



PostgreSQL

prise en main

Cours conçu par **Razvan BIZOI**
Reproduction interdite

Table des matières

MODULE 1 : L'INTRODUCTION	1-1
Qu'est-ce qu'une base de données ?	1-2
Objets de la base de données	1-3
Table	1-6
Intégrité d'une base de données	1-7
Le langage SQL	1-9
Les limites de SQL	1-11
Le langage PL/SQL	1-12
pgAdmin	1-13
Query tool	1-14
psql	1-15
Meta-commandes.....	1-17
Atelier 1	1-20
 MODULE 2 : INTERROGATION DES DONNEES	 2-1
Projection.....	2-2
Les constantes.....	2-5
Opérateur de concaténation	2-6
Opérateurs arithmétiques.....	2-7
Opérateurs de type DATE	2-9
Le traitement de la valeur NULL	2-10
La clause LIMIT	2-11
Atelier 3.2	2-12
Tri du résultat d'une requête	2-13
Atelier 3.3	2-15
La sélection ou restriction	2-16

L'opérateur LIKE	2-18
L'opérateur ~	2-20
L'opérateur IS NULL	2-23
Atelier 4.1	2-24
Les opérateurs BETWEEN et IN	2-26
Les opérateurs AND et OR	2-27
L'opérateur NOT	2-30
Atelier 4.2	2-32
MODULE 3 : INTERROGATION DES DONNEES	3-1
Types de données	3-2
Fonctions et opérateurs de chaînes	3-4
Atelier 5	3-9
Fonctions de calcul arithmétique	3-11
Atelier 6	3-13
Les fonctions de dates	3-14
Opérateurs	3-16
Types intervalle	3-17
Atelier 7	3-18
Les fonctions de conversion	3-19
Atelier 8.1	3-22
Les fonctions générales	3-23
Atelier 8.2	3-24
Les fonctions d'agrégat	3-25
Le groupe	3-27
La sélection de groupe	3-30
Transformer les tables en XML	3-32
Traiter du XML	3-35
MODULE 4 : LES REQUETES MULTI-TABLES	4-1
L'opérateur CROSS JOIN	4-2
L'opérateur JOIN USING	4-3
L'opérateur JOIN ON	4-4
Atelier 10.1	4-5
L'opérateur OUTER JOIN	4-6
Les opérateurs ensemblistes	4-8
L'opérateur UNION	4-10
L'opérateur INTERSECT	4-13
L'opérateur DIFFERENCE	4-14
Atelier 12.1	4-15

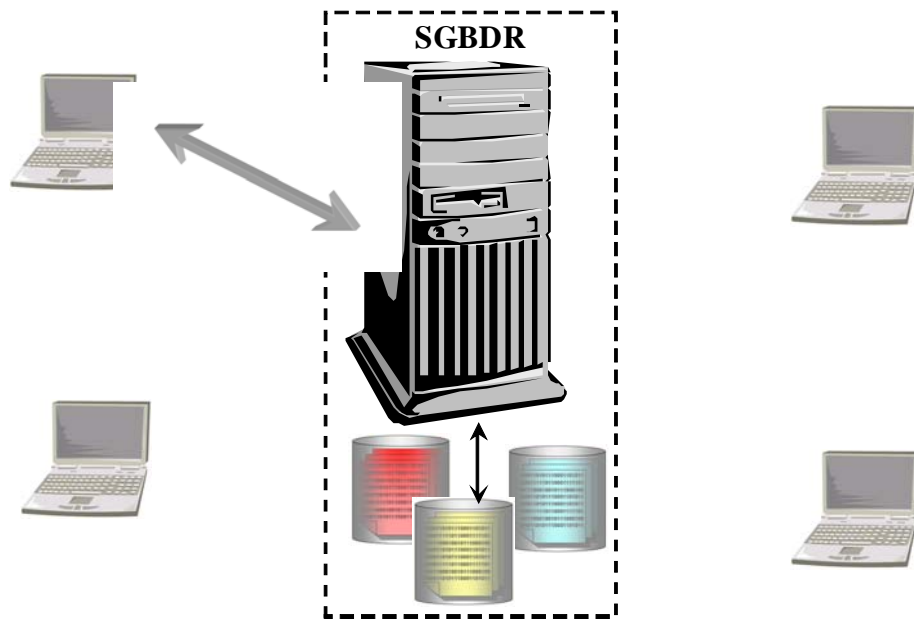
Les sous-requêtes.....	4-16
Sous-requête monolignes.....	4-17
Sous-requête multilignes	4-19
Les opérateurs ANY et ALL	4-22
Sous-requête renvoyant un tableau.....	4-25
Sous-requête synchronisée	4-26
Sous-requête dans la clause FROM.....	4-29
La clause WITH.....	4-30
MODULE 5 : LA MISE A JOUR DES DONNEES.....	5-1
Insertion d'une ligne.....	5-2
Modification des données.....	5-4
Suppression des données	5-5
Contraintes d'intégrité	5-6
Atelier 12.....	5-7
L'isolation.....	5-8
Début et fin de transaction.....	5-11
MODULE 6 : LES OBJETS	6-1
La création d'une table	6-2
Définition de contraintes	6-6
Modifier une table	6-13
Création d'une Vue.....	6-15
Mise à jour dans une Vue	6-16
Contrôle d'intégrité dans une Vue	6-17
Les séquences	6-18
Index B-tree	6-19
Utiliser EXPLAIN	6-21
Utiliser VACUUM	6-22
Création d'un utilisateur	6-23
Création d'un schéma	6-25
Les droits objets.....	6-26
La création d'un groupe.....	6-28
MODULE 7 : LES FONCTIONS PL/PGSQL	7-1
Structure de bloc	7-2
Les fonctions	7-3
Déclaration de variables	7-5
Variables basées	7-6
Interrogation	7-7

La suppression des fonctions.....	7-9
MODULE 8 : LES STRUCTURES DE CONTROLE.....	8-1
Instructions de contrôle	8-2
Structures conditionnelles	8-3
CASE.....	8-5
Structure répéter	8-7
Structure tant que	8-8
Structure pour.....	8-9
MODULE 9 : LES CURSEURS	9-1
L'exécution d'une interrogation	9-2
Les curseurs explicites	9-4
Déclaration	9-5
Ouverture.....	9-6
Traitement des lignes	9-8
Les boucles et les curseurs	9-9
Fermeture	9-11
Atelier Module 9 : .1	9-13
Les curseurs FOR UPDATE	9-14
Accès concurrent et verrouillage.....	9-18
WHERE CURRENT OF.....	9-21
La variable curseur	9-22
MODULE 10 : LES PROCEDURES TRIGGER.....	10-1
La création.....	10-2
Le moment d'exécution.....	10-3
Le niveau d'exécution	10-6
L'utilisation :OLD et :NEW.....	10-8
Le déclenchement conditionnel.....	10-10
Atelier Module 10 : .1	10-11
MODULE 11 : LA MAINTENANCE DES DONNEES.....	11-1
Lancer le serveur	11-2
La création d'une base de données	11-3
Les tablespaces.....	11-4
La sauvegarde.....	11-5
La restauration.....	11-6
La commande COPY	11-7

1

L'Introduction

Qu'est-ce qu'une base de données ?



SQL pour MySQL et PostgreSQL

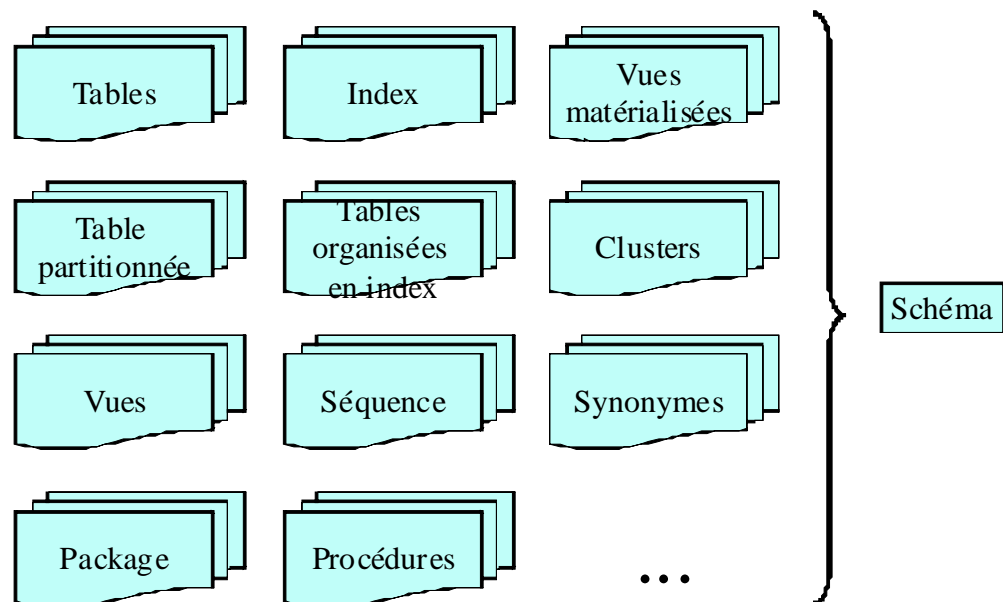
Module 1 : Présentation de l'environnement - Diapo 1.2

On peut définir une base de données simplement comme un stockage permanent de données dans un ou plusieurs fichiers. Une base de données contient non seulement des données, mais aussi leur description. Un système de gestion de base de données est un logiciel qui contrôle ces données et qui inclut la gestion des éléments suivants :

- uniformité de données ;
- gestion de l'utilisateur et de la sécurité ;
- fiabilité ;
- intégrité de données.

Toutes les manipulations s'effectuent au moyen du langage **SQL** (structured query language). Ce langage permet à l'utilisateur de demander au SGBD de créer des tables, d'y ajouter des colonnes, d'y ranger des données et de les modifier, de consulter les données, de définir les autorisations d'accès. Les instructions de consultation des données sont essentiellement de nature prédictive. On y décrit les propriétés des données qu'on recherche, notamment en spécifiant une condition de sélection, mais on n'indique pas le moyen de les obtenir, décision qui est laissée à l'initiative du SGBD.

Objets de la base de données



SQL pour MySQL et PostgreSQL

Module 1 : Présentation de l'environnement - Diapo 1.3

Les modules précédents ont traité des éléments relatifs à la structure logique et physique de la base de données tout en ignorant les questions d'implémentation du modèle relationnel dans une base de données.

L'ensemble des objets qui appartiennent à un compte utilisateur est désigné par le terme schéma. Une base de données peut supporter plusieurs utilisateurs, chacun d'eux possédant un schéma, qui se réfèrent à des structures de données physiques stockées dans des tablespaces.

Une fois que la base de données est conçue, vous pouvez créer le ou les schémas pour supporter les applications. Voici les éléments qui constituent un schéma :

- les tables, colonnes, contraintes et types de données (dont les types abstraits)
- les tables temporaires
- les index
- les vues
- les vues matérialisées
- les synonymes

Les Tables

Les tables représentent le mécanisme de stockage des données dans une base de données. Elles contiennent un ensemble fixe de colonnes, chaque colonne possède un nom ainsi que des caractéristiques spécifiques.

Les tables sont mises en relation via les colonnes qu'elles ont en commun. Vous pouvez faire en sorte que la base de données applique ces relations au moyen de l'intégrité référentielle.

Une table d'objets est une table dont toutes les lignes sont des types de données abstraits possédant chacun un identifiant d'objet (OID, Object ID).

Les tables temporaires

Une table temporaire constitue un mécanisme de stockage de données dans une base de donnée. A l'instar d'une table traditionnelle, elle compte également des colonnes qui se voient assigner chacune un type de données et une longueur. Par contre, même si la définition d'une table temporaire est maintenue de façon permanente dans la base de données, les données qui y sont insérées sont conservées seulement le temps d'une session ou d'une transaction.

Les index

L'index est une structure de base de données utilisée par le serveur pour localiser rapidement une ligne dans une table.

Il en existe trois types principaux :

- L'index de cluster stocke les valeurs de clé d'un cluster de tables.
- L'index de table stocke les valeurs des lignes d'une table, pour la colonne, ou l'ensemble des colonnes, sur laquelle il a été défini, ainsi que les identifiants d'enregistrements correspondants.
- L'index bitmap est un type particulier d'index conçu pour supporter des requêtes sur des tables volumineuses dont les colonnes contiennent peu de valeurs distinctes.

La vue

La vue, ou table virtuelle, n'a pas d'existence propre; aucune donnée ne lui est associée. Seule sa description est stockée, sous la forme d'une requête faisant intervenir des tables de la base ou d'autres vues.

Au niveau conceptuel, vous pouvez vous représenter une vue comme étant un masque qui recouvre une ou plusieurs tables de base et en extrait ou modifié les données demandées.

Lorsque vous interrogez une vue, celle-ci extrait de la table sous-jacente les valeurs demandées, puis les retourne dans le format et l'ordre spécifiés dans sa définition. Etant donné qu'aucune donnée physique n'est directement associée aux vues, ces dernières ne peuvent pas être indexées.

Les vues sont souvent employées pour assurer la sécurité des données au niveau lignes et colonnes.

Les vues d'objets

Les vues d'objets représentent un moyen d'accès simplifié aux types de données abstraits. Vous pouvez employer ces vues pour obtenir une représentation relationnelle objet de vos données relationnelles. Les tables sous-jacentes demeurent inchangées ; ce sont les vues qui supportent les définitions de types de données abstraits.

Les vues matérialisées

Une vue matérialisée est un objet générique utilisé pour synthétiser, précalculer, répliquer ou distribuer des données. Vous pouvez utiliser des vues matérialisées pour fournir des copies locales de données distantes à vos utilisateurs ou pour stocker des données dupliquées dans la même base de données.

Vous pouvez implémenter des vues matérialisées de façon qu'elles soient en lecture seule ou qu'elles puissent être mises à jour.

Les utilisateurs peuvent interroger une vue matérialisée, ou bien l'optimiseur peut dynamiquement rediriger des requêtes vers une vue matérialisée si elle permet d'accéder plus rapidement aux données qu'en interrogeant directement la source.

Contrairement aux vues traditionnelles, les vues matérialisées stockent des données et occupent de l'espace dans la base de données.

Les séquences

Les séquences sont utilisées pour simplifier les tâches de programmation ; elles fournissent une liste séquentielle de numéros uniques. Les définitions de séquences sont stockées dans le dictionnaire de données.

