. **GROUP BY ARTMAR ;**

**Réponse :**

**2**

**1**

**2**

**2**

**3**

**1**

***E610I NUMBER OF ROWS DISPLAYED IS 6***

**Il est plus judicieux dans ce cas d'afficher le contenu de la colonne comptée.**

1. **SELECT ARTMAR, COUNT(ARTMAR)**
2. **FROM API16.ART**
3. **GROUP BY ARTMAR ;**

**Réponse :**

**ARTMAR**

**---------+---------+---------+---------**

**AB 2**

**AC 1**

**AD 2**

**AE 2**

**AF 3**

**AG 1**

***DSNE610I NUMBER OF ROWS DISPLAYED IS 6***

**A savoir :** en général, en terme de performance, il est plutôt conseillé de filtrer les lignes avec GROUP BY si c’est possible, puis d’effectuer un COUNT(\*).

**Exemple**

Imaginons une table qui liste les utilisateurs d’un site web d’e-commerce :

| id | nom | ville | date\_inscription | nombre\_achat | id\_dernier\_achat |
| --- | --- | --- | --- | --- | --- |
| 1 | Charles | Pau | 2010-04-22 | 5 | 24 |
| 2 | Louis | Marseille | 2011-08-18 | 3 | 36 |
| 3 | Paul | Nantes | 2011-11-02 | 0 | NULL |
| 4 | Léon | Pau | 2012-09-01 | 1 | 7 |
| 5 | Paul | Bordeaux | 2013-01-10 | 0 | NULL |

**Utilisation de COUNT(\*)**

Pour compter le nombre d’utilisateurs total depuis que le site existe, il suffit d’utiliser COUNT(\*) sur toute la table :

1. **SELECT COUNT(**\***) FROM** utilisateur

**Résultat : 5**

\*

**Utilisation de COUNT(\*) avec WHERE**

Pour compter le nombre d’utilisateurs qui ont effectué au moins un achat, il suffit de faire la même chose mais en filtrant les enregistrements avec WHERE :

1. **SELECT COUNT(**\***) FROM** utilisateur
2. **WHERE** nombre\_achat > 0

**Résultat : 3**

**Utilisation de COUNT(colonne)**

Une autre méthode permet de compter le nombre d’utilisateurs ayant effectué au moins un achat. Il est possible de compter le nombre d’enregistrement sur la colonne « id\_dernier\_achat ». Sachant que la valeur est nulle s’il n’y a pas d’achat, les lignes ne seront pas comptées s’il n’y a pas eu d’achat. La requête est donc la suivante :

1. **SELECT COUNT(**id\_dernier\_achat**)**
2. **FROM** utilisateur

**Résultat : 3**

**Utilisation de COUNT(DISTINCT colonne)**

L’utilisation de la clause DISTINCT peut permettre de connaître le nombre de villes distinctes sur lesquels les visiteurs sont répartis.

1. **SELECT COUNT(DISTINCT** ville**)**
2. **FROM** utilisateur

Sachant qu’il y a 4 villes distinctes (cf. Pau, Marseille, Nantes et Bordeaux), le résultat se présente comme ceci :

**Résultat : 4**

**DATE ????**

* CURRENT\_DATE() date actuelle
* LOWER() afficher une chaîne en minuscule

## Fonction MAX() calcul

La fonction d’agrégation MAX() retourne la valeur maximale d’une colonne dans un ensemble de lignes (fonctionne bien pour un données numériques ou alphanumériques) pour récupérer la valeur maximum d’une colonne sur un ensemble de ligne. Cela s’applique à la fois pour des données numériques ou alphanumérique

Il est par exemple possible de rechercher le produit le plus cher dans une table d’une boutique en ligne.

**Syntaxe**

**SELECT MAX(**nom\_colonne**) FROM** table

Avec la commande GROUP BY, la requête peut ressembler à l’exemple ci-dessous:

**SELECT** colonne\_1, **MAX(**colonne\_2**) FROM** table

**GROUP BY** colonne₁

**Exemple**

**Soit un site d’e-commerce avec une table « produit » sur lequel on peut retrouver des produits informatiques.**

| id | nom | prix |
| --- | --- | --- |
| 1 | clavier | 50 |
| 2 | souris | 21 |
| 3 | écran | 120 |
| 4 | disque dur | 150 |

Pour extraire uniquement le tarif le plus élevé, voici la requête :

**SELECT MAX(**prix**) FROM** produit ;

Le résultat sera le suivant : **max(prix) : 150**

**Connaître l’enregistrement ayant la valeur maximale**

On peut afficher le nom du produit le plus cher et en même temps afficher son prix. Pour cela, il suffit simplement d’utiliser la fonction d’agrégation MAX() tout en sélectionnant les autres lignes.

1. **SELECT** id, nom, **MAX(**prix**) FROM** produit

**Le résultat de cette requête sera le suivant :**

| id | nom | max(prix) |
| --- | --- | --- |
| 4 | disque dur | 150 |

## Fonction MIN() calcul

La fonction d’agrégation MIN() retourne la plus petite valeur d’une colonne sélectionnée. Elle fonctionne comme la fonction MAX(). Elle s’applique donc aussi bien à des données numériques qu’à des données alphanumériques.

**Syntaxe**

**SELECT MIN(**nom\_colonne**) FROM** table

Comme il s’agit d’une fonction d’agrégation, il est possible de l’utiliser en complément de la commande GROUP BY. Cela permet de grouper des colonnes et de connaître la plus petite valeur pour chaque groupe. La syntaxe est alors la suivante:

**SELECT** colonne\_1, **MIN(**colonne\_2**)**

**FROM** table

**GROUP BY** colonne\_1

****ORDER BY** colonne\_1**

Cet exemple permet de grouper tous les enregistrements de « colonne\_1″ de la table et de connaître la plus petite valeur de « colonne\_2″ pour chacun de ces regroupement. Le ORDER BY est facultatif.

**Exemple**

Imaginons une table de produits divers affectés d'une catégorie, un nom, un prix et la date à laquelle ils ont été ajoutés.

**Table produits :**

| id | categorie | nom | prix | date\_ajout |
| --- | --- | --- | --- | --- |
| 1 | informatique | Ordinateur | 980 | 2013-01-24 |
| 2 | informatique | Imprimante | 70 | 2013-02-10 |
| 3 | maison | Canapé | 450 | 2013-02-11 |
| 4 | maison | Aspirateur | 200 | 2013-04-04 |

\*\*

**Utilisation simple**

Pour extraire le prix du produit le moins cher de la catégorie « maison », il est possible d’effectuer la requête SQL ci-dessous:

**SELECT MIN(**prix**) FROM** `produits`

**WHERE** `categorie` = 'maison'

**Résultat :**

**prix 200**

**Utilisation dans un GROUP BY**

Pour connaître le plus ancien produit par catégorie, on demande la date du premier ajout dans chaque catégorie, on ajoute la colonne dans la requête.

**SELECT** `categorie`, **MIN(**date\_ajout**)**

**FROM** `produits`

**GROUP BY** `categorie`

**Résultats :**

**categorie MIN(date\_ajout)**

maison 2013-02-11

**informatique 2013-02-11**

* + 1. Il est aussi possible de connaître le plus ancien produit par catégorie en utilisant la fonction MAX().

## Fonction RAND() calcul

Elle retourne un nombre aléatoire à virgule, compris entre 0 et 1. Le résultat de cette fonction est différent à chaque fois que la fonction est exécutée dans une requête SQL.

On peut l'utiliser pour ordonner une liste au hazard.

SELECT \* FROM `ville` ORDER BY RAND();

On peut l'utiliser pour tirer au sort une ligne d'une table. La fonction ROUND ne renvoie que les entiers. Si les identifiants se suivent avec des trous, la requête peut ne ramener aucun résultat.

SELECT \* FROM `ville`

WHERE `id` = ROUND( RAND() \* 9 ) + 1

## Fonction ROUND() calcul

Cette fonction ajoute 0,5 et retourne un entier.

**SELECT ARTIDE, DECIDE**

**, DECPRIX**

**, DECPRIX / 3**

**, **ROUND**(DECPRIX / 3, 0)**

**, **ROUND**(DECPRIX / 3, -1)**

**, **ROUND**(123.45,0)**

**, **ROUND**(123.54,-1)**

****FROM** API16.DEC**

****GROUP** **BY** ARTIDE, DECIDE**

****ORDER** **BY** ARTIDE, DECIDE ;**

**-----+---------+---------+---------+---------**

**ARTIDE DECIDE DECPRIX**

**-----+---------+---------+---------+---------**

**1 1. 121.60 40.533333333333 41.000000000000 40.00**

**1 2. 889.76 296.586666666666 297.000000000000 300.00**

**2 1. 80.00 26.666666666666 27.000000000000 30.00**

**2 2. 160.00 53.333333333333 53.000000000000 50.00**

**2 3. 121.60 40.533333333333 41.000000000000 40.00**

**3 1. 80.00 26.666666666666 27.000000000000 30.00**

**4 1. 984.00 328.000000000000 328.000000000000 330.00**

## Fonction SUM() calcul

La fonction d’agrégation SUM() calcule la somme totale des colonnes non NULL contenant des valeurs numériques.

Le résultat montre bien que le prix le moins cher est celui de l’aspirateur qui coûte 200€.

**Syntaxe**

**SELECT** **SUM**(nom\_colonne)

**FROM** table

**A savoir :** on peut filtrer les lignes avec la commande WHERE.

**Exemple**

Soit un système qui gère des factures, qui enregistre chaque achat dans une table, et qui utilise une table « facture » avec une ligne pour chaque produit.

**Table facture :**

| id | facture\_id | produit | prix |
| --- | --- | --- | --- |
| 1 | 1 | calculatrice | 17 |
| 2 | 1 | agrafeuse | 4 |
| 3 | 1 | ciseaux | 3 |
| 4 | 1 | agenda | 15 |
| 5 | 2 | calculatrice | 17 |
| 6 | 2 | agenda | 15 |

**Somme avec WHERE**

Pour calculer le montant de la facture n°1 il est possible d’utiliser la requête SQL suivante:

**SELECT SUM(**prix**) AS** prix\_total

**FROM** facture

**WHERE**facture\_id = 1

**Résultat : prix\_total 39**

En effet tous les achats de la facture n°1 représente un montant de 39€ (somme de 17€ + 4€ + 3€ + 15€).

**SUM avec GROUP BY**

Pour calculer la somme de chaque facture, il est possible de grouper les lignes en se basant sur la colonne « facture\_id ».

**SELECT** facture\_id, **SUM(**prix**) AS** prix\_total

**FROM** facture

**GROUP BY** facture\_id

**Résultat :**

**facture\_id prix\_total**

**1 39**

**2 32**

# Fonctions de chaînes de caractères

Les fonctions SQL sur les chaînes de caractères permettent d’ajouter de nombreuses fonctions mono-lignes, c'est à dire qu'’elles ne s’appliquent qu’à une seule ligne en même temps.

Certaines fonctions possèdent une syntaxe différente selon le SGBD ou y sont absentes.

| Database | CHR | CHAR | ASCII | NCHR | NCHAR | ASCIISTR |
| --- | --- | --- | --- | --- | --- | --- |
| Oracle | O | N | O | O | N | O |
| SQL Server | N | O | O | N | O | N |
| MySQL | N | O | O | N | N | N |
| Postgres | O | N | O | N | N | N |

## Fonction BTRIM() caractère

**BTRIM supprime les caractères spécifiés du début et de la fin d’une chaîne.**

**Voir la onction TRIM().**

## Fonction CHAR() caractère

La fonction CHAR() retourne l'octet correspondant à l’encodage du classement de la base de données active. Le paramètre est un entier compris entre 0 et 255, ou une expression valant un entier compris entre 0 et 255.

**Syntaxe** : CHAR(entier)

**Exemples :**

1. CHAR(9) : tabulation horizontale
2. CHAR(10) : saut de ligne
3. CHAR(13) : retour chariot
4. CHAR(32) : espace
5. CHAR(65) : A majuscule
6. CHAR(128) : signe Euro
7. CHAR(176) : signe degré

**Remarque** : cf la fonction CHR() dans laSGBD PostgreSQL

## Fonction CHAR\_LENGTH() caractère

Cette fonction renvoie la longueur d'une chaîne, mesurée en octets. Elle prend en compte les espaces, les caractères spéciaux et les caractères alphanumériques standard. Chaque lettre (æ), (Æ), (œ), (Œ) compte pour 1 octet.

1. SELECT CHAR\_LENGTH("Conseils SQL")
2. AS longueur ;

**Réponse**: 12

**Remarque** : pour MySQL utiliser CHARACTER\_LENGTH().

## Fonction CHARACTER\_LENGTH() caractère

SGBD : MySQL

La fonction CHARACTER\_LENGTH est identique à la fonction CHAR\_LENGTH().