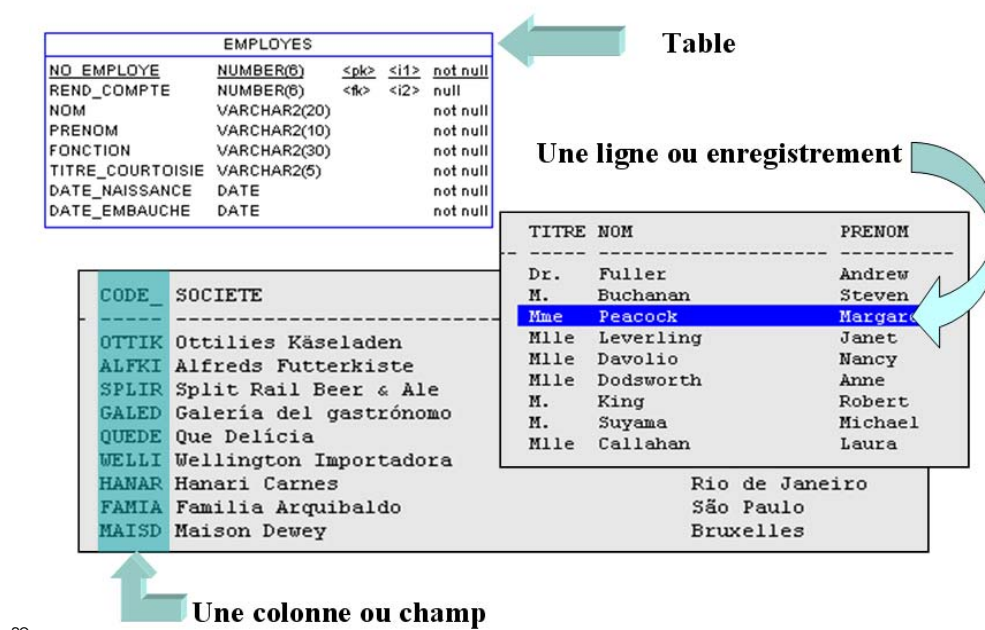
Chaque requête consécutive sur la séquence retourne une valeur incrémentée, telle qu'elle est spécifiée dans la définition de la séquence. Les numéros d'une séquence peuvent être employés en boucle ou bien incrémentés jusqu'à atteindre un seuil prédéfini.

Les synonymes

Les synonymes sont des objets qui cachent la complexité de la base de données. Ils peuvent servir de pointeurs vers des tables, des vues, des procédures, des fonctions, des packages et des séquences. Ils peuvent pointer vers des objets dans la base locale ou bien dans des bases distantes, ce qui requiert l'utilisation de liens de base de données.

L'utilisateur a donc uniquement besoin de connaître le nom du synonyme. Les synonymes publics sont partagés par tous les utilisateurs d'une base, tandis que les synonymes privés appartiennent à des comptes individuels.

Table



sq

Une table sert à stocker les données auxquelles l'utilisateur doit accéder. C'est l'unité fondamentale de stockage physique des données dans une base. Généralement, c'est aux tables que font référence les utilisateurs pour accéder aux données. Une base peut être constituée de plusieurs tables reliées entre elles. Une table contient un ensemble fixe de colonnes.

Colonnes

Une colonne, ou champ représente une partie d'une table et constitue la plus petite structure logique de stockage d'une base de données. Chaque colonne possède un nom ainsi qu'un type de donnée, qui déterminent ses caractéristiques spécifiques. Dans la représentation d'une table, une colonne est une structure verticale qui contient des valeurs sur chaque ligne de la table.

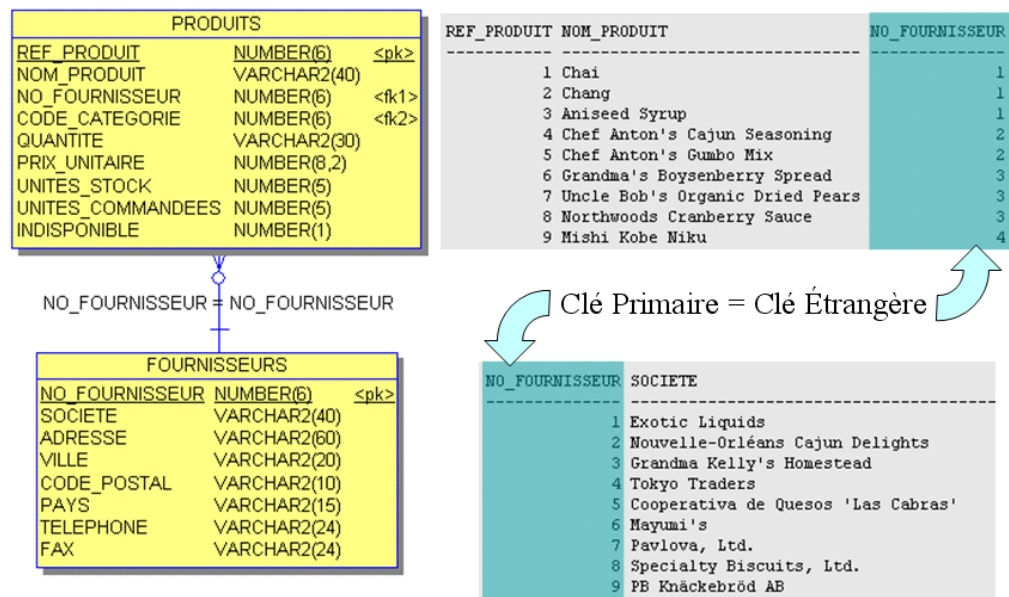
Lignes

Une ligne de données est une collection de valeurs inscrites dans les colonnes successives d'une table, l'ensemble formant un enregistrement unique. Par exemple, la table **EMPLOYES** compte 9 enregistrements ou lignes de données. Le nombre de lignes augmente ou diminue en fonction des ajouts et suppressions des employés.

Types de données

Un type de donnée détermine l'ensemble des valeurs qu'il est possible de stocker dans une colonne de la base de données. Une colonne se voit attribuer un type de données et une longueur. Pour les colonnes de type **number**, il est possible de spécifier des caractéristiques additionnelles relatives à la précision et à l'échelle. La précision détermine le nombre total de chiffres que peut prendre la valeur numérique, l'échelle le nombre de chiffre que peut prendre la partie décimale. Par exemple, **number (10,2)** spécifie une colonne à dix chiffres, avec deux chiffres après la virgule. La précision par défaut (maximale) est de trente-huit chiffres.

Intégrité d'une base de données



SQL pour MySQL et PostgreSQL

Module 1 : Présentation de l'environnement - Diapo 1.5

L'intégrité des données garantit que les données de la base sont exactes, en d'autres termes qu'elles vérifient des règles d'intégrité exprimées sous la forme de contraintes sur les colonnes. Ces contraintes valident les valeurs des données placées dans la base, garantissent l'absence de données dupliquées ou le respect des règles de gestion après modification ou ajout de données. Elles peuvent être mises en place aussi bien au niveau de la colonne qu'au niveau de la table.

Durant la conception d'une base de données, les règles d'intégrité sont d'abord intégrées à travers l'utilisation de contraintes. Sur le plan technique, les contraintes d'une base de données sont constituées :

- des clés primaires ;
- des clés étrangères ;
- des contraintes uniques ;
- des contraintes de contrôle ;
- de la précision et du nombre de décimales des données ;
- « **NULL** » / « **NOT NULL** ».

Deux tables peuvent être reliées entre elles si les valeurs d'une colonne dépendent des valeurs d'une colonne d'une autre table ; une telle relation est appelée parent/enfant. L'intégrité référentielle garantit que les données de tables reliées sont cohérentes et synchronisées. Ces données doivent vérifier des règles exprimées sous la forme de contraintes référentielles. La représentation de ces contraintes nécessite la définition de clés. Une clé est une valeur de colonne d'une table, ou une combinaison de valeurs de colonnes, qui permet d'identifier une ligne de cette table ou d'établir une relation avec une autre table. Il existe deux types de clés : primaires et étrangères.

La présentation faite dans ce module concerne les contraintes principales, pour les autres contraintes et pour plus de détails, rappelez-vous à la création des tables.

Clés primaires

Une clé primaire rend une ligne de données unique dans une table. Elle sert généralement à joindre des tables apparentées ou à interdire la saisie d'enregistrements dupliqués. Par exemple, le numéro de Sécurité sociale d'un employé est considéré comme la clé primaire idéale car il est unique.

Attention

Une table ne peut comporter qu'une seule clé primaire, même lorsque celle-ci est constituée d'une combinaison de plusieurs colonnes.



Contrainte unique

Il est possible de spécifier une contrainte unique pour une colonne de clé non primaire afin de garantir que toutes les valeurs de cette colonne seront uniques. Par exemple, une contrainte unique conviendra à une colonne de type numéro de Sécurité sociale. Une entreprise de téléphonie peut appliquer une contrainte unique à la colonne PHONE_NUMBER, car les clients ne doivent posséder que des numéros de téléphone uniques.

Clés étrangères

Une clé étrangère d'une table référence une clé primaire d'une autre table. Elle est définie dans des tables enfant et assure qu'un enregistrement parent a été créé avant un enregistrement enfant et que l'enregistrement enfant sera supprimé avant l'enregistrement parent.

L'image montre comment s'appliquent les contraintes de clés étrangère et primaire. La colonne NO_FOURNISSEUR de la table PRODUITS référence la colonne NO_FOURNISSEUR de la table FOURNISSEURS. FOURNISSEURS est la table parent et PRODUITS la table enfant. Pour créer un enregistrement dans la table PRODUITS, il faut que le NO_FOURNISSEUR existe d'abord dans la table FOURNISSEURS.

Clause NOT NULL

Si vous examinez la table COMMANDES (voir Annexe 1), vous pouvez remarquer une clause NOT NULL pour les colonnes DATE_ENVOI et PORT.

Cette clause signifie que la base n'acceptera pas l'insertion d'une ligne ne comportant pas de valeur pour ces colonnes. En d'autres termes, il s'agit de champs obligatoires.

La clause NOT NULL est synonyme d'obligation. Une colonne définie avec cette clause ne sera jamais vide.

Le langage SQL

LMD		LDD	
LID		LCD	
SELECT	INSERT UPDATE DELETE	GRANT REVOKE	CREATE ALTER TRUNCATE DROP RENAME

SQL pour MySQL et PostgreSQL

Module 1 : Présentation de l'environnement - Diapo 1.6

Les **SGBD** (systèmes de gestion de bases de données) proposent un langage de requête dénommé **SQL** (structured query language) pour la création et l'administration des objets de la base, pour l'interrogation et les manipulations des informations stockées. Présenté pour la première fois en 1973 par une équipe de chercheurs d'IBM, ce langage a rapidement été adopté comme standard potentiel, et pris en charge par les organismes de normalisation ANSI et ISO.

Une instruction SQL constitue une **requête**, c'est-à-dire la description d'une opération que le SGBD doit exécuter. Une requête peut être introduite au terminal, auquel cas le résultat éventuel (par exemple dans le cas d'une consultation de données) de l'exécution de la requête apparaît à l'écran. Cette requête peut également être envoyée par un programme (écrit en Pascal, C, COBOL, Basic ou Java) au SGBD. Nous développerons plus particulièrement la formulation interactive des requêtes SQL.

Les instructions SQL peuvent être regroupées en deux catégories principales :

- Le Langage de Manipulation de Données et de modules, ou LMD (en anglais DML), pour déclarer les procédures d'exploitation et les appels à utiliser dans les programmes.
On peut également rajouter une composante pour l'interrogation de la base : Langage d'interrogation de Données.
- Le Langage de Définition de Données ou LDD (en anglais DDL), à utiliser pour déclarer les structures logiques de données et leurs contraintes d'intégrité; on peut également rajouter une composante pour la gestion des accès aux données : Langage de Contrôle de Données (en anglais DCL).

Langage de manipulation de données

Le LMD permet d'insérer, de modifier, de supprimer, et de sélectionner des données dans la base. Comme son nom l'indique, il permet de travailler avec les informations contenues dans les structures d'accueil de la base de données.

Les instructions de base LMD sont :

INSERT	Ajoute des lignes de données dans une table
DELETE	Supprime des lignes de données d'une table
UPDATE	Modifie des données dans une table
SELECT	Extrait des lignes de données directement à partir d'une table ou au moyen d'une vue
COMMIT	Applique des changements qui deviennent permanents pour les transactions en cours
ROLLBACK	Annule les changements apportés depuis la dernière validation « COMMIT »

Langage de définition de données

Le LDD permet d'accomplir les tâches suivantes :

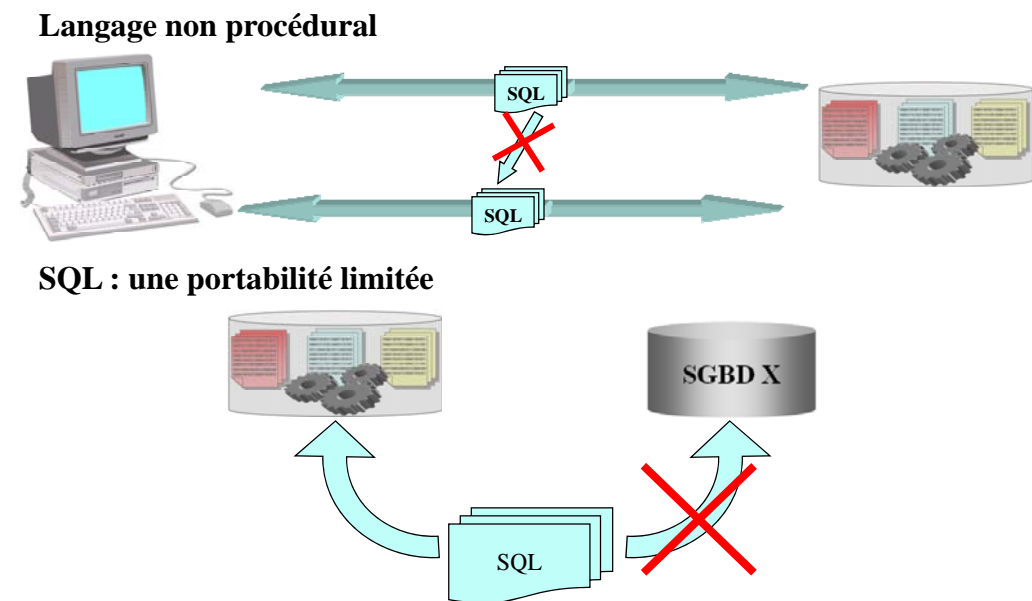
- créer un objet de base de données ;
- supprimer un objet de base de données ;
- modifier un objet de base de données ;
- accorder des privilèges sur un objet de base de données ;
- retirer des privilèges sur un objet de base de données.

Il est important de comprendre qu'une base de données valide une transaction en cours, avant ou après chaque instruction LDD. Ainsi, si vous étiez en train d'insérer des enregistrements dans la base de données et qu'une instruction LDD comme `CREATE TABLE` était émise, les données insérées seraient validées et écrites dans la base.

Les instructions de base LDD sont :

ALTER PROCEDURE	Recompile une procédure stockée
ALTER TABLE	Ajoute une colonne, redéfinit une colonne, modifie une allocation d'espace
ANALYZE	Recueille des statistiques de performances pour les objets de base de données qui doivent alimenter l'optimiseur statistique
CREATE TABLE	Crée une table
CREATE INDEX	Crée un index
DROP INDEX	Supprime un index
DROP TABLE	Supprime une table
GRANT	Accorde des privilèges ou des rôles à un utilisateur ou à un autre rôle
TRUNCATE	Supprime toutes les lignes d'une table
REVOKE	Supprime les privilèges d'un utilisateur ou d'un rôle

Les limites de SQL



PostgreSQL prise en main

Module 1 : Présentation de l'environnement - Diapo 1.7

Langage non procédural

SQL est un langage non procédural. Vous l'utilisez pour indiquer au système quelles données rechercher ou modifier sans lui indiquer comment réaliser ce travail.

SQL ne dispose pas d'instructions pour contrôler le flux d'exécution du programme, pour définir une fonction ou exécuter une boucle, ni d'expressions conditionnelles du type `if ... then ... else`. Toutefois, comme vous pourrez le constater par la suite dans ce module, le système fournit un langage procédural appelé PL/SQL qui constitue une extension au langage SQL.

SQL dispose d'un ensemble fixe de types de données ; vous ne pouvez pas en définir de nouveaux.

Une portabilité limitée

Lorsqu'une application utilise une base de données, sa portabilité concerne les domaines suivants :

- portabilité des données vers des matériels différents, où leur représentation est différente,
- portabilité de l'architecture physique de la base,
- portabilité des requêtes d'accès au SGBD avec, sous-jacent, le problème des types de données,
- portabilité des permissions administratives d'accès.

C'est le concept de modèle tabulaire de données, où l'on peut accéder aux informations par le contenu, qui a la portabilité la plus importante dans SQL. Dans une moindre mesure, la manipulation simple de données est portable. Dans une mesure encore moindre, la définition des données est réutilisable d'un SGBDR à l'autre. Mais en pratique le portage demandera encore beaucoup d'attention et d'efforts, du fait des différences entre les différents SGBDR commercialisés par les éditeurs.

Le langage PL/SQL

■ PL/SQL comprend :

- la partie LID de SQL
- la partie LMD de SQL
- la gestion des transactions
- les fonctions de SQL
- plus une partie procédurale



SQL
if...then
SQL
else
SQL
end if...



■ PL/SQL est donc un langage algorithmique complet.

■ Restriction : la partie LDD

PostgreSQL prise en main

Module 1 : Présentation de l'environnement - Diapo 1.8

Le langage PL/SQL (Procedural Language/SQL) comme son nom l'indique est une extension du langage SQL. Il vous permet à la fois d'insérer, de supprimer, de mettre à jour des données et d'utiliser également des techniques de programmation propres aux langages procéduraux tels que des boucles ou des branchements.

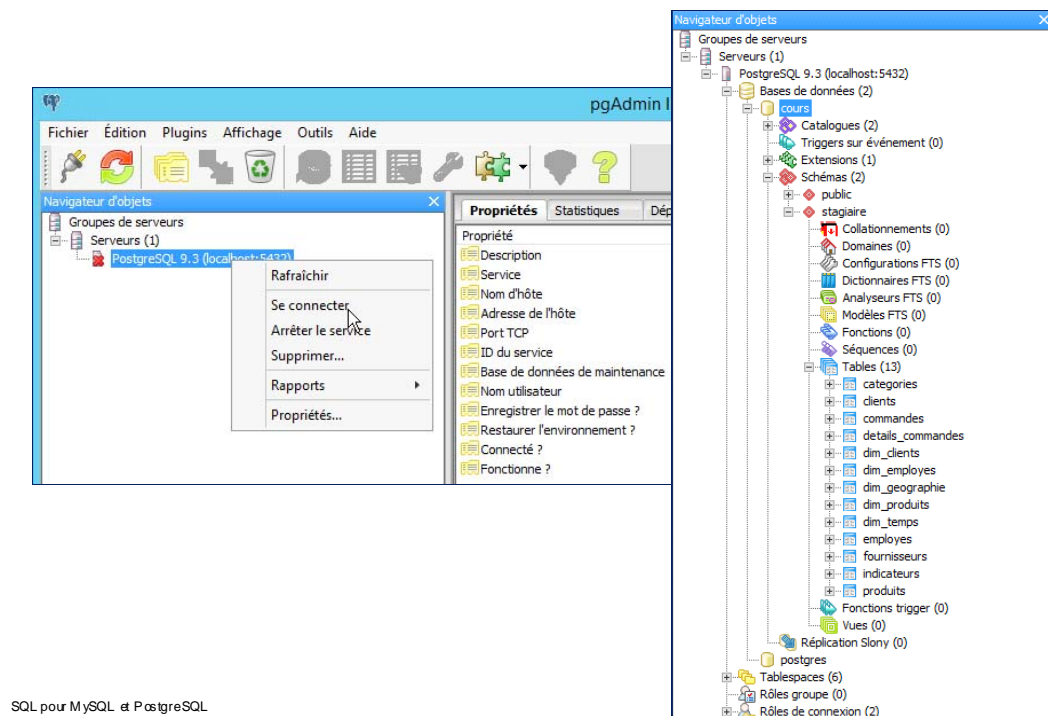
Ainsi, le langage PL/SQL combine la puissance de manipulation des données du SQL avec la puissance de traitement d'un langage procédural.

De plus, PL/SQL vous permet de grouper de manière logique un ensemble d'instructions et de les envoyer vers le noyau sous la forme d'un seul bloc. Cette caractéristique permet de réduire fortement les temps de communication entre l'application et la base de données.

PL/SQL, langage de programmation éprouvé, offre de nombreux avantages :

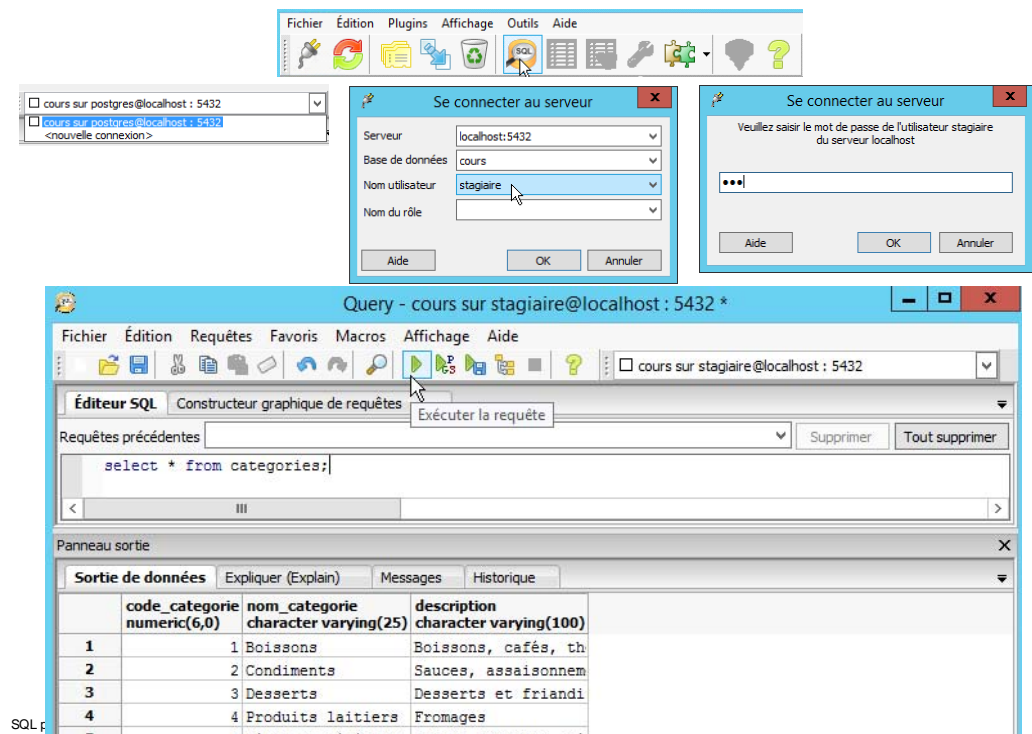
- intégration parfaite du SQL,
- support de la programmation orientée objet,
- très bonnes performances,
- portabilité,
- facilité de programmation,
- parfaite intégration à Java.

pgAdmin

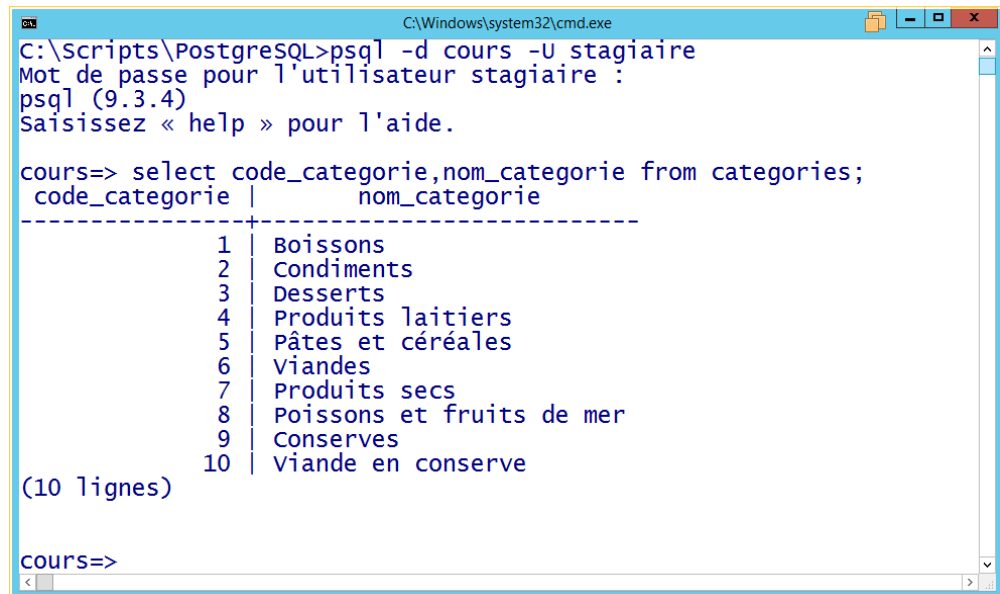


pgAdmin est un outil graphique composé d'une interface pour créer, exécuter et optimiser des instructions SQL (LDD, LMD, LID et LCD). Bien que toutes les instructions générées par le biais de cet outil soient exécutables en ligne de commande, la documentation affirme que certaines requêtes peuvent y être composées graphiquement de manière plus intuitive.

Query tool



psql



```

C:\Windows\system32\cmd.exe
C:\Scripts\PostgreSQL>psql -d cours -U stagiaire
Mot de passe pour l'utilisateur stagiaire :
psql (9.3.4)
Saisissez « help » pour l'aide.

cours=> select code_categorie,nom_categorie from categories;
 code_categorie |      nom_categorie
-----+-----
1 | Boissons
2 | Condiments
3 | Desserts
4 | Produits laitiers
5 | Pâtes et céréales
6 | Viandes
7 | Produits secs
8 | Poissons et fruits de mer
9 | Conserves
10 | Viande en conserve
(10 lignes)

cours=>

```

SQL pour MySQL et PostgreSQL

Module 1 : Présentation de l'environnement - Diapo 1.29

PostgreSQL a une interface en mode texte psql. Il vous permet de saisir des requêtes de façon interactive, de les exécuter et de voir les résultats de ces requêtes. Alternativement, les entrées peuvent être lues à partir d'un fichier. De plus, il fournit un certain nombre de méta-commandes et plusieurs fonctionnalités style shell pour faciliter l'écriture des scripts et automatiser un nombre varié de tâches.

Options

-c commande, --command=commande

Indique que psql doit exécuter une chaîne de commande, commande, puis s'arrêter.

-d nombase, --dbname=nombase

Indique le nom de la base de données où se connecter. Ceci est équivalent à spécifier nombase comme premier argument de la ligne de commande qui n'est pas une option.

-f nomfichier, --file=nomfichier

Utilise le fichier nomfichier comme source des commandes au lieu de lire les commandes de façon interactive.

-h nomhôte, --host=nomhôte

Indique le nom d'hôte de la machine sur lequel le serveur est en cours d'exécution. Si la valeur commence avec un slash, elle est utilisée comme répertoire du socket de domaine Unix.

-L nomfichier, --log-file=nomfichier

Écrit tous les résultats des requêtes dans le fichier nomfichier en plus de la destination habituelle.

-o nomfichier, --output=nomfichier

Dirige tous les affichages de requêtes dans le fichier nomfichier. Ceci est équivalent à la commande \o.

-p port, --port=port

Indique le port TCP ou l'extension du fichier socket de domaine local Unix sur lequel le serveur attend les connexions. Par défaut, il s'agit de la valeur de la variable d'environnement PGPORT ou, si elle n'est pas initialisée, le port spécifié au moment de la compilation, habituellement 5432.

-U nomutilisateur, --username=nomutilisateur

Se connecte à la base de données en tant que l'utilisateur nomutilisateur au lieu de celui par défaut.

-, --help

Affiche de l'aide sur les arguments en ligne de commande de psql et quitte.



```
C:\Scripts\PostgreSQL>psql -d cours -U stagiaire
Mot de passe pour l'utilisateur stagiaire : PWD
psql (9.3.4)
Saisissez « help » pour l'aide.

cours=> select code_categorie, nom_categorie from categories;
 code_categorie |      nom_categorie
-----+-----
          1 | Boissons
          2 | Condiments
          3 | Desserts
          4 | Produits laitiers
          5 | Pâtes et céréales
          6 | Viandes
          7 | Produits secs
          8 | Poissons et fruits de mer
          9 | Conserves
         10 | Viande en conserve
(10 lignes)
```

Meta-commandes

Tout ce que vous saisissez dans psql qui commence par un antislash est une méta-commande psql qui est traitée par psql lui-même. Ces commandes aident à rendre psql plus utile pour l'administration ou pour l'écriture de scripts.

Le format d'une commande psql est l'antislash suivi immédiatement d'un verbe de commande et de ses arguments. Les arguments sont séparés du verbe de la commande et les uns des autres par un nombre illimité d'espaces blancs.

\q

Pour sortir de psql et retourner à votre shell.

\! [commande]

Lance un environnement de commande séparé ou exécute la commande.

\?

Affiche l'aide sur les commandes antislash.