## Fonction CHARINDEX() caractère [SQL Server]

**La fonction SQL CHARINDEX trouve la position d'une sous-chaîne ou d'une expression dans une chaîne donnée. La réponse est 0 si elle n'est pas trouvée.**

**Il existe des options pour spécifier si on veut tenir compte de la casse.**

1. **SELECT CHARINDEX**
2. **( 'EXEMPLE', 'CECI EST UN EXEMPLE') ;**

**Réponse : 13**

## Fonction CHR() caractère [PostgreSQL]

**Voir la fonction CHAR().**

## ****Fonction SUBSTRING() caractère****

La fonction SUBSTRING() dans le langage SQL (ou SUBSTR() ) est utilisée pour segmenter une chaîne de caractère. Autrement dit, cela permet d’extraire une partie d’un chaîne, par exemple pour tronquer un texte.

**Compatibilité :**

* **MySQL :** SUBSTR() ou SUBSTRING()
* **PostgreSQL :** SUBSTR() ou SUBSTRING()
* **Oracle :** SUBSTR() ou SUBSTRING()
* **SQL Server :** SUBSTRING()
* **SQLite :** SUBSTR()
* **Firebird :** SUBSTRING()

**Syntaxe**

La fonction SUBSTRING() peut s’utiliser de 4 façons différentes, que voici :

* **SUBSTRING(chaine, debut) :** retourne la chaîne de caractère de « chaine » à partir de la position définie par « debut » (position en nombre de caractères)
* **SUBSTRING(chaine FROM debut) :** idem que précédent
* **SUBSTRING(chaine, debut, longueur) :** retourne la chaîne de caractère « chaine » en partant de la position définie par « debut » et sur la longueur définie par « longueur »
* **SUBSTRING(chaine FROM debut FOR longueur) :** idem que précédent

**A savoir :** il ne faut pas oublier que la fonction est aussi parfois appelée SUBSTR().

### Exemple de requête

Cette fonction peut s’utiliser dans une requête SQL en utilisant une syntaxe comme celle-ci:

**SELECT SUBSTR(**nom\_colonne, 3, 10**) FROM** tableau

Dans cet exemple, le contenu de la colonne « nom\_colonne » sera tronqué à partir du 4ème caractère sur 10 caractères.

### Exemple

Imaginons une base de données qui contient une liste de pays en français et en anglais.

**Table pays :**

| id | nom\_fr\_fr | nom\_en\_gb |
| --- | --- | --- |
| 1 | FRANCE | FRANCE |
| 2 | ESPAGNE | SPAIN |
| 3 | ALLEMAGNE | GERMANY |
| 4 | CHINE | CHINA |

Une requête SQL va être construite pour extraire les informations suivantes :

* L’identifiant du pays
* Le nom du pays en français
* Les 2 premiers caractères du nom en français (soit à partir du 1er caractère et sur une longueur de 2 caractères)
* Tout le nom en français, sans les 2 premiers caractères (soit tout le nom à partir du 3ème caractère)

Pour récupérer toutes ces données, il convient d’utiliser la requête SQL suivante :

**SELECT** id, nom\_fr\_fr, **SUBSTR(**nom\_fr\_fr, 1,2**)**

, **SUBSTR(**nom\_fr\_fr, 3**)**

**FROM** pays

Cette requête va retourner les résultats suivants :

| id | nom\_fr\_fr | SUBSTR(nom\_fr\_fr, 1, 2) | SUBSTR(nom\_fr\_fr, 3) |
| --- | --- | --- | --- |
| 1 | FRANCE | FR | ANCE |
| 2 | ESPAGNE | ES | PAGNE |
| 3 | ALLEMAGNE | AL | LEMAGNE |
| 4 | CHINE | CH | INE |

Ce table de résultat démontre bien qu’il est possible de segmenter un texte simplement.

### Exemple d’utilisation

* Astuce SQL pour supprimer le premier ou dernier caractère d’une colonne
* SUBSTRING\_INDEX() [MySQL]
* TO\_ASCII() [PostgreSQL]
* TO\_HEX() [PostgreSQL]
* TRANSLATE() [PostgreSQL]
* TRIM() supprime les caractères vides en début et fin de chaîne [MySQL, PostgreSQL]
* UCASE() synonyme de UPPER() [MySQL]
* UNHEX() [MySQL]
* UNICODE() [SQL Server]
* UPPER() tout retourner en majuscule [MySQL, PostgreSQL, SQL Server]

## Fonction UPPER() caractère

* afficher une chaîne en majuscule

# ****Fonctions mathématiques / numérique****

Dans le langage SQL il existe une multitude de fonctions pour retourner les résultats de la façon souhaités. Il existe également de nombreuses fonctions mathématiques pour effectuer des calculs ou des statistiques concernant les données contenus dans une base de données.

## ****Fonction ROUND() calcul****

**SGBD** : tous. Mais MySQL, PostgreSQL, SQL Server offrent des options supplémentaires.

La fonction ROUND() arrondit un résultat numérique. Cette fonction permet soit d’arrondir sans utiliser de décimale pour retourner un nombre entier (c’est-à-dire : aucun chiffre après la virgule), soit de choisir le nombre de chiffres après la virgule.

**Syntaxe**

1. **SELECT** **ROUND(**nom\_colonne**) FROM** nom\_table ;

Dans cet exemple la colonne « nom\_colonne » contient des données numériques (exemple : FLOAT, INT, NUMERIC …) et retournera uniquement des entiers.

Le résultat est arrondi à l’entier le plus proche : au chiffre entier supérieur si les décimales sont supérieur à 0,5, sinon le nombre est tronqué.

Pour obtenir le résultat avec 2 chiffres de décimal il convient de spécifier le nombre de décimales souhaité comme 2ème argument.

1. **SELECT ROUND(**nom\_colonne, 2**) FROM** nom\_table ;

**Exemple**

**Nombre\_1 = 78,421**

**Nombre\_2 = 78,857**

1. **SELECT** **Nombre\_1 **ROUND**, Nombre\_2** **ROUND** 
   * + 1. **FROM** table\_chiffre ;

**Résultat :**

**Nombre\_1 = 78**

**Nombre\_2 = 79**

**Exemple d'arrondi à 2 chiffres après la virgule**

**Nombre\_1 = 78,421**

**Nombre\_2 = 78,857**

1. **SELECT** **Nombre\_1**  **ROUND(Nombre\_1**, 2)
   * 1. **, Nombre\_2  ROUND(Nombre\_2**, 2)
2. **FROM** `resultat` ;

**Résultat :**

**Nombre\_1 = 78,42**

**Nombre\_2 = 78,86**

**Nombre\_1 = 78,421**

**Nombre\_3 = 71,576**

**Exemple d'arrondi à la dizaine**

1. **SELECT** **Nombre\_1**  **ROUND(Nombre\_1**, -1)
   * 1. **, Nombre\_3  ROUND(Nombre\_3**, -1)
2. **FROM** `resultat` ;

**Résultat :**

**Nombre\_1 = 80**

**Nombre\_2 = 70**

# ****Fonctions de dates et d’heures****

Il est important en SQL de pouvoir connaître la date et l’heure. Il existe une multitude de fonctions qui concernent les éléments temporels pour lire ou écrire plus facilement des données à une date précise ou à un intervalle de date.

## Fonctions SQL utiles pour les dates et les heures

AGE() soustraire 2 dates [PostgreSQL]

ADDDATE() ajouter une période sous forme d’heures à une date [MySQL]

ADDTIME() ajouter une période sous forme d’une date à une autre date [MySQL]

CONVERT\_TZ() convertir d’une « timezone » à une autre [MySQL]

CURDATE() récupérer la date courante [MySQL]

CURRENT\_DATE() synonyme de CURDATE() [MySQL, PostgreSQL]

CURRENT\_TIME() synonyme de CURTIME() [MySQL, PostgreSQL]

CURRENT\_TIMESTAMP() synonyme de NOW() [MySQL, PostgreSQL, SQL Server]

CURTIME() Return the current time [MySQL]

DATE() extraire une date à partir d’une chaîne contenant une valeur au format DATE ou DATETIME [MySQL]

DATE\_ADD() ajouter une valeur au format TIME à une date [MySQL]

DATE\_FORMAT() formater la date pour l’afficher selon le format choisi [MySQL]

DATE\_PART() extraire un élément d’un DATETIME (cf. heure, minute, jour, mois …) [PostgreSQL]

DATE\_SUB() soustraire une valeur au format TIME à une date [MySQL]

DATE\_TRUNC() tronquer un DATETIME avec la précision souhaitée (cf. mois, jour, heure, minute …) [PostgreSQL]

DATEADD() ajoute un élément (minute, heure, jour, mois, année …) à une date spécifiée [SQL Server]

DATEDIFF() déterminer le nombre de jours entre 2 dates [MySQL, SQL Server]

DATENAME() retourner une partie d’une date (cf. minute, heure, jour, semaine, mois …) [SQL Server]

DATEPART() retourne un entier qui représente la partie d’une date (cf. minute, heure, jour, mois, année …) [SQL Server]

DAY() synonyme de DAYOFMONTH() [MySQL, SQL Server]

DAYNAME() retourne le nom du jour de la semaine [MySQL]

DAYOFMONTH() retourner le jour dans le mois (de 1 à 31) [MySQL]

DAYOFWEEK() retourner le jour dans la semaine (1=dimanche, 2=lundi, 3=mardi …) [MySQL]

DAYOFYEAR() retourner le jour dans l’année (de 1 à 366) [MySQL]

EXTRACT() extraire une partie d’une date [MySQL, PostgreSQL]

FROM\_DAYS() convertir un nombre de jour en une date [MySQL]

FROM\_UNIXTIME() convertir un timestamp UNIX en un date au format DATETIME [MySQL]

GET\_FORMAT() retourne le format d’une date dans une chaîne de caractère [MySQL]

GETDATE() obtenir la date courante du système, sans le décalage horaire [SQL Server]

GETUTCDATE() obtenir la date courante UTC [SQL Server]

HOUR() extraire le nombre d’heure pour une heure au format HH:MM:SS [MySQL]

ISDATE() retourne 1 si la valeur en paramètre est dans l’un des formats suivants : TIME, DATE ou DATETIME [SQL Server]

ISFINITE() tester pour savoir si une date ou une période de temps est finie [PostgreSQL]

JUSTIFY\_HOURS() ajuster un intervalle de 24 heures en tant que « 1 jour » ou un intervalle de 30 jours en tant que « 1 mois » [PostgreSQL]

LAST\_DAY() retourner le dernier jour du mois d’une date [MySQL]

LOCALTIME() synonyme de NOW() [MySQL, PostgreSQL]

LOCALTIMESTAMP() synonyme de NOW() [MySQL, PostgreSQL]

MAKEDATE() retourne une date à partir d’une année et du numéro du jour dans cette année [MySQL]

MAKETIME() créer une heure au format TIME à partir d’une heure, des minutes et du nombre de secondes [MySQL]

MICROSECOND() retourne le nombre de microsondes à partir d’une heure ou d’un DATETIME [MySQL]

MINUTE() extraire le nombre de minutes d’une heure au format HH:MM:SS [MySQL]

MONTH() extraire le numéro du mois à partir d’une date [MySQL, SQL Server]

MONTHNAME() retourne le nom du mois [MySQL]

NOW() obtenir la date courante [MySQL, PostgreSQL]

PERIOD\_ADD() ajoute un nombre définie de mois à une période [MySQL]

PERIOD\_DIFF() retourne le nombre de mois entre 2 périodes définies [MySQL]

QUARTER() Return the quarter from a date argument [MySQL]

SEC\_TO\_TIME() convertir un nombre de secondes en une heure et minutes au format HH:MM:SS [MySQL]

SECOND() extraire le nombre de secondes d’une heure au format HH:MM:SS [MySQL]

STR\_TO\_DATE() convertir une chaîne de caractère en date [MySQL]

SUBDATE() synonyme de DATE\_SUB() lorsque la fonction est invoquée avec 3 arguments [MySQL]

SUBTIME() soustraire une heure [MySQL]

SWITCHOFFSET() retourne une valeur DATETIMEOFFSET en changeant le fuseau horaire [SQL Server]

SYSDATE() retourne la date et l’heure courante [MySQL]

SYSDATETIME() retourne le DATETIME (date et heure) de l’ordinateur sur lequel est installé la base de données [SQL Server]

SYSDATETIMEOFFSET() retourne le DATETIMEOFFSET du système avec le décalage du fuseau horaire inclus [SQL Server]

SYSUTCDATETIME() retourne le DATETIME en heure UTC de l’ordinateur sur lequel la base est installées [SQL Server]

TIME\_FORMAT() formater une date dans un autre format [MySQL]

TIME\_TO\_SEC() convertir une heure de format HH:MM:SS en nombre de secondes [MySQL]

TIME() extraire l’heure/minutes/secondes au format HH:MM:SS à partir d’une date [MySQL]

TIMEDIFF() retourne la durée entre 2 heures [MySQL]

TIMESTAMP() permet de convertir une DATE au format DATETIME [MySQL]

TIMESTAMPADD() ajoute un intervalle à une expression au format DATETIME [MySQL]

TIMESTAMPDIFF() soustraire un intervalle à partir d’une expression au format DATETIME [MySQL]

TIMEOFDAY() retourne la date et heure courante [PostgreSQL]

TO\_DAYS() retourne le nombre de jour à partir d’une date [MySQL]

TO\_SECONDS() Return the date or datetime argument converted to seconds since Year 0 [MySQL]

TODATETIMEOFFSET() retourne un DATETIMEOFFSET à partir d’un DATETIME [SQL Server]

UNIX\_TIMESTAMP() retourner le timestamp UNIX (nombre de secondes depuis le 1er janvier 1970) [MySQL]

UTC\_DATE() retourne la date GMT courante [MySQL]

UTC\_TIME() retourne l’heure GMT courante [MySQL]

UTC\_TIMESTAMP() retourne la date et heure GMT courante [MySQL]

WEEK() déterminer le numéro de la semaine dans une année, à partir d’une date [MySQL]

WEEKDAY() déterminer le jour de la semaine à partir d’une date (0=lundi, 1:mardi, 2=mercredi …) [MySQL]

WEEKOFYEAR() déterminer le numéro de la semaine dans une année, à partir d’une date [MySQL]

YEAR() extraire l’année d’une date [MySQL, SQL Server]

YEARWEEK() retourne l’année et la semaine à partir d’une date [MySQL]

## ****Fonction NOW() date****

La fonction NOW() retourne la date et l’heure du système.

Elle est notamment utilisée pour mémoriser le moment de l'ajout ou de la modification d’une donnée, dans un DATETIME ou un DATE().

**Syntaxe : SELECT NOW()**;

**Exemple :** 2024-09-22 16:19:43

La date est l’heure sont affiché au format AAAA-MM-JJ HH:MM:SS (A=Année, M=Mois, J=Jour, H=Heure, M=Minute, S=Seconde).

**Exemple :**

1. **INSERT INTO** utilisateur (login, date\_ajout )
2. **VALUES** ( 'jojo', **NOW()** );

**Résultat** : jojo, 2013-04-11 17:36:11

\*\*\*

**Exemple qui renvoie la date courante :**

La fonction DATE() permet d’obtenir une date à partir d’une chaîne de caractère au format DATE ou DATETIME.

**SELECT DATE( NOW() )**;

**Résultat ce jour-là :** 2025-07-22

\*\*\*

**Exemple qui extrait la date avec la fonction CAST() :**

La fonction CAST() convertit un type en un autre type, comme par exemple, un format DATETIME en un résultat de type DATE.

1. **SELECT CAST(NOW() AS DATE)**;

**Résultat ce jour-là :** 2024-09-22

## ****Fonction MONTH() date****

La fonction MONTH() extrait le numéro de mois à partir d’une date au format AAAA-MM-JJ. Si la date d’entrée est par exemple ‘2025-02-03′, la valeur de retour sera ‘3’. Cela permet d’associer le numéro de mois au nom du mois (3 = Mars).

**SGBD SQL Server :** la fonction accepte les valeurs de type time, date, smalldatetime, datetime, datetime2 ou datetimeoffset.

**SGBD PostgreSQL :** la fonction MONTH() n’existe pas avec ce SGBD. Pour simuler le fonctionnement, il est possible d’utiliser la fonction EXTRACT() en spécifiant le type « month ».

**Syntaxe**

1. **SELECT MONTH(** date **)**;

**Exemples**

1. **SELECT MONTH(**'2025-06-06'**)**; -- résultat : 6
2. **SELECT MONTH(**'2025-01-01'**)**; -- résultat : 1
3. **SELECT MONTH(**'1999-12-01'**)**; -- résultat : 12
4. **SELECT MONTH(**'1815-06-18'**)**; -- résultat : 6

**Important :** la fonction ne retourne pas les zéros pour les mois de ’01’ à ’09’.

**Exemple** avec la commande CASE

On peut renvoyer un libellé en fonction du numéro du mois.

**SELECT CASE MONTH(**'2025-07-18')

**WHEN** 1 **THEN** 'janvier'

**WHEN** 2 **THEN** 'février'

**WHEN** 3 **THEN** 'mars'

**WHEN** 4 **THEN** 'avril'

**WHEN** 5 **THEN** 'mai'

**WHEN** 6 **THEN** 'juin'

**WHEN** 7 **THEN** 'juillet'

**WHEN** 8 **THEN** 'août'

**WHEN** 9 **THEN** 'septembre'

**WHEN** 10 **THEN** 'octobre'

**WHEN** 11 **THEN** 'novembre'

**ELSE** 'décembre'

**END**;

-- résultat : 'juillet'

**Voir aussi :**

* DAY() pour extraire le jour d’une date
* YEAR() pour extraire l’année d’une date

## ****Fonction YEAR() date****

La fonction YEAR() extrait l'année à partir d’une date au format AAAA-MM-JJ.

**SGBD SQL Server :** la fonction accepte les valeurs de type time, date, smalldatetime, datetime, datetime2 ou datetimeoffset.

**SGBD PostgreSQL :** la fonction n’existe pas avec PostgreSQL. Pour extraire une date avec ce SGBD on utilise la fonction EXTRACT() et on spécifie le type « year ».

**Syntaxe SELECT YEAR(** date );

Dans cet exemple le paramètre « date » peut correspondre à une date telle que ‘2014-01-01′ et retournera une chaîne de 4 caractères.

**Exemple**

1. **SELECT YEAR(**'2025-01-01'); -- résultat : 2025
2. **SELECT YEAR(**'1999-12-31'); -- résultat : 1999
3. **SELECT YEAR(**'1815-06-06'); -- résultat : 1815
4. **SELECT YEAR(**'2000-12-31'); -- résultat : 2000

**A noter :** si la date d’entrée est au format AAAA-MM-JJ, on peut utiliser la fonction LEFT() qui tronque la date aux 4 premiers caractères.

On peut aussi utiliser la [fonction SUBSTR() ou SUBSTRING()](http://sql.sh/fonctions/substring) qui tronque également les premiers caractères.

**Voir aussi :**

* DAY() pour extraire le jour d’une date
* [MONTH()](http://sql.sh/fonctions/month) pour extraire l’année d’une date

# ****Fonctions de chiffrements****

# Dates

## Fonction NOW() date

date et heure actuelle

### cobol

timestamps always require PIC X(26) host variables  
date PIC X(10)  
time PIC X(08)

### cf IBM

Cf [http://www.ibm.com/developerworks/data/library/techarticle/0211yip/0211yip3.html#ibm-content](http://www.ibm.com/developerworks/data/library/techarticle/0211yip/0211yip3.html" \l "ibm-content)

This article is written for IBM® DB2® for Linux, UNIX®, and Windows®.

## Introduction

**IBM DB2 e-kit for Database Professionals**

Learn how easy it is to get trained and certified for DB2 for Linux, UNIX, and Windows with the IBM DB2 e-kit for Database Professionals. Register now, and expand your skills portfolio, or extend your DBMS vendor support to include DB2.

This short article is intended for those who are new to DB2 and wish to understand how to manipulate dates and times. Most people who have worked with other databases are pleasantly surprised by how easy it is in DB2.

## The basics

To get the current date, time, and timestamp using SQL, reference the appropriate DB2 registers:

**SELECT current date FROM** sysibm.sysdummy1

**SELECT current time FROM** sysibm.sysdummy1

**SELECT current timestamp** FROM sysibm.sysdummy1

The sysibm.sysdummy1 table is a special in-memory table that can be used to discover the value of DB2 registers as illustrated above. You can also use the VALUES keyword to evaluate the register or expression. For example, from the DB2 Command Line Processor (CLP), the following SQL statements reveal similar information:

VALUES current date

VALUES current time

VALUES current timestamp

For the remaining examples, I will simply provide the function or expression without repeating SELECT ... FROM sysibm.sysdummy1 or using the VALUES clause.

To get the current time or current timestamp adjusted to GMT/CUT, subtract the current timezone register from the current time or timestamp:

current time - current timezone

current timestamp - current timezone

Given a date, time, or timestamp, you can extract (where applicable) the year, month, day, hour, minutes, seconds, and microseconds portions independently using the appropriate function:

**YEAR (current timestamp)**

**MONTH (current timestamp)**

**DAY (current timestamp)**

**HOUR (current timestamp)**

**MINUTE (current timestamp)**

**SECOND (current timestamp)**

**MICROSECOND (current timestamp)**

Extracting the date and time independently from a timestamp is also very easy:

**DATE (current timestamp)**

**TIME (current timestamp)**

You can also perform date and time calculations using, for lack of a better term, English:

**current date +** 1 **YEAR**

**current date** + 3 **YEARS** + 2 **MONTHS** + 15 **DAYS**

**current time** + 5 **HOURS** - 3 **MINUTES** + 10 **SECONDS**

To calculate how many days there are between two dates, you can subtract dates as in the following:

**days (current date)** - **days (date**('1999-10-22'))

And here is an example of how to get the current timestamp with the microseconds portion reset to zero:

**CURRENT TIMESTAMP** - **MICROSECOND**

**(current timestamp) MICROSECONDS**

If you want to concatenate date or time values with other text, you need to convert the value into a character string first. To do this, you can simply use the CHAR() function:

**char(current date)**

**char(current time)**

**char(current date +** 12 **hours)**

To convert a character string to a date or time value, you can use:

**TIMESTAMP ('**2002-10-20-12.00.00.000000')

**TIMESTAMP (**'2002-10-20 12:00:00')

**DATE** ('2002-10-20')

**DATE** ('10/20/2002')

**TIME** ('12:00:00')

**TIME** ('12.00.00')

The TIMESTAMP(), DATE() and TIME() functions accept several more formats. The above formats are examples only and I'll leave it as an exercise for the reader to discover them.

**Warning**: What happens if you accidentally leave out the quotes in the DATE function? The function still works, but the result is not correct:

**SELECT DATE(**2001-09-22) F**ROM SYSIBM.SYSDUMMY1**;

Answer:

05/24/0006

Why the 2,000 year difference in the above results? When the DATE function gets a character string as input, it assumes that it is valid character representation of a DB2 date, and converts it accordingly. By contrast, when the input is numeric, the function assumes that it represents the number of days minus one from the start of the current era (that is, 0001-01-01). In the above query the input was 2001-09-22, which equals (2001-9)-22, which equals 1970 days.

## Date functions

Sometimes, you need to know how the difference between two timestamps. For this, DB2 provides a built in function called TIMESTAMPDIFF(). The value returned is an approximation, however, because it does not account for leap years and assumes only 30 days per month. Here is an example of how to find the approximate difference in time between two dates:

timestampdiff (<n>

, char( timestamp('2002-11-30-00.00.00')

- timestamp('2002-11-08-00.00.00')))

In place of <n>, use one of the following values to indicate the unit of time for the result:

* 1 = Fractions of a second
* 2 = Seconds
* 4 = Minutes
* 8 = Hours
* 16 = Days
* 32 = Weeks
* 64 = Months
* 128 = Quarters
* 256 = Years

Using timestampdiff() is more accurate when the dates are close together than when they are far apart. If you need a more precise calculation, you can use the following to determine the difference in time (in seconds):

(DAYS(t1) - DAYS(t2)) \* 86400 +

(MIDNIGHT\_SECONDS(t1) - MIDNIGHT\_SECONDS(t2))

For convenience, you can also create an SQL user-defined function of the above:

CREATE FUNCTION secondsdiff

(t1 TIMESTAMP, t2 TIMESTAMP)

RETURNS INT

RETURN (

(DAYS(t1) - DAYS(t2)) \* 86400 +

(MIDNIGHT\_SECONDS(t1) - MIDNIGHT\_SECONDS(t2))

)

If you need to determine if a given year is a leap year, here is a useful SQL function you can create to determine the number of days in a given year:

CREATE FUNCTION daysinyear(yr INT)

RETURNS INT

RETURN (CASE (mod(yr, 400))

WHEN 0 THEN 366

ELSE CASE (mod(yr, 4))

WHEN 0 THEN

CASE (mod(yr, 100))

WHEN 0 THEN 365

ELSE 366 END

ELSE 365 END

END)

Finally, here is a chart of built-in functions for date manipulation. The intent is to help you quickly identify a function that might fit your needs, not to provide a full reference. Consult the SQL Reference for more information on these functions.