\d

Pour chaque relation (table, vue, index, séquence ou table distante) ou type composite correspondant au motif, affiche toutes les colonnes, leur types, le tablespace (s'il ne s'agit pas du tablespace par défaut) et tout attribut spécial tel que NOT NULL ou les valeurs par défaut. Les index, contraintes, règles et déclencheurs associés sont aussi affichés, ainsi que la définition de la vue si la relation est une vue.



```
cours=# \d categories
```

Table « stagiaire.categories »

Colonne	Type	Modificateurs
code_categorie	numeric(6,0)	non NULL
nom_categorie	character varying(25)	non NULL
description	character varying(100)	non NULL

Index :

"categories_pk" PRIMARY KEY, btree (code_categorie), tablespace « itb_tran »

Référencé par :

TABLE "produits" CONSTRAINT "prod_cate_fk" FOREIGN KEY (code_categorie) REFERENCES categories(code_categorie) ON UPDATE RESTRICT

```
cours=> \di
```

Liste des relations

Schéma	Nom	Type	Propriétaire	Table
stagiaire	categories_pk	index	stagiaire	categories
stagiaire	clients_pk	index	stagiaire	clients
stagiaire	comm_clie_fk	index	stagiaire	commandes
stagiaire	comm_empl_fk	index	stagiaire	commandes
stagiaire	commandes_pk	index	stagiaire	commandes
stagiaire	det_comm_comm_fk	index	stagiaire	details_commandes
stagiaire	det_comm_prod_fk	index	stagiaire	details_commandes
stagiaire	details_commandes_pk	index	stagiaire	details_commandes
stagiaire	dim_clients_pk	index	stagiaire	dim_clients
stagiaire	dim_employes_pk	index	stagiaire	dim_employes

```

stagiaire | dim_geographie_pk | index | stagiaire | dim_geographie
stagiaire | dim_produits_pk   | index | stagiaire | dim_produits
stagiaire | dim_temps_pk       | index | stagiaire | dim_temps
stagiaire | empl_empl_fk       | index | stagiaire | employes
stagiaire | employes_pk        | index | stagiaire | employes
stagiaire | fournisseurs_pk    | index | stagiaire | fournisseurs
stagiaire | indicateurs_pk     | index | stagiaire | indicateurs
stagiaire | prod_cate_fk       | index | stagiaire | produits
stagiaire | prod_four_fk       | index | stagiaire | produits
stagiaire | produits_pk        | index | stagiaire | produits
(20 lignes)

```

```
cours=> \dt
```

```

Liste des relations

```

Schéma	Nom	Type	Propriétaire
stagiaire	categories	table	stagiaire
stagiaire	clients	table	stagiaire
stagiaire	commandes	table	stagiaire
stagiaire	details_commandes	table	stagiaire
stagiaire	dim_clients	table	stagiaire
stagiaire	dim_employes	table	stagiaire
stagiaire	dim_geographie	table	stagiaire
stagiaire	dim_produits	table	stagiaire
stagiaire	dim_temps	table	stagiaire
stagiaire	employes	table	stagiaire
stagiaire	fournisseurs	table	stagiaire
stagiaire	indicateurs	table	stagiaire
stagiaire	produits	table	stagiaire

(13 lignes)

```
cours=> \db
```

```

Liste des tablespaces

```

Nom	Propriétaire	Emplacement
dtb_star	postgres	S:\Program Files\PostgreSQL\9.5\data\pg_tblspc\COURS_PG\DTB_STAR
dtb_tran	postgres	S:\Program Files\PostgreSQL\9.5\data\pg_tblspc\COURS_PG\DTB_TRAN
itb_star	postgres	S:\Program Files\PostgreSQL\9.5\data\pg_tblspc\COURS_PG\ITB_STAR
itb_tran	postgres	S:\Program Files\PostgreSQL\9.5\data\pg_tblspc\COURS_PG\ITB_TRAN
pg_default	postgres	
pg_global	postgres	

(6 lignes)

\pset

Vous permet de spécifier les options d'affichage. Vous pouvez utiliser les formats suivants : format, border, expanded, fieldsep, footer, null, recordsep, tuples_only, title, tableattr, pager



```

cours=# \pset
border                1
columns               0
expanded              off
fieldsep              '|'
fieldsep_zero         off
footer                on

```



```

format                aligned
linestyle             ascii
null                  ''
numericlocale         off
pager                 1
pager_min_lines        0
recordsep              '\n'
recordsep_zero         off
tableattr
title
tuples_only           on
unicode_border_linestyle single
unicode_column_linestyle single
unicode_header_linestyle single

cours=> \pset null '-----'
L'affichage de null est « ----- ».
cours=> select nom,commission from employees
cours-> where commission is null;
Giroux      | -----
Fuller      | -----
Brasseur    | -----
Poupard     | -----
Maurer      | -----
Callahan    | -----
Etienne     | -----
Grangirard  | -----
Guerdon     | -----
Devie       | -----
Pouetre     | -----
Ziliox      | -----
Lampis      | -----

```

Atelier 1



- Configurer l'environnement de travail
- Comprendre la base de données utilisée pour les ateliers



PostgreSQL prise en main

Module 1 : Présentation de l'environnement - Diapo 1.12

Questions

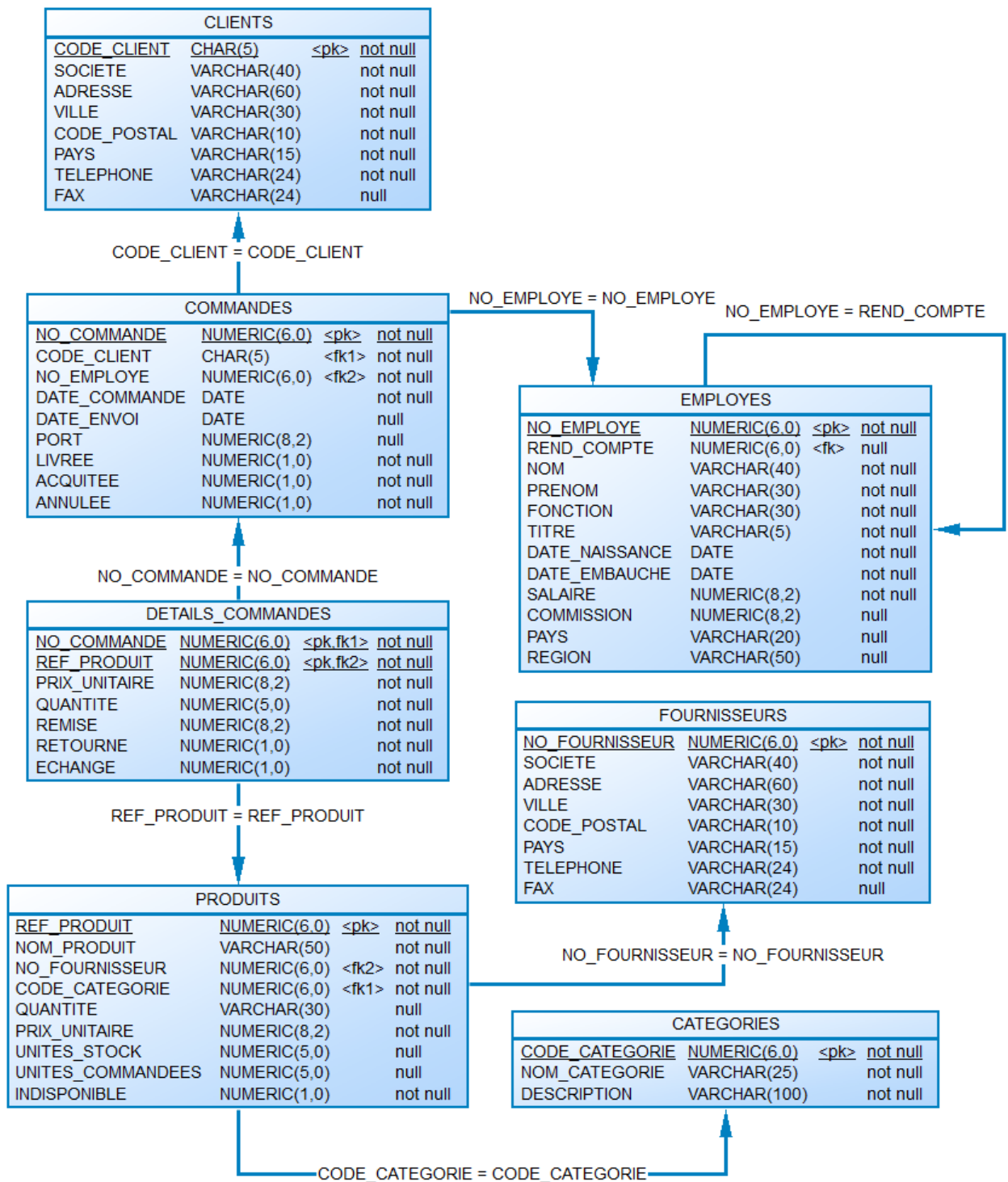


- 1-1. Une table peut-elle avoir plusieurs clés primaires ?
- 1-2. Une table peut-elle avoir une contrainte unique si elle possède déjà une clé primaire ?
- 1-3. Une table qui possède une clé étrangère est-elle une table enfant ou une table parent ?
- 1-4. Que signifie LMD ?
- 1-5. Que signifie LDD ?

Exercice n° 1 Les tables utilisées pour les ateliers

Sachant que le symbole <pk> signifie clé primaire et <fk> la clé étrangère.

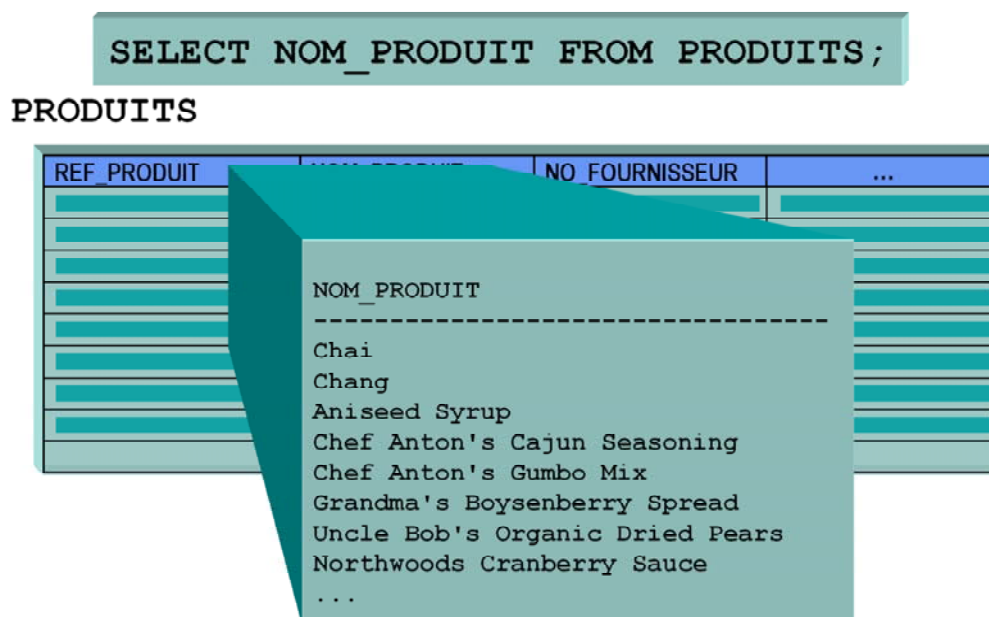
Quelles sont les tables en relation parent enfant ?



2

Interrogation des données

Projection



PostgreSQL prise en main

Module 1 : Présentation de l'environnement - Diapo 1.35

L'opération de projection permet de retenir certaines ou toutes les colonnes d'une table et retourne l'intégralité des enregistrements de la table.

Une projection s'exprime à l'aide du langage SQL par la clause « **SELECT** ».

Des quatre instructions du LMD, « **SELECT** » est celle qui est exécutée le plus souvent dans une application réelle, car les enregistrements sont plus souvent lus qu'ils ne sont modifiés.

L'instruction « **SELECT** » est un outil puissant et sa syntaxe est compliquée en raison des nombreuses possibilités qui vous sont offertes pour former une instruction valide en combinant les tables, les colonnes, les fonctions et les opérateurs. Par conséquent, au lieu d'examiner la syntaxe complète de cette instruction, on va commencer par découvrir la syntaxe au fur et à mesure de son utilisation.

```
SELECT [ALL | DISTINCT] {*,[COLONNE1 [AS] ALIAS1[,...]}
FROM NOM_TABLE;
```

ALL La requête extrait l'intégralité des enregistrements de la table. C'est l'option par défaut.

DISTINCT | UNIQUE La requête extrait les enregistrements de la table qui sont uniques, la règle d'unicité s'applique à l'ensemble des colonnes sélectionnées.

***** La projection totale, permet d'extraire l'ensemble des colonnes pour la table mentionné dans la clause FROM.

COLONNE Une liste des noms de colonnes séparées par virgule, de la table mentionnée dans la clause FROM, que vous souhaitez extraire dans la projection.

[AS] ALIAS

Si l'en-tête de colonne n'est pas assez significatif, il est possible de définir un alias qui se déclare immédiatement après la colonne ; il peut être précédé par AS, sous la forme d'une chaîne de caractères placée ou non entre guillemets.

Dans PostgreSQL l'opérateur « DESC » à la syntaxe suivante :

\d nom_table

cours=> **\d categories**

Table « stagiaire.categories »

Colonne	Type	Modificateurs
code_categorie	numeric(6,0)	non NULL
nom_categorie	character varying(25)	non NULL
description	character varying(100)	non NULL

Index :

"categories_pk" PRIMARY KEY, btree (code_categorie), tablespace « itb_tran »

Référencé par :

TABLE "produits" CONSTRAINT "prod_cate_fk" FOREIGN KEY (code_categorie) REFERENCES categories(code_categorie) ON UPDATE RESTRICT

La requête suivante est une projection partielle des tables CATEGORIES et COMMANDES.

cours=> **select code_categorie code,**

cours-> **NOM_CATEGORIE "Catégorie de produits" from CATEGORIES ;**

code	Catégorie de produits
1	Boissons
2	Condiments
3	Desserts
4	Produits laitiers
5	Pâtes et céréales
6	Viandes
7	Produits secs
8	Poissons et fruits de mer
9	Conserves
10	Viande en conserve

(10 lignes)

cours=> **SELECT code_client, date_commande, no_employe, port**

cours-> **FROM COMMANDES ;**

code_client	date_commande	no_employe	port
LONEP	2010-02-02	84	50.10
PERIC	2010-02-02	78	97.60
BOTTM	2010-02-02	72	89.30
SPECD	2010-02-02	111	86.20
WELLI	2010-02-02	39	71.90
GROSR	2010-02-02	88	61.40
MAGAA	2010-02-02	51	58.20
GALED	2010-02-02	3	69.70
BONAP	2010-02-02	45	77.20
PRINI	2010-02-02	65	79.30

OCEAN		2010-02-02		90		74.60
WARTH		2010-02-02		106		89.20
OTTIK		2010-02-02		40		66.70
WHITC		2010-02-02		84		95.90
BSBEV		2010-02-02		1		57.60
...						

Dans les deux exemples précédents, vous avez remarqué que SQL*Plus applique une certaine mise en forme aux données qu'il présente.

La présentation de résultats se fait sous forme tabulaire où :

- Il convertit tous les noms de colonnes ou les alias, qui ne sont pas placés entre les guillemets, en majuscules.
- Les en-têtes de colonnes ne peuvent pas être plus longs que la longueur définie des colonnes.



Les guillemets « " » sont utilisés seulement pour définir l'alias d'une colonne, par exemple "Alias de Colonne". Le symbole de délimitation des chaînes de caractères est la simple cote « ' », par exemple : 'Chaîne de caractères'.