SQL Date and Time functions are as follows:

1. **DAYNAME**: Returns a mixed case character string containing the name of the day (e.g., Friday) for the day portion of the argument.
2. **DAYOFWEEK**: Returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Sunday.
3. **DAYOFWEEK\_ISO**: Returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Monday.
4. **DAYOFYEAR**: Returns the day of the year in the argument as an integer value in the range 1-366.
5. **DAYS**: Returns an integer representation of a date.
6. **JULIAN\_DAY**: Returns an integer value representing the number of days from January 1, 4712 B.C. (the start of Julian date calendar) to the date value specified in the argument.
7. **MIDNIGHT\_SECONDS**: Returns an integer value in the range 0 to 86 400 representing the number of seconds between midnight and the time value specified in the argument.
8. **MONTHNAME**: Returns a mixed case character string containing the name of month (e.g., January) for the month portion of the argument.
9. **TIMESTAMP\_ISO**: eturns a timestamp value based on date, time or timestamp argument.
10. **TIMESTAMP\_FORMAT**: Returns a timestamp from a character string that has been interpreted using a character template.
11. **TIMESTAMPDIFF**: Returns an estimated number of intervals of the type defined by the first argument, based on the difference between two timestamps.
12. **TO\_CHAR**: Returns a character representation of a timestamp that has been formatted using a character template. TO\_CHAR is a synonym for VARCHAR\_FORMAT.
13. **TO\_DATE**: Returns a timestamp from a character string that has been inter-preted using a character template. TO\_DATE is a synonym for TIMESTAMP\_FORMAT.
14. **WEEK**: Returns the week of the year of the argument as an integer value in range 1-54. The week starts with Sunday.
15. **WEEK\_ISO**: Returns the week of the year of the argument as an integer value in the range 1-53.

## Changing the date format

A common question I get often relates to the presentation of dates. The default format used for dates is determined by the territory code of the database (which can be specified at database creation time). For example, my database was created using territory=US. Therefore the date format looks like the following:

values current date

1

----------

05/30/2003

1 record(s) selected.

That is, the format is MM/DD/YYYY. If you want to change the format, you can bind the collection of db2 utility packages to use a different date format. The formats supported are:

* **DEF**: Use a date and time format associated with the territory code.
* **EUR**: Use the IBM standard for Europe date and time format.
* **ISO**: Use the date and time format of the International Standards Organization.
* **JIS**: Use the date and time format of the Japanese Industrial Standard.
* **LOC**: Use the date and time format in local form associated with the territory code of the database.
* **USA**: Use the IBM standard for U.S. date and time format.

To change the default format to ISO on windows (YYYY-MM-DD), do the following steps:

* On the command line, change your current directory to sqllib\bnd.

For example:   
**On Windows**: c:\program files\IBM\sqllib\bnd  
**On UNIX**:/home/db2inst1/sqllib/bnd

* Connect to the database from the operating system shell as a user with SYSADM authority:

db2 connect to DBNAME

db2 bind @db2ubind.lst datetime ISO blocking all grant public

(In your case, substitute your database name and desired date format for DBNAME and ISO, respectively.)

Now, you can see that the database uses ISO date format:

values current date

1

----------

2003-05-30

1 record(s) selected.

**Custom Date/Time Formatting**

In the last example, we demonstrated how to change the way DB2 presents dates in some localized formats. But what if you wish to have a custom format such as 'yyyymmdd'? The best way to do this is by writing your own custom formatting function.

Here is the UDF:

create function ts\_fmt(TS timestamp, fmt varchar(20))

returns varchar(50)

return

with tmp (dd,mm,yyyy,hh,mi,ss,nnnnnn) as

(

select

substr( digits (day(TS)),9),

substr( digits (month(TS)),9) ,

rtrim(char(year(TS))) ,

substr( digits (hour(TS)),9),

substr( digits (minute(TS)),9),

substr( digits (second(TS)),9),

rtrim(char(microsecond(TS)))

from sysibm.sysdummy1

)

select

case fmt

when 'yyyymmdd'

then yyyy || mm || dd

when 'mm/dd/yyyy'

then mm || '/' || dd || '/' || yyyy

when 'yyyy/dd/mm hh:mi:ss'

then yyyy || '/' || mm || '/' || dd || ' ' ||

hh || ':' || mi || ':' || ss

when 'nnnnnn'

then nnnnnn

else

'date format ' || coalesce(fmt,' <null> ') ||

' not recognized.'

end

from tmp

</null>

The function code may appear complex at first, but upon closer examination, you'll see that it is actually quite simple and elegant. First, we use a common table expression (CTE) to strip apart a timestamp (the first input parameter) into its individual components. From there, we check the format provided (the second input parameter) and reassemble the timestamp using the requested format and parts.

The function is also very flexible. To add another pattern simply append another WHEN clause with the expected format. When an unexpected pattern is encountered, an error message is returned.

Usage examples:

values ts\_fmt(current timestamp,'yyyymmdd')

'20030818'

values ts\_fmt(current timestamp,'asa')

'date format asa not recognized.'

# Dates

### cobol

timestamps always require PIC X(26) host variables  
date - PIC X(10)  
time PIC X(08)

### cf IBM

Cf [http://www.ibm.com/developerworks/data/library/techarticle/0211yip/0211yip3.html#ibm-content](http://www.ibm.com/developerworks/data/library/techarticle/0211yip/0211yip3.html" \l "ibm-content)

This article is written for IBM® DB2® for Linux, UNIX®, and Windows®.

## Introduction

**IBM DB2 e-kit for Database Professionals**

Learn how easy it is to get trained and certified for DB2 for Linux, UNIX, and Windows with the IBM DB2 e-kit for Database Professionals. Register now, and expand your skills portfolio, or extend your DBMS vendor support to include DB2.

This short article is intended for those who are new to DB2 and wish to understand how to manipulate dates and times. Most people who have worked with other databases are pleasantly surprised by how easy it is in DB2.

## The basics

To get the current date, time, and timestamp using SQL, reference the appropriate DB2 registers:

SELECT current date FROM sysibm.sysdummy1

SELECT current time FROM sysibm.sysdummy1

SELECT current timestamp FROM sysibm.sysdummy1

The sysibm.sysdummy1 table is a special in-memory table that can be used to discover the value of DB2 registers as illustrated above. You can also use the VALUES keyword to evaluate the register or expression. For example, from the DB2 Command Line Processor (CLP), the following SQL statements reveal similar information:

VALUES current date

VALUES current time

VALUES current timestamp

For the remaining examples, I will simply provide the function or expression without repeating SELECT ... FROM sysibm.sysdummy1 or using the VALUES clause.

To get the current time or current timestamp adjusted to GMT/CUT, subtract the current timezone register from the current time or timestamp:

current time - current timezone

current timestamp - current timezone

Given a date, time, or timestamp, you can extract (where applicable) the year, month, day, hour, minutes, seconds, and microseconds portions independently using the appropriate function:

YEAR (current timestamp)

MONTH (current timestamp)

DAY (current timestamp)

HOUR (current timestamp)

MINUTE (current timestamp)

SECOND (current timestamp)

MICROSECOND (current timestamp)

Extracting the date and time independently from a timestamp is also very easy:

DATE (current timestamp)

TIME (current timestamp)

You can also perform date and time calculations using, for lack of a better term, English:

current date + 1 YEAR

current date + 3 YEARS + 2 MONTHS + 15 DAYS

current time + 5 HOURS - 3 MINUTES + 10 SECONDS

To calculate how many days there are between two dates, you can subtract dates as in the following:

days (current date) - days (date('1999-10-22'))

And here is an example of how to get the current timestamp with the microseconds portion reset to zero:

CURRENT TIMESTAMP - MICROSECOND (current timestamp) MICROSECONDS

If you want to concatenate date or time values with other text, you need to convert the value into a character string first. To do this, you can simply use the CHAR() function:

char(current date)

char(current time)

char(current date + 12 hours)

To convert a character string to a date or time value, you can use:

TIMESTAMP ('2002-10-20-12.00.00.000000')

TIMESTAMP ('2002-10-20 12:00:00')

DATE ('2002-10-20')

DATE ('10/20/2002')

TIME ('12:00:00')

TIME ('12.00.00')

The TIMESTAMP(), DATE() and TIME() functions accept several more formats. The above formats are examples only and I'll leave it as an exercise for the reader to discover them.

**Warning**: What happens if you accidentally leave out the quotes in the DATE function? The function still works, but the result is not correct:

SELECT DATE(2001-09-22) FROM SYSIBM.SYSDUMMY1;

Answer:

05/24/0006

Why the 2,000 year difference in the above results? When the DATE function gets a character string as input, it assumes that it is valid character representation of a DB2 date, and converts it accordingly. By contrast, when the input is numeric, the function assumes that it represents the number of days minus one from the start of the current era (that is, 0001-01-01). In the above query the input was 2001-09-22, which equals (2001-9)-22, which equals 1970 days.

## Date functions

Sometimes, you need to know how the difference between two timestamps. For this, DB2 provides a built in function called TIMESTAMPDIFF(). The value returned is an approximation, however, because it does not account for leap years and assumes only 30 days per month. Here is an example of how to find the approximate difference in time between two dates:

timestampdiff (<n>, char(

timestamp('2002-11-30-00.00.00')-

timestamp('2002-11-08-00.00.00')))

In place of <n>, use one of the following values to indicate the unit of time for the result:

* 1 = Fractions of a second
* 2 = Seconds
* 4 = Minutes
* 8 = Hours
* 16 = Days
* 32 = Weeks
* 64 = Months
* 128 = Quarters
* 256 = Years

Using timestampdiff() is more accurate when the dates are close together than when they are far apart. If you need a more precise calculation, you can use the following to determine the difference in time (in seconds):

(DAYS(t1) - DAYS(t2)) \* 86400 +

(MIDNIGHT\_SECONDS(t1) - MIDNIGHT\_SECONDS(t2))

For convenience, you can also create an SQL user-defined function of the above:

CREATE FUNCTION secondsdiff(t1 TIMESTAMP, t2 TIMESTAMP)

RETURNS INT

RETURN (

(DAYS(t1) - DAYS(t2)) \* 86400 +

(MIDNIGHT\_SECONDS(t1) - MIDNIGHT\_SECONDS(t2))

)

If you need to determine if a given year is a leap year, here is a useful SQL function you can create to determine the number of days in a given year:

CREATE FUNCTION daysinyear(yr INT)

RETURNS INT

RETURN (CASE (mod(yr, 400)) WHEN 0 THEN 366 ELSE

CASE (mod(yr, 4))WHEN 0 THEN

CASE (mod(yr, 100)) WHEN 0 THEN 365 ELSE 366 END

ELSE 365 END

END)

Finally, here is a chart of built-in functions for date manipulation. The intent is to help you quickly identify a function that might fit your needs, not to provide a full reference. Consult the SQL Reference for more information on these functions.

SQL Date and Time functions are as follows:

1. **DAYNAME**: Returns a mixed case character string containing the name of the day (e.g., Friday) for the day portion of the argument.
2. **DAYOFWEEK**: Returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Sunday.
3. **DAYOFWEEK\_ISO**: Returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Monday.
4. **DAYOFYEAR**: Returns the day of the year in the argument as an integer value in the range 1-366.
5. **DAYS**: Returns an integer representation of a date.
6. **JULIAN\_DAY**: Returns an integer value representing the number of days from January 1, 4712 B.C. (the start of Julian date calendar) to the date value specified in the argument.
7. **MIDNIGHT\_SECONDS**: Returns an integer value in the range 0 to 86 400 representing the number of seconds between midnight and the time value specified in the argument.
8. **MONTHNAME**: Returns a mixed case character string containing the name of month (e.g., January) for the month portion of the argument.
9. **TIMESTAMP\_ISO**: eturns a timestamp value based on date, time or timestamp argument.
10. **TIMESTAMP\_FORMAT**: Returns a timestamp from a character string that has been interpreted using a character template.
11. **TIMESTAMPDIFF**: Returns an estimated number of intervals of the type defined by the first argument, based on the difference between two timestamps.
12. **TO\_CHAR**: Returns a character representation of a timestamp that has been formatted using a character template. TO\_CHAR is a synonym for VARCHAR\_FORMAT.
13. **TO\_DATE**: Returns a timestamp from a character string that has been inter-preted using a character template. TO\_DATE is a synonym for TIMESTAMP\_FORMAT.
14. **WEEK**: Returns the week of the year of the argument as an integer value in range 1-54. The week starts with Sunday.
15. **WEEK\_ISO**: Returns the week of the year of the argument as an integer value in the range 1-53.

## Changing the date format

A common question I get often relates to the presentation of dates. The default format used for dates is determined by the territory code of the database (which can be specified at database creation time). For example, my database was created using territory=US. Therefore the date format looks like the following:

values current date

1

----------

05/30/2003

1 record(s) selected.

That is, the format is MM/DD/YYYY. If you want to change the format, you can bind the collection of db2 utility packages to use a different date format. The formats supported are:

* **DEF**: Use a date and time format associated with the territory code.
* **EUR**: Use the IBM standard for Europe date and time format.
* **ISO**: Use the date and time format of the International Standards Organization.
* **JIS**: Use the date and time format of the Japanese Industrial Standard.
* **LOC**: Use the date and time format in local form associated with the territory code of the database.
* **USA**: Use the IBM standard for U.S. date and time format.

To change the default format to ISO on windows (YYYY-MM-DD), do the following steps:

* On the command line, change your current directory to sqllib\bnd.

For example:   
**On Windows**: c:\program files\IBM\sqllib\bnd  
**On UNIX**:/home/db2inst1/sqllib/bnd

* Connect to the database from the operating system shell as a user with SYSADM authority:

db2 connect to DBNAME

db2 bind @db2ubind.lst datetime ISO blocking all grant public

(In your case, substitute your database name and desired date format for DBNAME and ISO, respectively.)

Now, you can see that the database uses ISO date format:

values current date

1

----------

2003-05-30

1 record(s) selected.

**Custom Date/Time Formatting**

In the last example, we demonstrated how to change the way DB2 presents dates in some localized formats. But what if you wish to have a custom format such as 'yyyymmdd'? The best way to do this is by writing your own custom formatting function.

Here is the UDF:

create function ts\_fmt(TS timestamp, fmt varchar(20))

returns varchar(50)

return

with tmp (dd,mm,yyyy,hh,mi,ss,nnnnnn) as

(

select

substr( digits (day(TS)),9),

substr( digits (month(TS)),9) ,

rtrim(char(year(TS))) ,

substr( digits (hour(TS)),9),

substr( digits (minute(TS)),9),

substr( digits (second(TS)),9),

rtrim(char(microsecond(TS)))

from sysibm.sysdummy1

)

select

case fmt

when 'yyyymmdd'

then yyyy || mm || dd

when 'mm/dd/yyyy'

then mm || '/' || dd || '/' || yyyy

when 'yyyy/dd/mm hh:mi:ss'

then yyyy || '/' || mm || '/' || dd || ' ' ||

hh || ':' || mi || ':' || ss

when 'nnnnnn'

then nnnnnn

else

'date format ' || coalesce(fmt,' <null> ') ||

' not recognized.'

end

from tmp

</null>

The function code may appear complex at first, but upon closer examination, you'll see that it is actually quite simple and elegant. First, we use a common table expression (CTE) to strip apart a timestamp (the first input parameter) into its individual components. From there, we check the format provided (the second input parameter) and reassemble the timestamp using the requested format and parts.

The function is also very flexible. To add another pattern simply append another WHEN clause with the expected format. When an unexpected pattern is encountered, an error message is returned.

Usage examples:

values ts\_fmt(current timestamp,'yyyymmdd')

'20030818'

values ts\_fmt(current timestamp,'asa')

'date format asa not recognized.'

SQL DB2 - programmation

## Présentation

En cobol, chaque ordre doit encapsulé par EXEC et END-EXEC :

EXEC SQL

[ordre SQL

;]

END-EXEC

## SQLCA

SQLCA : SQL Communication Area, zone renseignée par DB2 après chaque ordre SQL

EXEC SQL

INCLUDE SQLCA

END-EXEC.

La compilation traduira en

\*\*\*\*\*EXEC SQL

\*\*\*\*\* INCLUDE SQLCA

\*\*\*\*\*END-EXEC.

01 SQLCA.

05 SQLCAID PIC X(8).

\*\*\* constante valant SQLCA, utilisée comme repère par ceux qui fouillent dans les Dumps

05 SQLCABC PIC S9(9) COMP-4.

\*\*\* longueur de la zone qui vaut X'88’

05 SQLCODE PIC S9(9) COMP-4.

\*\*\*\*\* en binaire. Il faut passer le code SQLCODE souvent négatif dans une zone éditable –(9)9

\*\*\* zéro  : OK

\*\*\* négatif : erreur

\*\*\* positif : warning

\*\*\* +100 : non trouvé

05 SQLERRM.

\*\*\* longueur et valeur du message explicative

49 SQLERRML PIC S9(4) COMP-4.

49 SQLERRMC PIC X(70).

05 SQLERRP PIC X(8).

05 SQLERRD OCCURS 6 TIMES

PIC S9(9) COMP-4.

05 SQLWARN.

\*\*\* indicateur du warning

10 SQLWARN0 PIC X.

\*\*\* W => au moins 1 warning

10 SQLWARN1 PIC X.

\*\*\* valeur tronquée

10 SQLWARN2 PIC X.

\*\*\* elimination de NULL pour function

10 SQLWARN3 PIC X.

\*\*\* nombre de colonnes > nombre de host-variables

10 SQLWARN4 PIC X.

\*\*\* concerne les SQL dynamiques

10 SQLWARN5 PIC X.

\*\*\* ordre SQL de SQL/DS non valide en DB2

10 SQLWARN6 PIC X.

\*\*\* reserve pour le future

10 SQLWARN7 PIC X.

\*\*\* reserve pour le futur

05 SQLEXT.

\*\*\* reserve pour le futur

10 SQLWARN8 PIC X.

10 SQLWARN9 PIC X.

10 SQLWARNA PIC X.

10 SQLSTATE PIC X(5).

## La description des tables et des vues

Pour les description des tables et des vues, le plus simple est d’utiliser l’outil

DB2I PRIMARY OPTION MENU

Par exemple, pour la table TEST :

2 DCLGEN (Generate SQL and source language declarations)

Avec

* 1 SOURCE TABLE NAME ===> test
* 4 DATA SET NAME ... ===> 'MATA01.DCLGEN.COBOL(TEST)'

Dans le programme cobol, il faut écrire les lignes

EXEC SQL

INCLUDE TEST

END-EXEC.

La compilation les transformera en :

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* COBOL DECLARATION FOR TABLE TEST\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

01 DCLTEST.

10 TESNO PIC S9(9) USAGE COMP.

10 TESLIBPIC X(20).

10 TESCODPIC X(02).

10 TESDATPIC X(10).

10 TESNB PIC S9(5)V99 COMP-3.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 5 \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Les variables hôtes (host variables)

Une variable hôte est une variable définie dans le programme et utilisée dans les ordres SQL

* Dans le résultat d’une sélection SELECT … INTO  :VARIABLE
* Dans la mise à jour INSERT …VALUE( :VARIABLE)
* Dans les critères de sélection … WHERE COLONNE = :VARIABLE
* Dans des expressions arithmétiques

Elle est précédée de « : » sans blanc à l’intérieur deEXEC SQL … END-EXEC.

Elle est de type compatible avec la colonne

Elle est de dernier niveau (de 01 à 48 ou 77) ou c’est une zone de groupe à 2 niveaux.

Elle peut être préfixée par une variable de niveau supérieur

Exemple :

MOVE '2015-01-02' TO TESDAT.

EXEC SQL

UPDATE TEST

SET TESDAT = :TESDAT

END-EXEC.

Exemple :

EXEC SQL

SELECT TESTDAT

INTO = :TESDAT

FROM TEST

WHERE TESTNO = :WS-TESTNO

END-EXEC.

## Les colonnes de longueur variables

Pour des colonnes de longueur variable, la variable doit regrouper deux variables de niveau 49, la première pour la longueur, la deuxième pour la valeur avec la longueur maximum

05 FOURNI.

49 FOURNILG PIC S9(4) COMP.

49 FOURNIVAL PIC X(25).

EXEC SQL

SELECT FOURNI

INTO = :FOURNI

FROM FLAQ

WHERE FLAQNO = :WS-FLQNO

END-EXEC.

Règle d’écriture :

01 WS-FOURNI-LIGNE.

05 WS-FLQNO PIC S9(3) COMP-3.

05 WS-LIBELLE PIC X(20).

Les 2 ordres suivants sont équivalents :

EXEC SQL

SELECT \* FROM FLAQ

INTO :WS-FOURNI-LIGNE

WHERE FLAQNO = :WS-FLQNO

END-EXEC.

EXEC SQL

SELECT WS-FLQNO , WS-LIBELLE

FROM FLAQ

INTO :WS-FLQNO , :WS-LIBELLE

WHERE FLAQNO = :WS-FLQNO

END-EXEC.

## Les variables indicateurs

Pour les colonnes qui peuvent être NULL , il faut associer à la variable hote une variable indicateur, d’un demi-mot binaire, qui a :

* une valeur négative si la colonne est NULL
* une valeur à zéro si la conne est renseignée
* une valeur positive si la colonne est renseignée mais que le résultat est tronqué

Les variables indicateurs ne peuvent être utilisées dans une clause WHERE. On utilisera alors la COLONNE is NULL

Exemple :

05 WS-CODERESP PIC X(4).

…

05 WS-CODERESP-I PIC S9(4) COMP.

EXEC SQL

SELECT CODERESP

INTO : WS-CODERESP:WS-CODERESP-I

FROM MAISON

WHERE SIREN = :WS-SIREN

END-EXEC.

IF WS-CODERESP-I < ZERO

DISPLAY “PAS DE RESPONSABLE POUR SIREN “ WS-SIREN.

## Mise à NULL d’une colonne

Une colonne est mise à NULL en positionnant son indicateur à une valeur négative.

EXEC SQL

UPDATE MAISON

SET WS-CODERESP-I = -1

WHERE SIREN = :WS-SIREN

END-EXEC.

## L’ordre DECLARE TABLE

La déclaration des tables en working permet au précompilateur de vérifier la cohérence entre l’ordre SQL et les tables de cet ordre SQL.

EXEC SQL

INCLUDE TEST

END-EXEC.

La compilation traduit ce lignes en

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* DCLGEN TABLE(TEST)\*

\* LIBRARY(MATA01.DCLGEN.COBOL(TEST)) \*

\* ACTION(REPLACE) \*

\* LANGUAGE(COBOL) \*

\* QUOTE\*

\* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS\*

\* ( TESNO INTEGER NOT NULL ,

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

EXEC SQL DECLARE TEST TABLE

( TESNO INTEGER ,

TESLIB CHAR(20),

TESCOD CHAR(02),

TESDAT DATE ,

TESNB DECIMAL (5,2)

) END-EXEC.

\* PRIMAREY KEY (TESNO)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* COBOL DECLARATION FOR TABLE TEST \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

01 DCLTEST.

10 TESNO PIC S9(9) USAGE COMP.

10 TESLIBPIC X(20).

10 TESCODPIC X(02).

10 TESDATPIC X(10).

10 TESNB PIC S9(5)V99 COMP-3 .

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 5 \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Lire une ligne

Si la requête ne ramène qu’une seule ligne, on peut utiliser l’ordre SELECT

EXEC SQL

SELECT

TESNO, TESLIB , TESCOD, TESDAT, TESNB

INTO

: WS-TESNO, : WS-TESLIB , : WS-TESCOD, : WS-TESDAT, : WS-TESNB

WHERE TESTNO = :WS-TESTNO

FROM TEST

END-EXEC.

SQLCOE = ZERO OK, la ligne a été trouvée, les host variables sont alimentées

SQLCOE = +100 aucune ligne n’a été trouvée

SQLCOE = -811 KO, plusieurs lignes ont été trouvées

## Lire plusieurs lignes

En cobol pour lire plusieurs lignes, on utilise un curseur, qu’on déclare, qu’on ouvre, qu’on lit (FETCH), et qu’on ferme.

La déclaration doit être faite avant toute utilisation.

EXEC SQL

**DECLARE** CURSEUR-TEST **CURSOR FOR**

**SELECT** TESNO, TESLIB

, TESCOD, TESDAT, TESNB

**FROM** TEST

END-EXEC.

EXEC SQL

**OPEN** CURSEUR-TEST

END-EXEC.

EXEC SQL

**FETCH** CURSEUR-TEST **INTO**

:TESNO, :TESLIB , :TESCOD

, :TESDAT, :TESNB

END-EXEC.

EXEC SQL

**CLOSE** CURSEUR-TEST

**END-EXEC.**

Le FETCH est séquentiel, sans retour arrière, sans positionnement sur une nième ligne.

Attention : tous les curseurs sont fermés lors d’un COMMIT

## UPDATE avec un curseur

Si on doit modifier une ligne qu’on lit avec un curseur, avec UPDATE ou DELETE, on ajoute FOR UPDATE à la déclaration du curseur.

Le nom du curseur est unique pour un programme donné.

La clause INTO est absente du SELECT de la déclaration du curseur, mais est présente dans le FETCH.

En cas de mise à jour, le SELECT du DECLARE CURSOR ne peut pas :

* Référencer plus d’une table
* Référencer une fonction colonne ( MIN, MAX, …)
* Utiliser les clauses DISTINCT, UNION, GROUP BY, ORDER BY

EXEC SQL

**DECLAR**E CURSEUR-TEST **CURSOR FOR**

**SELECT** TESNO, TESLIB

, TESCOD, TESDAT, TESNB

**FROM** TEST

**FOR UPDATE**

END-EXEC.

Ou FOR UPDATE colonne\_1, colonne\_2, etc

EXEC SQL

**FETCH** CURSEUR-TEST **INTO**

:TESNO, :TESLIB , :TESCOD, :TESDAT, :TESNB

END-EXEC.

EXEC SQL

UPDATE TEST

**SET** TESLIB = :TESTLIB, colonne\_2, etc

**WHERE CURRENT OF** CURSEUR-TEST

END-EXEC.

EXEC SQL

**DELETE**

**FROM** TEST

**WHERE CURRENT OF** CURSEUR-TEST

END-EXEC.