La requête suivante est une projection partielle de la table EMPLOYEES, pour extraire les différentes fonctions des employés.



```
cours=> select fonction from employees;
        fonction
-----
Président
Vice-Président
Vice-Président
Chef des ventes
Chef des ventes
Chef des ventes
Chef des ventes
Chef des ventes
Chef des ventes
Représentant(e)
...

cours=> select distinct fonction from employees;
        fonction
-----
Vice-Président
Représentant(e)
Assistante commerciale
Président
Chef des ventes
(5 lignes)
```

Dans le premier exemple, on peut remarquer que la clause « **ALL** » et la requête extraient l'intégralité des enregistrements de la table.

Dans le deuxième exemple la requête extrait les enregistrements de la table qui sont uniques, la règle d'unicité s'applique à la seule colonne sélectionnée.

Les constantes

Une constante est une variable dont la valeur, fixée au moment de sa définition, n'est pas modifiable.

Constante numérique

Une constante numérique définit un nombre contenant éventuellement un signe, un point décimal et un exposant, puissance de dix. Le point décimal ne peut être défini que par le caractère point « . ». Les caractères numériques doivent être contigus (sans espaces pour milliers par exemple).



```
cours=# SELECT 0 "1",
cours=# -1234567890 "2",
cours=# +1234567890 "3",
cours=# -123.456 "4",
cours=# +123.456 "5",
cours=# -1E+123 "6";
```

1	2	3	4	5	
0	-1234567890	1234567890	-123.456	123.456	-10000000

```
cours=# SELECT -123,456, +123,456,-1E+123;
?column? | ?column? | ?column? | ?column? |
```

-123	456	123	456	-1000000

Constante chaîne de caractère

Une constante chaîne de caractère est représentée par une chaîne de caractères entre cotes « ' » où les lettres en majuscules et en minuscules sont considérées comme deux caractères différents. Il est possible d'insérer une apostrophe à l'intérieur d'une chaîne de caractères en la représentant par deux apostrophes consécutives.



```
cours=# SELECT 'Bonjour aujourd'hui c'est le : '
cours=# "Aujourd'hui", NOW() "Date";
```

Aujourd'hui	Date
Bonjour aujourd'hui c'est le :	2014-09-18 18:28:13.831+02

```
cours=> SELECT 'Bonjour aujourd'hui c'est le : '
cours-> "Aujourd'hui", CURRENT_DATE "Date";
```

Aujourd'hui	Date
Bonjour aujourd'hui c'est le :	2016-03-04

(1 ligne)

Si vous voulez afficher uniquement la date sans l'heure vous pouvez utiliser la pseudo-colonne « **CURRENT_DATE** ».

Opérateur de concaténation

```
SELECT NOM || ' ' || PRENOM "Employé"
FROM EMPLOYES;
```

EMPLOYES

NO_EMPLOYE	NOM	PRENOM	...
	Employé		

	Fuller Andrew		
	Buchanan Steven		
	Peacock Margaret		
	Leverling Janet		
	Davolio Nancy		
	Dodsworth Anne		
	King Robert		
	Suyama Michael		
	Callahan Laura		
9 ligne(s) sélectionnée(s).			

Post

La concaténation est le seul opérateur disponible des chaînes de caractères. Le résultat d'une concaténation est la chaîne de caractères obtenue en mettant bout à bout les deux chaînes de caractères passées en arguments.

Cet opérateur se note au moyen de deux caractères barre verticale accolés « || », selon la syntaxe présente dans l'exemple suivant. Une projection partielle de la table EMPLOYES, pour extraire une chaîne de caractères qui résulte de la concaténation du numéro employé, nom, prénom et date de naissance.



```
cours=# SELECT NO_EMPLOYE || ' -- ' || NOM || ' -- ' || PRENOM ||
cours=# ' -- ' || DATE_NAISSANCE "Liste des employés"
cours=# FROM EMPLOYES ;
               Liste des employés
-----
14 -- Fuller -- Andrew -- 1984-06-27
24 -- Buchanan -- Steven -- 1972-11-26
95 -- Leger -- Pierre -- 1989-09-01
11 -- Belin -- Chantal -- 1986-09-15
33 -- Chambaud -- Axelle -- 1977-11-18
...
```



L'opérateur de concaténation peut travailler avec des expressions qui retournent une chaîne de caractère, un numérique ou une date, des constantes de type chaînes de caractères et numériques; les conversions entre ces différents types sont effectuées implicitement.

Opérateurs arithmétiques

Opérateur	Description	Exemple	Résultat
+	addition	2 + 3	5
-	soustraction	2 - 3	-1
*	multiplication	2 * 3	6
/	division (la division entière tronque les résultats)	4 / 2	2
%	modulo (reste)	5 % 4	1
^	exposant (association de gauche à droite)	2.0 ^ 3.0	8
/	racine carrée	/ 25.0	5
/	racine cubique	/ 27.0	3
!	factoriel	5 !	120
!!	factoriel (opérateur préfixe)	!! 5	120
@	valeur absolue	@ -5.0	5
&	AND bit à bit	91 & 15	11
	OR bit à bit	32 3	35
#	XOR bit à bit	17 # 5	20
~	NOT bit à bit	~1	-2
<<	décalage gauche	1 << 4	16
>>	décalage droit	8 >> 2	2

Une expression arithmétique peut comporter plusieurs opérateurs. Dans ce cas, le résultat de l'expression peut varier selon l'ordre dans lequel les opérations sont effectuées.

La priorité des opérateurs :

- La multiplication et la division sont prioritaires par rapport à l'addition et à la soustraction.
- Les opérateurs de même priorité sont évalués de la gauche vers la droite.
- Les parenthèses sont utilisées pour forcer la priorité de l'évaluation et pour clarifier les instructions SQL.

L'exemple suivant illustre une projection de la table `PRODUIT` pour extraire le nom du produit, la valeur du stock et la valeur commandée.



```
cours=> select nom_produit "Produit",
cours-> prix_unitaire*unites_stock "Stock",
cours-> unites_commandees*12 "Commandes"
cours-> from produits ;
```

Produit	Stock	Commandes
Tea	100.00	240
Pears	40.00	120

Peaches		40.00		120
Pineapple		40.00		120
Cherry Pie Filling		40.00		120
Green Beans		40.00		120
Corn		40.00		120
Peas		40.00		120
Tuna Fish		24.50		360
Smoked Salmon		100.00		360
Hot Cereal		180.00		600
Vegetable Soup		25.00		1200
Chicken Soup		60.00		1080
Chai		3510.00		
Chang		1615.00		480
Aniseed Syrup		650.00		840
Chef Anton's Cajun Seasoning		5830.00		
Grandma's Boysenberry Spread		15000.00		
...				



Les constantes numériques sont saisies avec ou sans signe sans espace entre les caractères et le caractère de séparation des décimales est le point « . ».

L'exemple suivant illustre une projection de la table EMPLOYES pour extraire le nom de l'employé et une prévision de salaire à la suite d'une augmentation de 10%.



```
cours=# SELECT NOM || ' ' || PRENOM "Employé",
cours-# SALAIRE * 1.1 "Nouveau Salaire"
cours-# FROM EMPLOYES ;
```

Employé	Nouveau Salaire
Fuller Andrew	105600.000
Buchanan Steven	14300.000
Leger Pierre	20900.000
Belin Chantal	11000.000
Chambaud Axelle	13200.000
...	

Opérateurs de type DATE

```
SELECT NOM, (SYSDATE - DATE_NAISSANCE) / 365
FROM EMPLOYES;
```

EMPLOYES

NO EMPLOYEE	NOM	DATE_NAISSANCE	...
	NOM	(SYSDATE-DATE_NAISSANCE) / 365	
	Fuller	50,2678913	
	Buchanan	47,2295352	
	Peacock	43,68159	
	Leverling	38,7336448	
	Davolio	33,4541927	
	Dodsworth	32,8898091	
	King	41,9884393	
	Suyama	38,8952886	
	Callahan	44,3747406	

PostgreSQL prise en main

Module 1 : Présentation de l'environnement - Diapo 1.40

SQL propose deux opérations possibles des expressions de type date.

L'ajout d'un nombre de jours à une date, le résultat étant une expression de type date.

DATE1 (+ ou -) NOMBRE = DATE2

Le calcul du nombre de jours séparant les deux dates, le résultat étant une expression de type numérique.

DATE1 - DATE2 = NOMBRE

Le résultat peut être exprimé sous forme de valeur décimale si les valeurs de DATE1 et/ou de DATE2 contiennent une notion d'heure.

« **SYSDATE** » est une pseudocolonne que l'on peut utiliser dans une expression de type date et qui a pour valeur la date et l'heure courantes du système d'exploitation hôte.



```
cours=# SELECT NOM,DATE_NAISSANCE,DATE_NAISSANCE+1,
```

```
cours=# NOW()-DATE_NAISSANCE
```

```
cours=# FROM EMPLOYES ;
```

nom	date_naissance	?column?	?column?
Fuller	1984-06-27	1984-06-28	11040 days 20:50:33.41
Buchanan	1972-11-26	1972-11-27	15271 days 19:50:33.41
Leger	1989-09-01	1989-09-02	9148 days 20:50:33.41
Belin	1986-09-15	1986-09-16	10230 days 20:50:33.41
Chambaud	1977-11-18	1977-11-19	13453 days 19:50:33.41
Ragon	1989-05-21	1989-05-22	9251 days 20:50:33.41
Splینگart	1978-09-11	1978-09-12	13156 days 20:50:33.41
...			

Le traitement de la valeur NULL

Lorsque l'un des termes d'une expression a la valeur « **NULL** », l'expression entière prend la valeur « **NULL** » ; pour pouvoir travailler avec des champs qui contiennent des valeurs « **NULL** », il faut une fonction qui puisse gérer cette valeur.

COALESCE

L'instruction « **COALESCE** » permet de retourner la première expression « **NOT NULL** » de la liste des paramètres.

```
COALESCE ( EXPRESSION1, EXPRESSION2 [,...] ) ;
```

CASE

L'instruction « **CASE** » permet de mettre en place une condition d'instruction conditionnelle « **IF..THEN..ELSE** » directement dans une requête.

```
CASE  
    WHEN CONDITION1 THEN RESULTAT1  
    [WHEN CONDITION2 THEN RESULTAT2,...]  
    [ELSE RESULTAT]  
END ;
```

CONDITION L'argument **CONDITION** est une expression logique.



```
cours=# select NOM, SALAIRE,  
cours=# CASE WHEN COMMISSION IS NULL THEN 0  
cours=# ELSE COMMISSION END,  
cours=# COALESCE (COMMISSION, 0)  
cours=# FROM EMPLOYES ;
```

nom	salaire	commission	commission
Fuller	96000.00	0	0
Buchanan	13000.00	12940.00	12940.00
Leger	19000.00	11150.00	11150.00
Belin	10000.00	10640.00	10640.00
Chambaud	12000.00	11600.00	11600.00
Ragon	13000.00	5980.00	5980.00
Splingart	16000.00	16480.00	16480.00
Brasseur	147000.00	0	0
Poupard	1800.00	0	0
...			

La clause LIMIT

```
SELECT PRIX_UNITAIRE*100
FROM PRODUITS LIMIT 7;
```

PRODUITS

NOM PRODUIT	NO FOURNISSEUR	PRIX UNITAIRE	...

PRIX_UNITAIRE*100	

	27500
	9000
	9500
	5000
	11000
	10675
	12500

A l'aide de la clause « **LIMIT** » il est possible de retourner un nombre des lignes limités au moment de l'exécution de la requête à l'aide de la syntaxe suivante :

LIMIT nbLignes [OFFSET rangDépart]



```
mysql> SELECT SOCIETE, IFNULL(FAX, 'Non affecté')
-> FROM CLIENTS LIMIT 5;
```

```
cours=# SELECT NOM, PRENOM FROM EMPLOYES LIMIT 5;
```

```
nom      | prenom
-----+-----
Fuller   | Andrew
Buchanan | Steven
Leger    | Pierre
Belin     | Chantal
Chambaud | Axelle
(5 lignes)
```

```
cours=# SELECT NOM, PRENOM FROM EMPLOYES LIMIT 5 OFFSET 2;
```

```
nom      | prenom
-----+-----
Leger     | Pierre
Belin      | Chantal
Chambaud   | Axelle
Ragon      | Andr|®
Splingart  | Lydia
(5 lignes)
```

Atelier 3.2



Atelier 3.2

- La concaténation
- Les opérateurs



Durée : 10 minutes

SQL pour MySQL et PostgreSQL

Module 1 : Présentation de l'environnement - Diapo 1.43

Exercice n° 1 La concaténation

Respectant les formats des modèles suivants, écrivez les requêtes vous permettant d'afficher :

- Le nom de l'employé et ses revenus annuels : commission + salaire * 12.

Employé	a un	gain annuel	sur 12 mois
-----	-----	-----	-----
Fuller	gagne	120000	par an.
Buchanan	gagne	96000	par an.
...			

- Le nom et le prénom de l'employé et sa fonction.

Employé	-----
Callahan Laura	est Assistante commerciale de cette société.
Buchanan Steven	est Chef des ventes de cette société.
...	

Exercice n° 2 Les opérateurs

Créez les requêtes vous permettant d'afficher :

- Les produits commercialisés, la valeur du stock par produit et la valeur des produits commandés. Dans la table PRODUITS, vous trouvez les champs UNITES_STOCK et UNITES_COMMANDEES que vous multipliez par le PRIX_UNITAIRE pour retrouver les valeurs des deux stocks.
- Le nom, le prénom, l'âge et l'ancienneté des employés, dans la société.

Tri du résultat d'une requête

Les lignes constituant le tableau résultat d'un ordre « **SELECT** » sont affichées dans un ordre indéterminé qui dépend des algorithmes internes du moteur du système de gestion de bases de données relationnelles.

En revanche, on peut, dans l'ordre « **SELECT** », demander que le résultat soit trié avant l'affichage selon un ordre ascendant ou descendant, en fonction d'un ou de plusieurs critères. Il est possible d'utiliser jusqu'à 16 critères de tri.

Les critères de tri sont spécifiés dans une clause « **ORDER BY** », figurant en dernière position de l'ordre « **SELECT** ».

La syntaxe de l'instruction « **SELECT** » :

```
SELECT [ALL | DISTINCT]{*,[EXPRESSION1 [AS] ALIAS1[,...]]}
FROM NOM_TABLE
WHERE PREDICAT
ORDER BY [NOM_COLONNE1|POSITION1|EXPRESSION1] [ASC|DESC],
        [NOM_COLONNE2|POSITION2|EXPRESSION2] [ASC|DESC]
        [,...] ;
```

NOM_COLONNE	Le nom de la colonne qui fournit la valeur qui entre en ligne de compte pour le tri. La colonne peut ou non faire partie des colonnes extraites par la requête mais elle doit être une des colonnes des tables mentionnées dans FROM.
EXPRESSION	L'expression ou l'alias de l'expression qui fournit la valeur qui entre en ligne de compte pour le tri.
POSITION	L'expression ou la colonne, identifiés par la position dans la clause « SELECT », qui fournit la valeur qui entre en ligne de compte pour le tri.
ASC	Le critère de tri est ascendant pour NOM_COLONNE ou EXPRESSION ou POSITION qui précède le critère. Les critères sont définis pour chaque expression si vous ne le précisez pas. Par défaut, il est ascendant.
DESC	Le critère de tri est descendant pour NOM_COLONNE ou EXPRESSION ou POSITION qui précède le critère. Par défaut, il est ascendant.

Note

Le tri se fait d'abord selon le premier critère spécifié dans la clause « **ORDER BY** », puis les lignes ayant la même valeur pour le premier critère sont triées selon le deuxième critère de la clause « **ORDER BY** », etc.

La requête suivante est une sélection de la table **PRODUIT** pour extraire les produits, les fournisseurs et les catégories de produits avec les résultats ordonnés par fournisseur et catégorie produits.

```
cours=# SELECT NOM_PRODUIT, NO_FOURNISSEUR, CODE_CATEGORIE
cours=# FROM PRODUITS
cours=# ORDER BY NO_FOURNISSEUR, CODE_CATEGORIE DESC;
          nom_produit              | no_fournisseur | code_categorie
```

Aniseed Syrup	1	2
Chai	1	1
Chang	1	1
Cherry Pie Filling	2	9
Amandes	2	7
Brownie Mix	2	3
Louisiana Fiery Hot Pepper Sauce	2	2
Louisiana Hot Spiced Okra	2	2
Chef Anton's Gumbo Mix	2	2
Chef Anton's Cajun Seasoning	2	2
Uncle Bob's Organic Dried Pears	3	7
Northwoods Cranberry Sauce	3	2
Grandma's Boysenberry Spread	3	2
Corn	4	9
Ikura	4	8
Longlife Tofu	4	7
Long Grain Rice	4	7
...		

PostgreSQL



PostgreSQL traite les valeurs « **NULL** » comme si elles étaient des valeurs infinies; on peut donc ajouter une clause « **ORDER BY** » avec les mots clés « **NULLS FIRST** » et « **NULLS LAST** ».

```
cours=# SELECT NOM,PRENOM,SALAIRE,COMMISSION
cours=# FROM EMPLOYES
cours=# WHERE SALAIRE > 15000
cours=# ORDER BY COMMISSION;
```

nom	prenom	salaire	commission
Leger	Pierre	19000.00	11150.00
Splingart	Lydia	16000.00	16480.00
Fuller	Andrew	96000.00	
Brasseur	Hervé	147000.00	
Giroux	Jean-Claude	150000.00	

(5 lignes)

```
cours=# SELECT NOM,PRENOM,SALAIRE,COMMISSION
cours=# FROM EMPLOYES
cours=# WHERE SALAIRE > 15000
cours=# ORDER BY COMMISSION NULLS FIRST;
```

nom	prenom	salaire	commission
Fuller	Andrew	96000.00	
Brasseur	Hervé	147000.00	
Giroux	Jean-Claude	150000.00	
Leger	Pierre	19000.00	11150.00
Splingart	Lydia	16000.00	16480.00

(5 lignes)

Atelier 3.3



Atelier 3.3

- Les ordres de tri
- Les pseudocolonnes et la table DUAL



Durée : 10 minutes

SQL pour MySQL et PostgreSQL

Module 1 : Présentation de l'environnement - Diapo 1.47

Exercice n° 1 Les ordres de tri

Écrivez les requêtes permettant d'afficher :

- Les employés par ordre alphabétique.
- Les employés depuis le plus récemment embauché jusqu'au plus ancien.
- Les fournisseurs dans l'ordre alphabétique de leur pays et ville de résidence.
- Les employés par ordre alphabétique de leur fonction et du plus grand salaire au plus petit.
- Les employés dans l'ordre de leur commission.

La sélection ou restriction

L'ordre « **SELECT** » permet de spécifier les lignes à sélectionner par utilisation de la clause « **WHERE** ». Cette clause est suivie de la condition de sélection, évaluée pour chaque ligne de la table. Seules les lignes pour lesquelles la condition est vérifiée sont sélectionnées.

La syntaxe de l'instruction « **SELECT** » :

```
SELECT [ALL | DISTINCT]{*,[EXPRESSION1 [AS] ALIAS1[,...]]}
FROM NOM_TABLE WHERE PREDICAT ;
```

Opérateur	Description
<	inférieur à
>	supérieur à
<=	inférieur ou égal à
>=	supérieur ou égal à
=	égal à
<> ou !=	différent de

La requête suivante est une sélection de la table **CLIENTS** pour extraire la société et l'adresse des clients localisés à Paris.



```
cours=> SELECT SOCIETE,ADRESSE FROM CLIENTS WHERE PAYS='France' ;
```

societe	adresse
Du monde entier	67, rue des Cinquante Otages
Folies gourmandes	184, chaussée de Tournai
France restauration	54, rue Royale
La corne d'abondance	67, avenue de l'Europe
La maison d'Asie	1 rue Alsace-Lorraine
Paris spécialités	265, boulevard Charonne
Spécialités du monde	25, rue Lauriston
Victuailles en stock	2, rue du Commerce
Vins et alcools Chevalier	59 rue de l'Abbaye
Blondel père et fils	24, place Kléber
Bon app'	12, rue des Bouchers

(11 lignes)

La requête suivante est une sélection de la table **EMPLOYES** pour extraire les employés embauchés en '17/10/93'.



```
cours=# SELECT NOM, PRENOM, DATE_EMBAUCHE
cours=# FROM EMPLOYES WHERE DATE_EMBAUCHE = '1990-07-11';
```

nom	prenom	date_embauche
Poupard	Claudette	1990-07-11

La requête suivante est une sélection de la table **EMPLOYES** pour extraire les employés encadrés par l'employé numéro 37.



```
cours=# SELECT NOM, PRENOM, REND_COMPTE FROM EMPLOYES
cours=# WHERE REND_COMPTE = 37;
```

nom	prenom	rend_compte
-----	--------	-------------

Fuller	Andrew	37
Brasseur	Hervé	37

La requête suivante est une sélection de la table EMPLOYES pour extraire le nom et le prénom des employés qui ont été embauchés après '31/12/93'.



```
cours=> SELECT NOM, PRENOM, DATE_EмбаUCHE FROM EMPLOYES
cours->WHERE DATE_EмбаUCHE > '2003-01-30';
```

nom	prenom	date_embauche
Fuller	Andrew	2003-08-09
Lamarre	Eric	2003-03-22
Courty	Jean-Louis	2003-04-16
Pagani	Hector	2003-07-02
Chaussende	Maurice	2003-08-10
Leverling	Janet	2003-04-30
Cremel	Brigitte	2003-08-28
Gregoire	Renée	2003-07-19
Lefebvre	Michel	2003-03-19
Di Clemente	Luc	2003-07-06
Jacquot	Philippe	2003-07-30
Dodsworth	Anne	2003-04-12
Rollet	Philippe	2003-03-07
Coutou	Myriam	2003-04-29
Marielle	Michel	2003-07-30
Espeche	Eric	2003-02-17
Ziliox	Francoise	2003-02-02
Lampis	Gabrielle	2003-05-29
Regner	Charles	2003-09-07

(19 lignes)

La requête suivante est une sélection de la table CLIENTS pour extraire les sociétés qui ont un code inférieur à 'B'.



```
cours=> SELECT CODE_CLIENT, SOCIETE FROM CLIENTS
cours-> WHERE CODE_CLIENT < 'B';
```

code_client	societe
ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno Taquería
AROUT	Around the Horn

(4 lignes)

La requête suivante est une sélection de la table EMPLOYES pour extraire le nom, le prénom et la fonction des employés qui ne sont pas des représentants.



```
cours=> SELECT NOM,PRENOM,FONCTION FROM EMPLOYES
cours-> WHERE FONCTION <>'Représentant(e)' LIMIT 3;
```

nom	prenom	fonction
Giroux	Jean-Claude	Président
Fuller	Andrew	Vice-Président
Brasseur	Hervé	Vice-Président

(3 lignes)

L'opérateur LIKE

```
SELECT QUANTITE
FROM PRODUITS
WHERE QUANTITE LIKE '%pièces%';
```

PRODUITS

[illegible]

LIKE

L'opérateur « **LIKE** » est très utile pour effectuer des recherches dans des chaînes alphanumériques.

Il utilise deux caractères spéciaux pour signifier le type de correspondance recherchée :

- un signe pourcentage « % », appelé **caractère générique**,
- et un caractère de soulignement « _ », appelé **marqueur de position**.

Le **caractère générique** placé dans une chaîne remplace une chaîne quelconque de caractères d'une longueur de zéro à n caractères.

Le **marqueur de position** placé dans une chaîne remplace un caractère quelconque mais impose l'existence de ce caractère.

```
EXPRESSION LIKE 'Chaîne de caractères avec des caractères
spéciaux'
```

La requête suivante est une sélection de la table `PRODUITS` pour extraire les noms du produit et la quantité des produits qui estiment leur quantité en boîtes et en kg.

```
cours=> SELECT NOM_PRODUIT, QUANTITE FROM PRODUITS
cours-> WHERE QUANTITE LIKE '%boîtes%kg%' ;
```

nom_produit	quantite
Konbu	1 boîtes (2 kg)
Alice Mutton	20 boîtes (1 kg)
Filo Mix	16 boîtes (2 kg)
Long Grain Rice	16 boîtes de 2 kg



Dans l'exemple précédent vous pouvez constater que les enregistrements extraits contiennent dans la chaîne de caractère `QUANTITE` deux chaînes de caractères la première 'boîtes' et la deuxième 'kg'.



Les valeurs contenues dans les colonnes sont sensibles à la case (majuscule, minuscule), les informations saisies dans les chaînes de caractères de comparaison doivent l'être aussi.

La requête suivante est une sélection de la table `PRODUITS` pour extraire les quantités des produits qui dans la colonne `QUANTITE` ont un '1' en première position et un '0' en troisième position.



```
cours=> SELECT QUANTITE FROM PRODUITS WHERE QUANTITE LIKE '1_0%' ;
          quantite
-----
100 unités par boîte
140 g
140 g
100 sacs (250 g)
100 pièces (100 g)
(5 lignes)
```

La requête suivante est une sélection de la table `PRODUITS` pour extraire les noms du produit et la quantité des produits qui dans la colonne `QUANTITE` commencent par trois caractères quelconques et finissant par 'pièces'.



```
cours=> SELECT NOM_PRODUIT,QUANTITE FROM PRODUITS
cours-> WHERE QUANTITE LIKE '___pièces' ;
          nom_produit      | quantite
-----+-----
Perth Pasties              | 48 pièces
Escargots de Bourgogne    | 24 pièces
(2 lignes)
```


La requête suivante est une sélection de la table `PRODUITS` pour extraire toutes les lignes dont la colonne `QUANTITE` commence par la chaîne '10' et finit par la chaîne 'pièces'.



```
cours=> SELECT QUANTITE FROM PRODUITS
cours-> WHERE QUANTITE ~ '^(10).*(pièces)$';
          quantite
-----
10 boîtes x 12 pièces
10 sacs x 8 pièces
10 boîtes de 12 pièces
(3 lignes)
```

La requête suivante est une sélection de la table `PRODUITS` pour extraire toutes les lignes dont la colonne `QUANTITE` contient n'importe quel caractère ',', ou 'x'.



```
cours=> SELECT QUANTITE FROM PRODUITS
cours-> WHERE QUANTITE ~ '[,x]';
          quantite
-----
10 boîtes x 20 sacs
10 boîtes x 12 pièces
24 cartons x 4 pièces
24 boîtes x 2 tartes
50 sacs x 30 saucisses
20 sacs x 4 pièces
10 sacs x 8 pièces
24 bouteilles (0,5 litre)
48 bocaux de 170 g
12 bocaux de 225 g
12 bocaux de 340 g
24 bocaux de 225 g
(12 lignes)
```

La requête suivante est une sélection de la table `PRODUITS` pour extraire toutes les lignes dont la colonne `QUANTITE` contient la chaîne 'sacs' et la chaîne 'pièces'.



```
cours=> SELECT QUANTITE FROM PRODUITS
cours-> WHERE QUANTITE ~ '(sacs).*(pièces)';
          quantite
-----
20 sacs x 4 pièces
10 sacs x 8 pièces
(2 lignes)
```

La requête suivante est une sélection de la table `PRODUITS` pour extraire toutes les lignes dont la colonne `QUANTITE` contient la chaîne 'carton' ou la chaîne 'pièces' ou la chaîne 'bouteilles' et commence par le caractère '2'.



```
cours=> SELECT QUANTITE FROM PRODUITS
cours-> WHERE QUANTITE ~ '^2.*(carton|pièces|bouteilles)';
          quantite
-----
24 bouteilles (1 litre)
24 cartons x 4 pièces
24 cartons (500 g)
24 cartons (200 g)
24 bouteilles (70 cl)
```

```

24 bouteilles (1 litre)
24 cartons (50 g)
24 cartons (250 g)
24 bouteilles (12 onces)
24 bouteilles (355 ml)
24 cartons (200 g)
24 bouteilles (250 ml)
20 cartons (2 kg)
24 cartons (250 g)
24 pièces
24 bouteilles (500 ml)
20 sacs x 4 pièces
24 bouteilles (0,5 litre)
24 paquets de 4 pièces
24 bouteilles de 35 cl
(20 lignes)

```

La requête suivante est une sélection de la table `PRODUITS` pour extraire toutes les lignes dont la colonne `QUANTITE` contient deux occurrences de la chaîne '100'.



```

cours=> SELECT QUANTITE FROM PRODUITS
cours-> WHERE QUANTITE ~ '(100.*){2}';
      quantite
-----
100 pièces (100 g)
(1 ligne)

```

La requête suivante est une sélection de la table `PRODUITS` pour extraire toutes les lignes dont la colonne `QUANTITE` contient deux occurrences successives du caractère 'l' et deux occurrences successives du numéro '0'.



```

cours=> SELECT QUANTITE FROM PRODUITS
cours-> WHERE QUANTITE ~ 'l{2,}.*0{2}';
      quantite
-----
1 bouteille (500 ml)
24 bouteilles (500 ml)
(2 lignes)

```

La requête suivante est une sélection de la table `PRODUITS` pour extraire toutes les lignes dont la colonne `QUANTITE` ne commence pas par un des numéros de la liste.



```

cours=> SELECT QUANTITE FROM PRODUITS
cours-> WHERE QUANTITE ~ '(100.*){2}';
      quantite

```

L'opérateur IS NULL

Oracle permet d'employer des opérateurs logiques, « = », « != », etc., avec « NULL » mais ce type de comparaison ne retourne généralement pas des résultats très parlants.

IS NULL

L'opérateur logique « **IS NULL** » vérifie si la valeur retournée par **EXPRESSION** est égale à « **NULL** » ; alors il retourne **VRAI**, sinon **FAUX**.