## L’ordre WHENEVER

EXEC SQL

**WHENEVER SQLERROR**

**GO TO** FIN-ERREUR

**END-EXEC.**

L’ordre WHENEVER a trois paramètres possibles

1. WHENEVER NOT FOUND
2. WHENEVER SQLERROR
3. WHENEVER SQLWARNING

## DCLGEN

Le générateur de déclaration DCLGEN génère dans un membre de bibliothèque un ordre SQL DECLARE … TABLE (description colonne\_1, description colonne\_2, …) END-EXEC avec les host variables.

En working, on l’appelle par

**EXEC** **SQL**

**INCLUDE** nom\_de\_la-table

**END-EXEC**.

L’appel au générateur se fait via :

* Le panel de DB2I
* La sous-commande DCLGEN du processeur de commande DSN

Attention, les variables indicateurs ne sont pas générées.

## Précompilation

La précompilation transforme le source en source étendu en ajoutant les spécificités DB2 et enregistre en working un TIMESTAMP

Dans le cas de CICS, il y a une étape supplémentaire.

SOURCE => PRECOMPIL

1 SOURCE ETENDU => COMPIL LINK => MODULE EXECUTABLE

2 DBRM => BIND => PLAN

MODULE EXECUTABLE

+ PLAN

+ TABLES DB2

=> EXECUTION

## Le BIND

Le BIND construit le plan :

* Il valide les ordres SQL contenus dans le DBRM via le catalogue DB2
* Il vérifie les autorisations
* Il sélectionne les chemins optimaux
* Il met à jour le catalogue avec les caractéristiques du plan et de ses DBRM
* Il crée une structure de contrôle, le plan d’application « load module » stocké dans le directory d DB2

Le BIND peut être exécuté

* Via un panel de DB2I
* via la sous-commande BIND du processeur de commande DSN , en batch

Paramètre ISOLATION :

* c’est le niveau de verrouillage des pages en lecture pour le programme associé
* - CS, Cursor Stability, libère la page verrouillée au changement de page
* - RR, Repeatable Read, libère la page verrouillée au point de synchronisation

Paramètres CREATOR et OWNER :

BIND PLAN(ce\_plan) … OWNER (proprio)

Proprio est le propriétaire du plan

* C’est le CREATOR dans le catalogue, enregistré comme BOUNDBY
* Il doit avoir les privilège de BINDAA (si ajout) ou BIND sur ce\_plan (si REPLACE)
* Il doit avoir les autorisations pour exécuter les ordres SQL du plan
* Il devient le qualifieur des tables et des vues référencées

Dans le cas d’un programme principal qui appelle des sous-programmes, il faut spécifier tous ceux qui ont des ordres SQL :

BIND PLAN(programme\_principal)

MEMBER (programme\_principal, sous\_programme1, sous\_programme2)

Si le programme\_principal n’a pas d’ordre SQL,

BIND PLAN(programme\_principal)

MEMBER ( sous\_programme1, sous\_programme2)

Le contrôle des timestamps des DBRM et du timestamps du load module signale la désynchronisation par SQLCODE = -818

## REBIND, BIND REPLACE et FREE

La modification d’un plan est exécutée par

* REBIND en cas de
  + changement d’autorisation
  + création et de suppression d’index
  + mise à jour des statistiques utilisées par l’optimiseur (RUNSTATS, …)
  + mise à jour des paramètres du plan
* BIND REPLACE dans tous les cas,
  + Mais obligatoire en cas de recompilation de DBRM

REBIND PLAN(le-plan)

FREE détruit le plan

FREE PLAN(le\_plan)

FREE PLAN(\*) : pour détruire tous les plans dont l’utilisateur a le privilège de BIND, y compris ceux l’autorisation est « PUBLIC »

Mise à jour des plans du catalogue :

* 1 ligne par plan dans SYSPLAN
* N lignes par plan dans SYSPLANDEP
* 1 ligne par DBRM d’un plan dans SYSDBRM
* N lignes par ordres SQL de chaque DBRM dans SYSSTMT

# Mécanisme de verrouillage

C’est le composant de DB2 « IRLM » qui assure l’intégralité des données :

* En gérant la simultanéité d’accès aux données
* En assurant la protection

An niveau d’une transaction, c’est le principe du tout ou rien.

Les points de cohérence sont :

* Implicites (fin de transaction)
* Explicites avec soit le COMMIT soit le ROOLBACK

In CICS Transactions, CICS, not DB2, is the Transaction Manager   
If you issue a EXEC CICS SYNCPOINT, DB2 gets committed as well.   
If you issue a EXEC SQL COMMIT, only DB2 stuff gets committed. Other resources - files, TSQs etc remain 'uncommitted'.

Le verrou permet d’assurer

* La cohérence de l’information
* L’intégrité des données

Il a 4 caractéristiques :

* La portée : le tablespace ou l’indexspace
* La taille : en page, table, tablespace ou en sous-page, indexspace
* Le mode : partageable ou exclusif
* La durée

### Le DEADLOCK

Le DEADLOCK est la situation de blocage mutuel.

DB2 provoque un retour en arrière, souvent, de celle qui a le moins fait de mises à jour avec un code retour -911 ou -913

Quand une transaction met à jour des données, DB2 pose un lock exclusif (X) pour interdire l’accès aux autres transactions

Quand une transaction lit des données, DB2 pose un lock partageable (S) qui autorise d’autres lectures mais interdit l’accès aux transactions de mise à jour

### Taille du verrou

La taille du verrou est précisée par le paramètre LOCKSIZE du CREATE TABLESPACE :

* TABLESPACE  : verrou global sur tablespace, sans verrou plus bas
* TABLE : verrou sur table, sans verrou plus bas, sans verrou individuel
* PAGE : verrou individuels posé sur les pages
* ANY : taille initiale du verrou choisie par DB2

PAGE : meilleurs concurrence et débits, mais augmentation de la CPU

TABLESPACE : moindres CPU et espace mémoire mais forte contention

### Durée de verrouillage

La durée du verrou est précisée lors du BIND

Verrou sur TABLESPACE ou Table

* ACQUIRE=ALLOCATE : verrouillage effectué à l’allocation du plan, de la manière ka plus exclusive, avant le démarrage, avec tous les verrous possibles
* ACQUIRE=USE : verrouillage au fur et à mesure de l’exécution du programme

Libération du verrou :

* RELEASE=COMMIT : libération de la ressource au prochain point de cohérence
* RELEASE=DEALLOCATE : libération à la désallocation du plan

La combinaison ALLOCATE-COMMIT est invalide

Verrou sur page

Pose du verrou : au 1er accès

* Libération de la ressource
  + au prochain point de cohérence si mise à jour de la page
  + en fonction de ISOLATION du BIND si lecture de la table
    - CS, Cursor Stability, verrou levé au prochain accès sur une autre page de tablespace
    - RR, Repeatable Read, verrou levé au prochain COMMIT

### Mode de verrouillage

Verrou individuel sur page : 3 modes

* S lecture pour tous
* U lecture avec mise à jour du propriétaire. Les demandes de mode S sont acceptées
* X exclusif par le propriétaire

Verrou sur table ou tablespace :63 modes

* IS lecture pour le propriétaire. Mises à jour acceptés pour les autres
* S lecture pour tous
* IX : lecture et mise à jour individuelles pour propriétaires et users concurrents
* U lecture avec mise à jour du propriétaire. Les demandes de mode S sont acceptées
* X exclusif par le propriétaire

### L’ordre LOCK TABLE

En début de programme, on pose un verrou avec la commande

LOCK TABLE nom\_de\_la\_table IN mode MODE

Avec mode = SHARE ou EXCLUSIVE

La libération dépend du paramètre RELEASE du BIND.

Limite : sur les tables locales seulement.

Uniquement pour les applications sensibles, dans un but de performance et de priorité.

### Verrouillage implicite

Un verrouillage implicite est posé lors des ordres SQL de définition et de contrôle des données et lors de l’opération de BIND.

Un verrouillage sur les tables du directory SKCT, la table des DBD et celles des autorisations lors de l’exécution d’un plan.

## Ordres SQL

CREATE TABLE **PLAN\_TABLE** ;

## CREATE TABLE

**CREATE** **TABLE** ALBUM(

CODEA CHAR(4) NOT NULL,

ALTITREVARCHAR(40) NOT NULL ,

STATUS CHAR (2) NOT NULL BY DEFAULT

PRIMARY KEY (CODEA)

)

**IN** DBMATE1.TSMATA01;

CREATE UNIQUE INDEX PK\_ALBUM

ON ALBUM (CODEA);

**CREATE** **TABLE** CHANSON (

CODEC CHAR(4)NOT NULL ,

CHTITREVARCHAR(50) NOT NULL ,

CODEA CHAR(4),

**PRIMARY KEY** (CODEC) ,

**FOREIGN KEY** CHAALB (CODEA)

**REFERENCES** ALBUM

**ON DELETE RESTRICT**

)

**IN** DBMATE1.TSMATA01;

**CREATE** **UNIQUE** **INDEX** PK\_CHANSON

**ON** CHANSON(CODEC);

**CREATE** **TABLE** MAISON (

SIREN CHAR(5)NOT NULL ,

RAISON VARCHAR(35) NOT NULL ,

ADRESSE1 VARCHAR(35) ,

ADRESSE2 VARCHAR(35) ,

CODPOST CHAR(6),

CODERESP CHAR(4),

VILLE CHAR(30) ,

TYPECHAR(1) NOT NULL ,

**PRIMARY KEY** (SIREN) ,

**FOREIGN KEY** MAISPERS (CODERESP)

**REFERENCES** PERSONNE

**ON DELETE SET NULL**

)

**IN** DBMATE1.TSMATA01;

CREATE UNIQUE INDEX PK\_MAISON ON MAISON (SIREN) ;

CODEA CHAR(4)NOT NULL ,

DATEVENTE DATE NOT NULL ,

SIREN CHAR(5)NOT NULL ,

QTE SMALLINT ,

PRIX DEC (5,2) NOT NULL ,

PRIMARY KEY (CODEA, DATEVENTE, SIREN )

, FOREIGN KEY VENTMAIS (SIREN)

REFERENCES MAISON

ON DELETE RESTRICT ,

FOREIGN KEY VENTALB(CODEA)

REFERENCES ALBUM

ON DELETE RESTRICT

)

IN DBMATE1.TSMATA01;

CREATE UNIQUE INDEX PK\_VENTES

ON VENTES (CODEA, DATEVENTE, SIREN )

;

CREATE TABLE CONTRAT (

CODEA CHAR(4)NOT NULL ,

SIREN CHAR(5)NOT NULL ,

CODEP CHAR(4)NOT NULL ,

DROITS DEC(5,2) NOT NULL ,

DATESIGN DATENOT NULL WITH DEFAULT ,

PRIMARY KEY (CODEA, SIREN, CODEP) ,

FOREIGN KEY CONTMAIS (SIREN)

REFERENCES MAISON

ON DELETE RESTRICT ,

FOREIGN KEY CONTALB(CODEA)

REFERENCES ALBUM

ON DELETE RESTRICT ,

FOREIGN KEY CONTPERS (CODEP)

REFERENCES PERSONNE

ON DELETE RESTRICT

)

IN DBMATE1.TSMATA01;

CREATE UNIQUE INDEX PK\_CONTRAT

ON CONTRAT(CODEA, SIREN, CODEP )

;

## INSERT

INSERT INTO ALBUM VALUES ('C001' , 'ALBUM 1') ;

INSERT INTO ALBUM VALUES ('C002' , 'ALBUM 2') ;

INSERT INTO ALBUM VALUES ('C003' , 'ALBUM 3') ;

INSERT INTO ALBUM VALUES ('C004' , 'ALBUM 4') ;

INSERT INTO MATA01.MAISON ;

EXEC SQL INSERT INTO STAFF

(ID, NAME, DEPT, JOB, YEARS, SALARY, COMM)

VALUES

(:ID-X, :NAME-X:NULL-NAME, :DEPT:NULL-DEPT,

:JOB:NULL-JOB,:YEARS:NULL-YEARS,

:SALARY:NULL-SALARY,:COMM:NULL-COMM)

END-EXEC

## SELECT

**SELECT** \* **FROM** SYSIBM.SYSTABLES

**WHERE** CREATOR = 'MATA01' ;

**SELECT** **SUBSTR**(C.TBCREATOR, 1 , 10 )

, **SUBSTR**(C.TBNAME, 1 , 10 )

, T.TYPE

, C.\*

**FROM** SYSIBM.SYSCOLUMNS C, SYSIBM.SYSTABLES T

**WHERE**

C.NAME = 'ADRESSE2'

AND C.**TBNAME** = T.**NAME**

AND C.**TBCREATOR** = T.**CREATOR**

**ORDER** BY 1, 2

;

**SELECT** **SUBSTR**(CODEC, 1, 4)

, **SUBSTR**(TITRE, 1, 15)

, **SUBSTR**(CODEA, 1, 4)

TITRECODEA

**WHERE** CODEA < '0003' **AND**

CODEA **NOT IN**

(**SELECT** CODEA

**FROM** MATA01.ALBUM)

;

**SELECT** V.CODEA

**FROM** MATA04.VENTES V

, MATA04.ALBUMA

**WHERE** A.CODEA = V.CODEA

**AND** TITRE LIKE 'P**%**''

**GROUP BY**  V.CODEA

;

**SELECT** **SUBSTR**(PLNAME , 1 , 10 )

, **SUBSTR**(PLCREATOR , 1 , 10 )

, **SUBSTR**(NAME , 1 , 10 )

, PDSNAME

, PRECOMPDATE, QUOTE, COMMA, HOSTLANG

, IBMREQD , CHARSET, MIXED

**FROM** **SYSIBM.SYSDBRM**

**WHERE**

PLCREATOR **LIKE** 'MATA0**%**'

**ORDER BY** 2, 1 , 3

;

SELECT \*

FROM SYSIBM.SYSINDEXES I

WHERE CREATOR LIKE 'MATA0%'

AND NOT EXISTS

(SELECT \*

FROM SYSIBM.SYSPLANDEP

WHERE BNAME = I.NAME

AND BTYPE = 'I'

)

ORDER BY CREATOR, TBNAME, NAME

;

SELECT SUBSTR(DBNAME , 1 , 10 )

, SUBSTR(TSNAME , 1 , 10 )

, CARD

, NEARINDREF

, FARINDREF

FROM SYSIBM.SYSTABLEPART

WHERE

TSNAME LIKE 'TSMATA0%'

;

SELECT SUBSTR(BCREATOR, 1 , 10 )

, SUBSTR(DNAME, 1 , 10 )

, SUBSTR(BNAME, 1 , 10 )

, BTYPE

FROM SYSIBM.SYSPLANDEP

WHERE

BCREATOR LIKE 'MATA0%'

AND BTYPE IN('T', 'V')

ORDER BY 1 , 2, 3 , 4

;

SELECT SUBSTR(C.TBCREATOR, 1 , 10 )

, SUBSTR(C.TBNAME, 1 , 10 )

, T.TYPE

FROM SYSIBM.SYSCOLUMNS C

, SYSIBM.SYSTABLES T

WHERE

C.NAME = 'CODEP'

AND C.TBNAME = T.NAME

AND C.TBCREATOR = T.CREATOR

ORDER BY 1 , 2

;

SELECT DISTINCT TCREATOR, TTNAME

FROM SYSIBM.SYSTABAUTH

WHERE

GRANTEE = 'PUBLIC'

OR GRANTEE LIKE 'MATA01%'

ORDER BY 1 , 2

;

-- PLCREATORLIKE 'MATA0%'

-- GRANTOR LIKE 'MATA0%'

EXPLAIN PLAN FOR

SELECT \*

FROM MATA01.MAISON

;

SELECT SUBSTR(IXNAME , 1 , 10 )

,SUBSTR(IXCREATOR , 1 , 10 )

,SUBSTR(COLNAME , 1 , 20 )

, COLNO , COLSEQ, ORDERING, IBMREQD

FROM SYSIBM.?

SELECT \*

FROM SYSIBM.SYSKEYS

WHERE

IXCREATORLIKE 'MATA01%'

ORDER BY 1, COLNO

;

SELECT\*

FROM SYSIBM.SYSTABLES

WHERE NAME LIKE 'ALB%'

OR NAME LIKE 'CHA%'

ORDER BY NAME , CREATOR

;

SELECT\*

FROM SYSIBM.SYSVIEWS

WHERE

CREATOR LIKE 'MATA01%'

OR NAME LIKE 'CHA%'

ORDER BY NAME , CREATOR

;

SELECT\*

FROM SYSIBM.SYSINDEXES

WHERE

CREATOR LIKE 'MATA01%'

WHERE NAME LIKE 'ALB%'

OR NAME LIKE 'CHA%'

ORDER BY NAME , CREATOR

;

SELECT SUBSTR(CREATOR , 1 , 10 )

, SUBSTR(TBNAME , 1 , 10 )

, SUBSTR(RELNAME , 1 , 10 )

, SUBSTR(COLNAME , 1 , 10 )

, COLNO, COLSEQ ,IBMREQD

FROM SYSIBM.SYSFOREIGNKEYS

WHERE

CREATOR LIKE 'MATA01%'

OR NAME LIKE 'CHA%'

ORDER BY CREATOR , TBNAME, RELNAME, COLNO

;

# Normes et Contrôles spécifiques

### Nommage

Pour la plupart des systèmes de gestion de bases de données relationnelles utilisant SQL, comme MySQL, PostgreSQL, et SQL Server, voici quelques règles courantes concernant les noms des objets de base de données (tables, colonnes, etc.) :

1. **Caractères valides** : Les noms peuvent contenir des lettres (majuscules ou minuscules), des chiffres et le caractère de soulignement (\_).
2. **Premier caractère** : Le premier caractère doit être une lettre. Cela signifie qu'il ne peut pas commencer par un chiffre ou un caractère de soulignement.
3. **Longueur** : La longueur maximale des noms peut varier selon le système de gestion de base de données, mais elle est généralement suffisamment grande pour couvrir la plupart des besoins pratiques.
4. **Sensibilité à la casse** : Certains systèmes de gestion de base de données sont sensibles à la casse pour les noms d'objets, tandis que d'autres ne le sont pas. Par exemple, MySQL est sensible à la casse sous Unix mais pas sous Windows, tandis que PostgreSQL convertit les noms non cités en minuscules.
5. **Mots réservés** : Les noms ne doivent pas être des mots réservés par SQL, comme SELECT, INSERT, TABLE, etc., à moins qu'ils ne soient correctement échappés.

Il est toujours bon de consulter la documentation spécifique de votre système de gestion de base de données pour connaître les règles exactes et les bonnes pratiques.

### L’email

Dans la création d'une base de données, il n'existe pas de propriété native ou de type de données spécifique qui garantit qu'un champ contient une adresse e-mail valide, c'est-à-dire avec exactement un caractère arobase (@). Cependant, on peut mettre en place des validations pour s’assurer que les données insérées dans un champ respectent le format d'une adresse e-mail.

Voici quelques approches courantes pour valider les adresses e-mail dans une base de données :

1. **Contraintes de validation** : Vous pouvez utiliser des contraintes CHECK dans SQL pour valider le format des données. Par exemple, en SQL, vous pouvez créer une contrainte qui vérifie la présence d'un seul caractère @.
2. **Validation au niveau de l'application** : Avant d'insérer des données dans la base de données, vous pouvez utiliser des expressions régulières dans votre code applicatif pour valider le format de l'adresse e-mail.
3. **Types de données personnalisés** : Certains systèmes de gestion de base de données permettent de créer des types de données personnalisés avec des règles de validation spécifiques.
4. **Triggers** : Vous pouvez utiliser des triggers pour valider les données avant qu'elles ne soient insérées ou mises à jour dans la base de données.

Voici un exemple de contrainte CHECK en SQL pour s'assurer qu'un champ contient exactement un caractère @ :

CREATE TABLE utilisateurs (

id SERIAL PRIMARY KEY,

email VARCHAR(255) CHECK (email LIKE '%\_@\_%.\_%' AND email NOT LIKE '%@%@%' ) );

Cette contrainte vérifie que l'adresse e-mail contient un seul caractère @ et qu'il est entouré de caractères valides. Cependant, cette méthode a ses limites et ne couvre pas tous les cas de validation possibles pour une adresse e-mail. Une validation plus robuste est généralement effectuée au niveau de l'application.

# Mots réservés

La liste de ces mots réservés ne doivent pas être affectés à des noms de bases de données, de tables, de colonnes ou de tout autre objet de base de données. Certains SGBD réservent d'autres mots.

**A**BORT ADMIN AGGREGATE ALIGN ALL ALLOCATE ANALYSE ANALYZE AND ANY AS ASC

**B**ETWEEN BINARY BIT BOTH

**C**ASE CAST CHAR CHARACTER CHECK CLUSTER COALESCE COLLATE COLLATION COLUMN CONSTRAINT COPY CROSS CURRENT CURRENT\_CATALOG CURRENT\_DATE CURRENT\_DB CURRENT\_SCHEMA CURRENT\_SID

CURRENT\_TIME CURRENT\_TIMESTAMP CURRENT\_USER CURRENT\_USERID CURRENT\_USEROID

**D**EALLOCATE DEC DECIMAL DECODE DEFAULT DEFERRABLE DESC DISTINCT DISTRIBUTE DO

**E**LSE END EXCEPT EXCLUDE EXISTS EXPLAIN EXPRESS EXTEND EXTERNAL EXTRACT

**F**ALSE FIRST FLOAT FOLLOWING FOR FOREIGN FROM FULL FUNCTION

**H**AVING

**G**ENSTATS GLOBAL GROUP

**I**DENTIFIER\_CASE ILIKE IN INDEX INITIALLY INNER INOUT INTERSECT INTERVAL INTO

**L**EADING LEFT LIKE LIMIT LISTEN LIMIT LOAD LOCAL LOCK

**M**ATERIALIZED MINUS MOVE

**N**ATURAL NCHAR NEW NOT NOTNULL NULL NULLIF NULLS NUMERIC NVL NVL2

**O**FF OFFSET OLD ON ONLINE ONLY OR ORDER OTHERS OUT OUTER OVER OVERLAPS

**P**ARTITION POSITION PRECEDING PRECISION PRESERVE PRIMARY

**R**ESET REUSE RIGHT ROWS ROWSET RULE REUSE

**S**EARCH SELECT SEQUENCE SESSION\_USER SETOF SHOW SOME SUBSTRING SYSTEM

**T**ABLE THEN TIES TIME TIMESTAMP TO TRAILING TRANSACTION TRIGGER TRIM TRUE

**U**NBOUNDED UNION UNIQUE USER USING

**V**ACUUM VARCHAR VERBOSE VERSION VIEW

**W**HEN WHERE WITH WRITE

# SGBD

Un **Système de Gestion de Base de Données** (SGBD) est un logiciel qui permet de stocker des informations dans une base de données. Un tel système permet de lire, écrire, modifier, trier, transformer ou même imprimer les données qui sont contenus dans la base de données.

Parmi les logiciels les plus connus il est possible de citer : MySQL, PostgreSQL, SQLite, Oracle Database, Microsoft SQL Server, Firebird ou Ingres.

Ces systèmes peuvent être catégorisés selon leur fonctionnement :

* **Système propriétaire :** Oracle Database, Microsoft SQLServer, DB2, MaxDB, 4D, dBase, Informix, Sybase
* **Système libre** MySQL, [PostgreSQL](http://sql.sh/sgbd/postgresql), MariaDB, Firebird, Ingres, HSQLDB, Derby
* **Orienté objet :** ZODB, db4o
* **Embarqué :** SQLite, Berkeley DB
* **NoSQL :** Cassandra, Redis, MongoDB, SimpleDB, BigTable, CouchDB, HBase, LevelDB, RethinkDB, Memcached
* **Autre système :** Access, OpenOffice.org Base, FileMaker, HyperFileSQL, Paradox, Neo4j

# Exercices

## Créer des tables mères et des tables filles

### Table mère : FAM

### Table mère : MAR

### Table mère : ART

### Table DEC

### Insertions de lignes dans la table FAM

### Insertion dans la table MAR

### Insertion dans la table ART

### Mise à jour, UPDATE de la table ART

### Suppression, DELETE dans la table MAR

### Affichage, SELECT

==================================================

-- AFFICHER LA TABLE ART SUR LE CODE MARCOD

==================================================

-- AFFICHER LA TABLE ART AVEC LE LIBELLE DE LA MARQUE

==================================================

--

-- PRODUIT CARTESIEN suite à un oubli de jointure

-- sur les colonnes

-- Le nombre de lignes affichés sera égal

-- au nombre de lignes d'une table

-- **multiplié**

-- par le nombre de lignes de l'autre table

--

==================================================

Quels articles ont un libellé égal à une valeur donnée :

==================================================

Quels articles oint un libellé différent d'une valeur donnée :

==================================================

Afficher une ligne qui a une colonne NULL

==================================================

Afficher une ligne qui a une colonne non NULL

==================================================

Jointure

==================================================

Calcul par SQL

--

-- SELECT QUANTITE \* PRIX .....