EXPRESSION IS NULL

La requête suivante est une sélection de la table **PRODUITS** pour extraire les qui n'ont pas de quantité renseignée.



```
cours=> SELECT REF_PRODUIT, NOM_PRODUIT from produits
cours-> WHERE QUANTITE = NULL;
  ref_produit | nom_produit
-----+-----
(0 ligne)
```

```
cours=> SELECT REF_PRODUIT, NOM_PRODUIT from produits
cours-> WHERE QUANTITE IS NULL;
  ref_produit | nom_produit
-----+-----
          118 | Hot Cereal
          104 | Granola
          105 | Potato Chips
(3 lignes)
```



Les opérateurs logiques « **IS NULL** » et « **IS NOT NULL** » peuvent être utilisés pour tous les types de données qui sont stockés dans la base.

Atelier 4.1



Atelier 4.1

- La restriction
- Le traitement des chaînes de caractères
- Le traitement de valeurs NULL



Durée : 35 minutes

SQL pour MySQL et PostgreSQL

Module 1 : Présentation de l'environnement - Diapo 1.56

Exercice n° 1 La restriction

Écrivez les requêtes permettant d'afficher :

- Le nom de la société et de la localité des clients qui habitent à Toulouse.
- Le nom, le prénom et la fonction des employés qui ne sont pas des représentants.
- Le nom du produit, la catégorie et le fournisseur des produits qui ne sont pas disponibles, le champ `INDISPONIBLE` est égal à 1.
- Le nom, prénom et fonction des employés qui ont un salaire inférieur à 3500.
- Le nom, prénom et fonction des employés dirigés par l'employé numéro 86.
- Le nom, prénom et fonction des employés recrutés après 01/01/2003.

Exercice n° 2 Le traitement des chaînes de caractères

Écrivez les requêtes permettant d'afficher :

- Les produits et leur quantité conditionnée en bouteilles d'un litre.
- Le nom de la société, la localité et le code postal des fournisseurs à condition que leur code postal soit composé uniquement des valeurs numériques.
- Les produits et leur quantité à condition que leur emballage soit de type cartons, boîtes ou unités et conditionnée par paquets de 24 ou 32.

Exercice n° 3 Le traitement de valeurs NULL

Écrivez les requêtes permettant d'afficher :

- Le nom de la société, la ville et le pays des clients qui n'ont pas de numéro de fax renseigné.
- Le nom, prénom et la fonction des employés qui ne sont pas commissionnés.
- Le nom, prénom et la fonction des employés qui n'ont pas de supérieur hiérarchique.
- Le nom de la société, la ville et le pays des fournisseurs qui ont un numéro de fax renseigné.

Les opérateurs BETWEEN et IN

Il existe également des opérateurs logiques qui permettent d'effectuer des comparaisons avec des listes de valeurs, comme décrit dans la présentation.

BETWEEN

L'opérateur logique « **BETWEEN** » vérifie si la valeur retournée par **EXPRESSION1** est égale à **EXPRESSION2**, **EXPRESSION3** ou toute valeur comprise entre **EXPRESSION2** et **EXPRESSION3** ; alors retourne **VRAI** sinon **FAUX**.

EXPRESSION1 BETWEEN EXPRESSION2 AND EXPRESSION3

La requête suivante est une sélection de la table **EMPLOYES** pour extraire le nom, le prénom et le salaire des employés qui ont un salaire compris entre 2500 et 3500.



```
cours=> SELECT NOM, PRENOM, SALAIRE FROM EMPLOYES
cours-> WHERE SALAIRE BETWEEN 10000 AND 15000;
```

nom	prenom	salaire
Buchanan	Steven	13000.00
Belin	Chantal	10000.00
Chambaud	Axelle	12000.00
Ragon	André	13000.00
Cheutin	Corinne	10000.00

(5 lignes)

IN

L'opérateur logique « **IN** » vérifie si la valeur retournée par **EXPRESSION1** est dans la **LISTE_DE_VALEURS** ; alors il retourne **VRAI**, sinon **FAUX**.

EXPRESSION1 IN (LISTE_DE_VALEURS)

LISTE_DE_VALEURS La liste des valeurs peut être une liste de constantes ou une liste de valeurs dynamiques (une sous-requête ; le traitement des sous-requêtes est présenté plus loin dans ce module) ; cependant les types de données des différentes constantes doivent être identiques au type retourné par **EXPRESSION1**.

La requête suivante est une sélection de la table **CLIENTS** pour extraire la société et la ville de résidence des clients situés à Paris, Strasbourg et Toulouse.



```
cours=> SELECT SOCIETE, VILLE FROM CLIENTS
cours-> WHERE VILLE IN ('Paris','Strasbourg','Toulouse') ;
```

societe	ville
La maison d'Asie	Toulouse
Paris spécialités	Paris
Spécialités du monde	Paris
Blondel père et fils	Strasbourg

(4 lignes)

Les opérateurs AND et OR

Les opérateurs logiques forment des expressions de type logique et ces expressions peuvent être combinées à l'aide des opérateurs logiques « **AND** », « **OR** » ou « **NOT** ».

AND

L'opérateur logique « **AND** » vérifie si **EXPRESSION1** et **EXPRESSION2** sont **VRAI** en même temps ; alors il retourne **VRAI**, sinon **FAUX**.

EXPRESSION1 AND EXPRESSION2

La requête suivante est une sélection de la table **PRODUIT** pour extraire les produits qui sont en stock et qui sont de type boîte.



```
cours=> SELECT REF_PRODUIT,QUANTITE,UNITES_STOCK FROM PRODUITS
cours-> WHERE UNITES_STOCK > 0 AND QUANTITE LIKE '12%boîte%' ;
```

ref_produit	quantite	unites_stock
77	12 boîtes	32
91	12 boîtes de 340 g	40
101	12 boîtes	60

(3 lignes)

La requête suivante est une sélection de la table **COMMANDES** pour extraire les commandes du client '**LEHMS**' passées par les employés de '**4-40**' et dont la date de commandes est supérieure à premier avril.



```
cours=> SELECT NO_COMMANDE, CODE_CLIENT, NO_EMPLOYE, DATE_ENVOI
cours-> FROM COMMANDES
cours-> WHERE CODE_CLIENT = 'LEHMS' AND
cours-> NO_EMPLOYE BETWEEN 4 AND 40 AND
cours-> DATE_COMMANDE > '2011-04-01';
```

no_commande	code_client	no_employe	date_envoi
226411	LEHMS	4	2011-05-12
226461	LEHMS	4	2011-05-25
227109	LEHMS	40	2011-06-15

(3 lignes)

La requête suivante est une sélection de la table **PRODUITS** pour extraire les produits du fournisseur numéro '1' et du fournisseur numéro '2'.



```
cours=> SELECT NOM_PRODUIT, CODE_CATEGORIE FROM PRODUITS
cours-> WHERE NO_FOURNISSEUR = 1 AND
cours-> NO_FOURNISSEUR = 5;
```

nom_produit	code_categorie
-------------	----------------

(0 ligne)

Attention



Il faut faire très attention aux abus de langage comme dans l'exemple précédent où l'on souhaite afficher les produits du fournisseur numéro '1' ou du fournisseur numéro '2'.

L'ensemble des conditions de la clause « **WHERE** » est exécuté pour valider chaque enregistrement de la table. Par conséquent le numéro fournisseur ne peut pas être '1' et '2' à la fois et pour le même enregistrement.

OR

L'opérateur logique « **OR** » vérifie si au moins une des deux est VRAI ; alors il retourne VRAI, sinon FAUX.

EXPRESSION1 OR EXPRESSION2



```
cours=>SELECT REF_PRODUIT P, QUANTITE Q,UNITES_STOCK US,
cours-> UNITES_COMMANDEES UC FROM PRODUITS
cours-> WHERE UNITES_STOCK IS NULL OR
cours->         UNITES_COMMANDEES IS NULL OR
cours->         QUANTITE IS NULL ;
```

p	q	us	uc
118		60	50
1	10 boîtes x 20 sacs	39	
4	48 pots (6 onces)	53	
6	12 pots (8 onces)	120	
7	12 cartons (1 kg)	15	
8	12 pots (12 onces)	6	
9	18 cartons (500 g)		
10	12 pots (200 g)	31	
...			

Astuce



L'opérateur logique « **AND** » est prioritaire par rapport à l'opérateur « **OR** ». Des parenthèses peuvent être utilisées pour imposer une priorité dans l'évaluation de l'expression, ou tout simplement pour rendre l'expression plus claire.



```
cours=> SELECT NO_EMPLOYE, REND_COMPTE, FONCTION, TITRE
cours-> FROM EMPLOYES
cours-> WHERE NO_EMPLOYE BETWEEN 30 AND 37
cours->         AND REND_COMPTE IS NOT NULL
cours->         AND(FONCTION LIKE 'Rep%' OR
cours->         TITRE LIKE 'M.');
```

no_employe	rend_compte	fonction	titre
31	11	Représentant(e)	M.
32	86	Représentant(e)	M.
34	86	Représentant(e)	Mlle
35	24	Représentant(e)	M.
36	11	Représentant(e)	Mme

(5 lignes)

```
cours=> SELECT NO_EMPLOYE, REND_COMPTE, FONCTION, TITRE
cours-> FROM EMPLOYES
```



```
cours-> WHERE NO_EMPLOYE BETWEEN 30 AND 37
cours-> AND REND_COMPTE IS NOT NULL
cours-> AND FONCTION LIKE 'Rep%' OR
cours-> TITRE LIKE 'M.';
no_employe | rend_compte | fonction | titre
-----+-----+-----+-----
37 | | Président | M.
14 | 37 | Vice-Président | M.
18 | 37 | Vice-Président | M.
24 | 14 | Chef des ventes | M.
95 | 18 | Chef des ventes | M.
86 | 18 | Chef des ventes | M.
1 | 86 | Représentant(e) | M.
4 | 11 | Représentant(e) | M.
5 | 95 | Représentant(e) | M.
7 | 23 | Représentant(e) | M.
8 | 11 | Représentant(e) | M.
9 | 24 | Représentant(e) | M.
10 | 23 | Représentant(e) | M.
13 | 86 | Représentant(e) | M.
15 | 33 | Représentant(e) | M.
16 | 33 | Représentant(e) | M.
...
(62 lignes)
```

L'opérateur NOT



- NOT EXPRESSION
- IS NOT NULL
- NOT LIKE
- NOT IN

NOT

L'opérateur logique « **NOT** » inverse le sens de **EXPRESSION**, explicitement si **EXPRESSION** est **FAUX** ; alors retourne il **VRAI**, sinon **FAUX**.

NOT EXPRESSION

La requête suivante est une sélection de la table **EMPLOYES** pour extraire le nom et la fonction des employés qui ont un salaire supérieur à 12500 et qui ne sont pas des représentants.



```
cours=# SELECT NOM, FONCTION FROM EMPLOYES
cours=# WHERE SALAIRE > 15000 AND
cours=#       NOT FONCTION LIKE 'Rep%';
```

nom	fonction
Giroux	Président
Fuller	Vice-Président
Brasseur	Vice-Président
Leger	Chef des ventes
Splingart	Chef des ventes

NOT IN

L'opérateur logique « **NOT IN** » vérifie si la valeur retournée par **EXPRESSION1** n'est pas dans la **LISTE_DE_VALEURS** ; alors il retourne **VRAI**, sinon **FAUX**.

EXPRESSION1 NOT IN (LISTE_DE_VALEURS)

```
cours=# SELECT NOM,PRENOM, TITRE FROM EMPLOYES
cours=# WHERE TITRE NOT IN ('M.', 'Mme');
```

nom	prenom	titre
-----	--------	-------



Davolio		Nancy		Mlle
Callahan		Laura		Mlle
Leverling		Janet		Mlle
Dodsworth		Anne		Mlle

La requête précédente est une sélection de la table `EMPLOYES` pour extraire le nom, prénom et titre des employés avec une valeur pour la colonne titre autre que 'M.', 'Mme' et 'Mlle'.

NOT BETWEEN

L'opérateur logique « **BETWEEN** » vérifie si la valeur retournée par `EXPRESSION1` n'est pas égale à `EXPRESSION2`, `EXPRESSION3` ou toute valeur comprise entre `EXPRESSION2` et `EXPRESSION3` ; alors retourne `VRAI` sinon `FAUX`.

`EXPRESSION1 NOT BETWEEN EXPRESSION2 AND EXPRESSION3`

La requête suivante est une sélection de la table `EMPLOYES` pour extraire le nom, le prénom et la date d'embauche des employés qui n'ont pas été recrutés en 1993.

IS NOT NULL

L'opérateur logique « **IS NOT NULL** » vérifie si la valeur retournée par `EXPRESSION` n'est pas égale à « **NULL** » ; alors retourne `VRAI`, sinon `FAUX`.

`EXPRESSION IS NOT NULL`

La requête suivante est une sélection de la table `EMPLOYES` pour extraire les noms et prénoms des employés qui ont une commission renseignée.

Atelier 4.2



Atelier 4.2

- L'opérateur BETWEEN
- La comparaison avec des listes
- L'assemblage des expressions



Durée : 25 minutes

SQL pour MySQL et PostgreSQL

Module 1 : Présentation de l'environnement - Diapo 1.63

Exercice n° 1 L'opérateur BETWEEN

Écrivez les requêtes permettant d'afficher :

- Le nom, prénom, fonction et salaire des employés qui ont un salaire compris entre 3500 et 6000.
- Le numéro de commande, code client et la date de commande pour les commandes passées entre le '01/01/2011' et '01/03/2011'.

Exercice n° 2 La comparaison avec des listes

Écrivez les requêtes permettant d'afficher :

- Le nom de la société, l'adresse, le téléphone et la ville des clients qui habitent à Toulouse, à Strasbourg, à Nantes ou à Marseille.
- Le nom du produit, le fournisseur, la catégorie et les quantités en stock pour les produits qui sont d'une des catégories 1, 3, 5 et 7.
- Le numéro de commande, code client et la date de commande pour les commandes passées dans une des dates : '18/02/2010', '20/02/2010' ou '25/02/2010'.

Exercice n° 3 L'assemblage des expressions

Écrivez les requêtes permettant d'afficher :

- Le nom, prénom, fonction et le salaire des représentants qui sont en activité depuis '10/10/2003'.

- Le nom, prénom, fonction et le salaire des employés qui sont âgés de plus de 45 ans ou qui ont une ancienneté de plus de 20 ans.
- Le nom du produit, le fournisseur, la catégorie et les quantités des produits qui ont le numéro fournisseur entre 1 et 3 ou un code catégorie entre 1 et 3 et pour lesquelles les quantités sont données en boîtes ou en cartons.
- Les produits et leur quantité à condition que leur emballage ne soit pas d'un de ces types : cartons, boîtes ou unités et qu'il ne soit pas conditionné par paquets de 24 ou 32. Il ne faut pas afficher les produits de catégorie 1, 4 et 8.

3

Interrogation des données

Types de données

PostgreSQL offre un large choix de types de données disponibles nativement. Les utilisateurs peuvent ajouter de nouveaux types à PostgreSQL en utilisant la commande CREATE TYPE.