--

==================================================

Mettre à jour une colonne endoublant sa valeur

==================================================

Afficher le produit de deux colonnes

--

--

==================================================

Calculer les sommes et les produits

--

==================================================

Afficher la date et le TIMESTAMP sur une seule ligne

--

SELECT CURRENT DATE, CURRENT TIMESTAMP

FROM SYSIBM.SYSDUMMY1 ;

==================================================

Afficher les marques absentes de la table article

==================================================

Table des matières

[Présentation 2](#__RefHeading___Toc424800907)

[La conception 5](#__RefHeading___Toc4248009071)

[Formes normales 5](#__RefHeading___Toc4248009081)

[1FN – Première forme normale 6](#__RefHeading___Toc424800911)

[2FN – Deuxième forme normale 8](#__RefHeading___Toc424800912)

[3FN – Troisième forme normale 10](#__RefHeading___Toc424800913)

[FNBC – Forme normale de Boyce-Codd 11](#__RefHeading___Toc424800914)

[4FN - quatrième forme normale 12](#__RefHeading___Toc424800915)

[5FN - cinquième forme normale 12](#__RefHeading___Toc424800916)

[FNDC - forme normale domaine clé 12](#__RefHeading___Toc424800917)

[Gestion du contenant 15](#__RefHeading___Toc625_3336632364)

[Commandes SQL 16](#__RefHeading___Toc42480092011)

[Commande CREATE 17](#__RefHeading___Toc424801049)

[Commande CREATE DATABASE 17](#__RefHeading___Toc4248010491)

[Commande CREATE TABLE 18](#__RefHeading___Toc424801056)

[Commande CREATE INDEX 20](#__RefHeading___Toc424801128)

[Analogie pour comprendre les index en SQL 20](#__RefHeading___Toc424801126)

[Index en SQL 20](#__RefHeading___Toc424801127)

[Créer un index ordinaire 21](#__RefHeading___Toc424801130)

[Créer un index unique 21](#__RefHeading___Toc424801131)

[Convention de nommage 22](#__RefHeading___Toc424801132)

[Commande CREATE VIEW 23](#__RefHeading___Toc429581228)

[Commande DROP 25](#__RefHeading___Toc424801053)

[Commande DROP DATABASE 25](#__RefHeading___Toc4248010531)

[Commande DROP TABLE 25](#__RefHeading___Toc424801071)

[Commande GRANT 27](#__RefHeading___Toc429581230)

[Commande Revoke 27](#__RefHeading___Toc21071_1074248952)

[Commande ALTER TABLE 28](#__RefHeading___Toc424801059)

[Format des colonnes 31](#__RefHeading___Toc424800920)

[Gestion du contenu 36](#__RefHeading___Toc627_3336632364)

[Commandes SQL 36](#__RefHeading___Toc4248009201)

[Commande INSERT INTO 37](#__RefHeading___Toc424801019)

[Insertion d’une ligne à la fois 37](#__RefHeading___Toc424801020)

[Insérer une ligne en spécifiant toutes les colonnes 37](#__RefHeading___Toc424801021)

[Insérer une ligne en spécifiant seulement les colonnes souhaitées 38](#__RefHeading___Toc424801022)

[Insertion de plusieurs lignes à la fois 38](#__RefHeading___Toc424801023)

[Instruction ON DUPLICATE KEY UPDATE 38](#__RefHeading___Toc424801024)

[Insérer une ligne ou ne rien faire 41](#__RefHeading___Toc424801027)

[Compatibilité 41](#__RefHeading___Toc424801028)

[Commande UPDATE 42](#__RefHeading___Toc683_3311445400)

[Modifier toutes les lignes 43](#__RefHeading___Toc424801033)

[Commandes DELETE 44](#__RefHeading___Toc424801034)

[Supprimer toutes les données 44](#__RefHeading___Toc424801040)

[Commande MERGE 48](#__RefHeading___Toc424801042)

[Compatibilité 49](#__RefHeading___Toc424801044)

[Commande TRUNCATE TABLE 50](#__RefHeading___Toc424801045)

[Différence entre TRUNCATE et DELETE 50](#__RefHeading___Toc424801048)

[Commande COMMIT 52](#__RefHeading___Toc429581226)

[Commande ROLLBACK 52](#__RefHeading___Toc4295812261)

[Commande SELECT 53](#__RefHeading___Toc424800921)

[Commande pour une table 53](#__RefHeading___Toc424800922)

[Les clauses 55](#__RefHeading___Toc629_3336632364)

[Clause WHERE 55](#__RefHeading___Toc424800926)

[Opérateurs de comparaisons 55](#__RefHeading___Toc424800929)

[Opérateurs AND et OR 56](#__RefHeading___Toc424800930)

[Opérateur IN 57](#__RefHeading___Toc424800936)

[Opérateur BETWEEN 57](#__RefHeading___Toc424800940)

[Opérateur LIKE 58](#__RefHeading___Toc424800945)

[Opérateur IS NULL ou IS NOT NULL 58](#__RefHeading___Toc719_1390794105)

[Mot-clé, clause 60](#__RefHeading___Toc21479_1836404655)

[Mot-clé DISTINCT ou UNIQUE 60](#__RefHeading___Toc424800955)

[Clause GROUP BY 60](#__RefHeading___Toc848_2157261243)

[Fonctions de statistiques 62](#__RefHeading___Toc424800972)

[Condition HAVING 62](#__RefHeading___Toc424800973)

[Commande ORDER BY 63](#__RefHeading___Toc424800976)

[Mot-clé AS (alias) 63](#__RefHeading___Toc424800979)

[Clauses LIMIT, OFFSET, TOP, FETCH NEXT, ROWNUM, FETCH FIRST n ROWS ONLY 65](#__RefHeading___Toc424800989)

[Utilisation avec OFFSET 65](#__RefHeading___Toc19852_1376464157)

[Expression CASE à compléter depuis le livre 67](#__RefHeading___Toc424800994)

[Syntaxe de CASE 67](#__RefHeading___Toc20003_3201136387)

[Jointure SQL 69](#__RefHeading___Toc424801075)

[Types de jointures 69](#__RefHeading___Toc424801077)

[Commande UNION 70](#__RefHeading___Toc20918_3201136387)

[Commande UNION ALL 71](#__RefHeading___Toc20920_3201136387)

[SQL Sous-requête 72](#__RefHeading___Toc471_968127466)

[Requête imbriquée qui retourne un seul résultat 72](#__RefHeading___Toc424801110)

[Requête imbriquée qui retourne plusieurs valeurs 74](#__RefHeading___Toc424801114)

[SQL EXISTS 75](#__RefHeading___Toc424801115)

[SQL ALL 77](#__RefHeading___Toc424801118)

[SQL ANY / SOME 78](#__RefHeading___Toc424801121)

[Astuce 79](#__RefHeading___Toc424801124)

[SQL EXPLAIN 82](#__RefHeading___Toc424801133)

[Commentaires en SQL 83](#__RefHeading___Toc424801141)

[Procédure stockée 84](#__RefHeading___Toc22944_3554795410)

[Fonctions d’agrégation SQL 85](#__RefHeading___Toc424801148)

[Fonction AVG() calcul 85](#__RefHeading___Toc424801152)

[Fonction CONCAT() agrégation 89](#__RefHeading___Toc21601_1836404655)

[Fonction COUNT calcul 90](#__RefHeading___Toc21603_1836404655)

[Fonction MAX() calcul 96](#__RefHeading___Toc21605_1836404655)

[Fonction MIN() calcul 98](#__RefHeading___Toc21607_1836404655)

[Fonction RAND() calcul 100](#__RefHeading___Toc21611_1836404655)

[Fonction ROUND() calcul 101](#__RefHeading___Toc21613_1836404655)

[Fonction SUM() calcul 102](#__RefHeading___Toc21615_1836404655)

[Fonctions de chaînes de caractères 104](#__RefHeading___Toc424801177)

[Fonction BTRIM() caractère 105](#__RefHeading___Toc4248011794)

[Fonction CHAR() caractère 106](#__RefHeading___Toc4248011795)

[Fonction CHAR\_LENGTH() caractère 107](#__RefHeading___Toc4248011796)

[Fonction CHARACTER\_LENGTH() caractère 107](#__RefHeading___Toc4248011797)

[Fonction CHARINDEX() caractère [SQL Server] 107](#__RefHeading___Toc4248011798)

[Fonction CHR() caractère [PostgreSQL] 107](#__RefHeading___Toc4248011799)

[Fonction SUBSTRING() caractère 108](#__RefHeading___Toc424801206)

[Exemple de requête 108](#__RefHeading___Toc424801209)

[Exemple 109](#__RefHeading___Toc424801210)

[Exemple d’utilisation 110](#__RefHeading___Toc424801211)

[Fonction UPPER() caractère 110](#__RefHeading___Toc670_968127466)

[Fonctions mathématiques / numérique 111](#__RefHeading___Toc424801272)

[Fonction ROUND() calcul 111](#__RefHeading___Toc424801279)

[Fonctions de dates et d’heures 114](#__RefHeading___Toc424801285)

[Fonctions SQL utiles pour les dates et les heures 114](#__RefHeading___Toc424801286)

[Fonction NOW() date 118](#__RefHeading___Toc424801287)

[Fonction MONTH() date 119](#__RefHeading___Toc424801305)

[Fonction YEAR() date 120](#__RefHeading___Toc424801311)

[Fonctions de chiffrements 122](#__RefHeading___Toc424801315)

[Dates 122](#__RefHeading___Toc4249073181)

[Fonction NOW() date 122](#__RefHeading___Toc21609_1836404655)

[cobol 122](#__RefHeading___Toc32928_1900359009)

[cf IBM 122](#__RefHeading___Toc4249073201)

[Introduction 122](#__RefHeading___Toc4249073211)

[IBM DB2 e-kit for Database Professionals 122](#__RefHeading___Toc868_3311445400)

[The basics 123](#__RefHeading___Toc4249073221)

[Date functions 126](#__RefHeading___Toc4249073231)

[Changing the date format 129](#__RefHeading___Toc4249073241)

[Custom Date/Time Formatting 130](#__RefHeading___Toc870_3311445400)

[Dates 132](#__RefHeading___Toc424907318)

[cobol 132](#__RefHeading___Toc424907319)

[cf IBM 132](#__RefHeading___Toc424907320)

[Introduction 133](#__RefHeading___Toc424907321)

[IBM DB2 e-kit for Database Professionals 133](#__RefHeading___Toc872_3311445400)

[The basics 133](#__RefHeading___Toc424907322)

[Date functions 136](#__RefHeading___Toc424907323)

[Changing the date format 139](#__RefHeading___Toc424907324)

[Custom Date/Time Formatting 141](#__RefHeading___Toc874_3311445400)

[Présentation 143](#__RefHeading___Toc429581201)

[SQLCA 143](#__RefHeading___Toc429581202)

[La description des tables et des vues 145](#__RefHeading___Toc429581203)

[Les variables hôtes (host variables) 146](#__RefHeading___Toc429581204)

[Les colonnes de longueur variables 147](#__RefHeading___Toc429581205)

[Les variables indicateurs 148](#__RefHeading___Toc429581206)

[Mise à NULL d’une colonne 149](#__RefHeading___Toc429581207)

[L’ordre DECLARE TABLE 150](#__RefHeading___Toc429581208)

[Lire une ligne 151](#__RefHeading___Toc429581209)

[Lire plusieurs lignes 152](#__RefHeading___Toc429581210)

[UPDATE avec un curseur 153](#__RefHeading___Toc429581211)

[L’ordre WHENEVER 154](#__RefHeading___Toc429581212)

[DCLGEN 154](#__RefHeading___Toc429581213)

[Précompilation 155](#__RefHeading___Toc429581214)

[Le BIND 155](#__RefHeading___Toc429581215)

[REBIND, BIND REPLACE et FREE 157](#__RefHeading___Toc429581216)

[Mécanisme de verrouillage 159](#__RefHeading___Toc429581217)

[Le DEADLOCK 160](#__RefHeading___Toc429581218)

[Taille du verrou 160](#__RefHeading___Toc429581219)

[Durée de verrouillage 160](#__RefHeading___Toc429581220)

[Mode de verrouillage 161](#__RefHeading___Toc429581221)

[L’ordre LOCK TABLE 162](#__RefHeading___Toc429581222)

[Verrouillage implicite 163](#__RefHeading___Toc429581223)

[Ordres SQL 163](#__RefHeading___Toc835_3311445400)

[CREATE TABLE 163](#__RefHeading___Toc429581227)

[INSERT 165](#__RefHeading___Toc429581231)

[SELECT 166](#__RefHeading___Toc429581232)

[Normes et Contrôles spécifiques 170](#__RefHeading___Toc21380_145764793)

[Nommage 170](#__RefHeading___Toc20870_1074248952)

[L’email 171](#__RefHeading___Toc21382_145764793)

[Mots réservés 173](#__RefHeading___Toc51547_3889704470)

[SGBD 175](#__RefHeading___Toc424801354)

[Exercices 176](#__RefHeading___Toc424801382_Copie_1)

[Créer des tables mères et des tables filles 176](#__RefHeading___Toc23384_749132057)

[Table mère : FAM 176](#__RefHeading___Toc23399_749132057)

[Table mère : MAR 176](#__RefHeading___Toc23401_749132057)

[Table mère : ART 176](#__RefHeading___Toc23403_749132057)

[Table DEC 176](#__RefHeading___Toc23405_749132057)

[Insertions de lignes dans la table FAM 176](#__RefHeading___Toc23407_749132057)

[Insertion dans la table MAR 176](#__RefHeading___Toc41841_879792418)

[Insertion dans la table ART 176](#__RefHeading___Toc23411_749132057)

[Mise à jour, UPDATE de la table ART 176](#__RefHeading___Toc23413_749132057)

[Suppression, DELETE dans la table MAR 177](#__RefHeading___Toc23415_749132057)

[Affichage, SELECT 177](#__RefHeading___Toc23417_749132057)