Nom	Alias	Description
bigint	int8	Entier signé sur 8 octets
bigserial	serial8	Entier sur 8 octets à incrémentation automatique
bit[(n)]		Suite de bits de longueur fixe
bitvarying[(n)]	varbit	Suite de bits de longueur variable
boolean	bool	Booléen (Vrai/Faux)
box		Boîte rectangulaire dans le plan
bytea		Donnée binaire (« tableau d'octets »)
character[(n)]	char[(n)]	Chaîne de caractères de longueur fixe
charactervarying[(n)]	varchar[(n)]	Chaîne de caractères de longueur variable
cidr		Adresse réseau IPv4 ou IPv6
circle		Cercle dans le plan
date		Date du calendrier (année, mois, jour)
doubleprecision	float8	Nombre à virgule flottante de double précision (sur huit octets)
inet		Adresse d'ordinateur IPv4 ou IPv6
integer	int,int4	Entier signé sur 4 octets
interval[champs][(p)]		Intervalle de temps
json		Données texte JSON
jsonb		Données binaires JSON, décomposées
line		Droite (infinie) dans le plan
lseg		Segment de droite dans le plan
macaddr		Adresse MAC
money		Montant monétaire
numeric[(p,s)]	decimal[(p,s)]	Nombre exact dont la précision peut être précisée
path		Chemin géométrique dans le plan
pg_lsn		Séquence numérique de journal (Log Sequence Number)
point		Point géométrique dans le plan
polygon		Chemin géométrique fermé dans le plan
real	float4	Nombre à virgule flottante de simple précision (sur quatre octets)
smallint	int2	Entier signé sur 2 octets
smallserial	serial2	Entier sur 2 octets à incrémentation automatique
serial	serial4	Entier sur 4 octets à incrémentation automatique
text		Chaîne de caractères de longueur variable
time[(p)][withouttimezone]		Heure du jour (pas du fuseau horaire)
time[(p)]withtimezone	timetz	Heure du jour, avec fuseau horaire

<code>timestamp[(p)][withouttimezone]</code>		Date et heure (pas du fuseau horaire)
<code>timestamp[(p)withtimezone]</code>	<code>timestampz</code>	Date et heure, avec fuseau horaire
<code>tsquery</code>		requête pour la recherche plein texte
<code>tsvector</code>		document pour la recherche plein texte
<code>txid_snapshot</code>		image de l'identifiant de transaction au niveau utilisateur
<code>uuid</code>		identifiant unique universel
<code>xml</code>		données XML

PostgreSQL prise en main

```

cours=# SELECT LTRIM('*****Chaîne');
          ltrim
-----
*****Chaîne

cours=# SELECT LTRIM('A2BD1AC3BCD4-Chaîne', 'ABCD1234');
          ltrim
-----
-Chaîne

cours=# SELECT RTRIM('Chaîne*****', '');
          rtrim
-----
Chaîne
cours=# SELECT TRIM(BOTH '*' FROM '*****Chaîne**') A;
          a
-----
Chaîne

cours=# SELECT TRIM(TRAILING '1234' FROM '12344-Chaîne-A1234') A;
          a
-----
12344-Chaîne-A

```

SUBSTR / SUBSTRING

La fonction extrait de la chaîne de caractère une sous-chaîne à partir d'une position et longueur donnée.



```

cours=> SELECT NOM, SUBSTR(NOM,3,5) A,
cours-> SUBSTR(NOM,3,25) B,SUBSTR(NOM,3) C FROM EMPLOYES LIMIT 3;

```

nom	a	b	c
Giroux	roux	roux	roux
Fuller	ller	ller	ller
Brasseur	asseu	asseur	asseur

(3 lignes)

```
substring(chaîne, [from int] [for int])
```

```
substring(chaîne from modele)
```

```

cours=> SELECT QUANTITE,
cours-> SUBSTRING(QUANTITE FROM
cours-> '(carton|bouteilles)^[^()') SUBSTR
cours-> FROM PRODUITS
cours-> WHERE QUANTITE ~ '(carton|bouteilles)^[^()
cours-> LIMIT 10;

```

quantite	substr
24 bouteilles (1 litre)	bouteilles
12 bouteilles (550 ml)	bouteilles
12 cartons (1 kg)	carton
18 cartons (500 g)	carton
1 carton (1 kg)	carton
10 cartons (500 g)	carton
24 cartons x 4 pièces	carton
24 cartons (500 g)	carton

```

12 cartons (250 g)      | carton
24 cartons (200 g)      | carton
(10 lignes)

cours=> SELECT QUANTITE,
cours->      SUBSTRING(QUANTITE FROM
cours(>      '[\([].*[\)]') SUBSTR
cours-> FROM PRODUITS
cours-> WHERE QUANTITE ~ '[\([].*[\)]'
cours-> LIMIT 10;

```

quantite	substr
24 bouteilles (1 litre)	(1 litre)
12 bouteilles (550 ml)	(550 ml)
48 pots (6 onces)	(6 onces)
12 pots (8 onces)	(8 onces)
12 cartons (1 kg)	(1 kg)
12 pots (12 onces)	(12 onces)
18 cartons (500 g)	(500 g)
12 pots (200 g)	(200 g)
1 carton (1 kg)	(1 kg)
10 cartons (500 g)	(500 g)

(10 lignes)

STRPOS

La fonction recherche la première occurrence du caractère ou de la chaîne de caractères donnée.



```

cours=# SELECT QUANTITE,
cours=#      STRPOS(QUANTITE, ' ') A,
cours=#      STRPOS(QUANTITE, 'kg') B
cours=# FROM PRODUITS
cours=# WHERE QUANTITE LIKE '%kg%' LIMIT 5;

```

quantite	a	b
12 cartons (1 kg)	3	15
1 carton (1 kg)	2	13
1 boîtes (2 kg)	2	13
20 boîtes (1 kg)	3	14
32 cartons (1 kg)	3	15

REPLACE

La fonction « **REPLACE** » permet de remplacer dans la chaîne de caractères, toutes les séquences du caractère ou de la chaîne de caractères donnée.



```

cours=# SELECT REPLACE('JACK et JUE', 'J', ' ') REPLACE1,
cours=#      REPLACE('JACK et JUE', 'J', 'BL') REPLACE2;

```

replace1	replace2
ACK et UE	BLACK et BLUE

REGEXP_REPLACE

La fonction « **REGEXP_REPLACE** » permet de localiser et de remplacer toutes les séquences d'une sous-chaîne à l'intérieur d'une chaîne. L'avantage réside dans le fait

qu'il n'est pas nécessaire de citer la sous chaîne, mais qu'il suffit de la décrire à l'aide d'une expression régulière pour la localiser.



```
cours=> SELECT QUANTITE, REGEXP_REPLACE(QUANTITE,'\(.*\)'.*$',
cours->      '--XXXXXX--') "REGEXP_REPLACE"
cours-> FROM PRODUITS
cours-> WHERE QUANTITE ~ '[(].*[\)]'
cours-> LIMIT 10;
```

quantite	REGEXP_REPLACE
24 bouteilles (1 litre)	24 bouteilles --XXXXXX--
12 bouteilles (550 ml)	12 bouteilles --XXXXXX--
48 pots (6 onces)	48 pots --XXXXXX--
12 pots (8 onces)	12 pots --XXXXXX--
12 cartons (1 kg)	12 cartons --XXXXXX--
12 pots (12 onces)	12 pots --XXXXXX--
18 cartons (500 g)	18 cartons --XXXXXX--
12 pots (200 g)	12 pots --XXXXXX--
1 carton (1 kg)	1 carton --XXXXXX--
10 cartons (500 g)	10 cartons --XXXXXX--

(10 lignes)

```
cours=> SELECT REGEXP_REPLACE(QUANTITE,'(\()(.*)(\))','-\1-\2-\3-')
cours->      A1,REGEXP_REPLACE(QUANTITE,'(\()(.*)(\))','[\2]') A2,
cours->      REGEXP_REPLACE(QUANTITE,'(\()(.*)(\))','\2') A3
cours-> FROM PRODUITS
cours-> WHERE QUANTITE ~ '(cartons|pots).*(.*)'
cours-> LIMIT 10;
```

a1	a2	a3
48 pots -(-6 onces)-	48 pots [6 onces]	48 pots : 6 onces
12 pots -(-8 onces)-	12 pots [8 onces]	12 pots : 8 onces
12 cartons -(-1 kg)-	12 cartons [1 kg]	12 cartons : 1 kg
12 pots -(-12 onces)-	12 pots [12 onces]	12 pots : 12 onces
18 cartons -(-500 g)-	18 cartons [500 g]	18 cartons : 500 g
12 pots -(-200 g)-	12 pots [200 g]	12 pots : 200 g
10 cartons -(-500 g)-	10 cartons [500 g]	10 cartons : 500 g
24 cartons -(-500 g)-	24 cartons [500 g]	24 cartons : 500 g
12 cartons -(-250 g)-	12 cartons [250 g]	12 cartons : 250 g
24 cartons -(-200 g)-	24 cartons [200 g]	24 cartons : 200 g

(10 lignes)

Les expressions '\1', '\2' et '\3' sont les chaînes de caractères de l'expression rationnelle, qui doivent être remplacées. Dans notre exemple ce sont : '(\()','(.*)' et '(\))'. Ainsi chacune de ces chaînes de caractères peut être utilisée ou non dans la nouvelle définition et on peut rajouter d'autres caractères pour formater la chaîne avec une grande flexibilité.



```
cours=> SELECT TELEPHONE, REGEXP_REPLACE(LPAD(
cours->      REGEXP_REPLACE( TELEPHONE,
cours->      '([^\0-9]',' ','g'),11),
cours->      '([[:digit:]]|[:space:]]{3})([[:digit:]]{2})' ||
cours->      '([[:digit:]]{2})([[:digit:]]{2})',
cours->      '(\1) \2.\3.\4.') "Téléphone"
cours-> FROM CLIENTS
cours-> LIMIT 6;
```

telephone	Téléphone
(71) 555-2282	(7) 15.55.22.82
0241-039123	(02) 41.03.91.23
02.40.67.88.88	(02) 40.67.88.88
(71) 555-0297	(7) 15.55.02.97
7675-3425	() 76.75.34.25
(11) 555-9857	(1) 15.55.98.57
(6 lignes)	

LENGTH

La fonction « **LENGTH** » renvoie la longueur, en nombre des caractères, de la chaîne.

ASCII

La fonction « **ASCII** » retourne le code ASCII du caractère.

CHR, CHAR

La fonction « **CHR** » ou « **CHAR** » retourne le caractère de la valeur ASCII.



```
cours=# SELECT LENGTH('Chaine'),ASCII('A'),CHR(65);
length | ascii | chr
-----+-----+-----
      6 |    65 |  A
```

FORMAT

La fonction produit une sortie formatée suivant une chaîne de formatage, dans un style similaire à celui de la fonction C **sprintf**.

Le type de conversion de format à utiliser pour produire la sortie du spécificateur de format. Les types suivants sont supportés :

- s** formate la valeur de l'argument comme une simple chaîne. Une valeur NULL est traitée comme une chaîne vide.
- I** traite la valeur de l'argument comme un identifiant SQL, en utilisant les guillemets doubles si nécessaire. Une valeur NULL est une erreur.
- L** met entre guillemets simple la valeur en argument pour un littéral SQL. Une valeur NULL est affichée sous la forme d'une chaîne NULL, sans guillemets.



```
cours=> select format('Employé : %s,%s',nom,prenom)
cours-> from employes limit 3;
      format
-----
Employé : Giroux,Jean-Claude
Employé : Fuller,Andrew
Employé : Brasseur,Hervé
(3 lignes)
```

Atelier 5



Atelier 5

- Le formatage des chaînes
- La manipulation des chaînes



Durée : 15 minutes

SQL pour MySQL et PostgreSQL

Module 1 : Présentation de l'environnement - Diapo 1.76

Exercice n° 1 Le formatage des chaînes

Écrivez les requêtes permettant d'afficher :

- Le nom et le prénom en majuscule concaténées avec un espace au milieu. Il faut prendre soin de ne pas dépasser une longueur maximum de 14 caractères.

Exercice n° 2 La manipulation des chaînes

Écrivez les requêtes permettant d'afficher :

- La liste des produits, type d'emballage ('boîte', 'boîtes', 'pots', 'cartons', ...) et quantité du type d'emballage ('36 boîtes', '12 pots (12 onces)', ...) triés par ordre alphabétique du type d'emballage. Le résultat de la requête doit être comme dans l'exemple suivant :

NOM_PRODUIT	Emballage	Quantité
Konbu	boîtes	1
Chai	boîtes	10
Zaanse koeken	boîtes	10
Teatime Chocolate Biscuits	boîtes	10
Ipoh Coffee	boîtes	16
Filo Mix	boîtes	16
Alice Mutton	boîtes	20
Boston Crab Meat	boîtes	24
Pâté chinois	boîtes	24
Pavlova	boîtes	32
...		

- Les employés et leur âge comme dans l'exemple suivant :

Employé	Âge
-----	---
FULLER Andrew	50
BUCHANAN Steven	47
CALLAHAN Laura	44
PEACOCK Margaret	43
KING Robert	41
LEVERLING Janet	38
SUYAMA Michael	38
DAVOLIO Nancy	33
DODSWORTH Anne	32

- La société et le numéro de téléphone des fournisseurs comme une liste des valeurs numériques.

Fonctions de calcul arithmétique

Une expression arithmétique est une combinaison de noms de colonnes, de constantes et de fonctions arithmétiques combinées au moyen des **opérateurs** arithmétiques addition « + », soustraction « - », multiplication « * » ou division « / ».

Les constantes et opérateurs arithmétiques ont été présentés précédemment ; les principales fonctions arithmétiques sont exposées ci-après.

MOD

La fonction « **MOD** » permet de calculer le reste de la division du premier argument par le deuxième.

POWER

La fonction « **POWER** » permet d'élever un nombre à une puissance.



```
cours=# SELECT MOD(7,2) A, 7%2 A, POWER(3,2) B, 3^2 B;
```

a	a	b	b
1	1	9	9

ABS

La fonction « **ABS** » permet de calculer la valeur absolue de l'argument.

CEIL

La fonction « **CEIL** » permet de calculer le plus petit entier supérieur ou égal à l'argument.

FLOOR

La fonction « **FLOOR** » permet de calculer le plus grand entier inférieur à l'argument.

ROUND

La fonction « **ROUND** » permet de calculer une valeur arrondie avec une précision donnée.

TRUNC

La fonction « **TRUNC** » permet de calculer une valeur tronquée à la précision indiquée.

TRUNC(ARGUMENT,PRECISION)



```
cours=# SELECT CEIL(-10.23) A,CEIL(0) B,CEIL(20.23) C,
cours=# FLOOR(-10.23) D,FLOOR(0) E,FLOOR(20.23) F;
```

a	b	c	d	e	f
-10	0	21	-11	0	20

```
cours=# SELECT ROUND(-10.2326,2) A,ROUND(10.2366,3) B,
cours=# ROUND(-10.2326) C, ROUND(102356,-2) D,
```

```
cours-# ROUND(102326,-3) E,ROUND(102326) F;
```

a	b	c	d	e	f
-10.23	10.237	-10	102400	102000	102326



Note

Les fonctions d'arrondis acceptent comme arguments des valeurs numériques mais également des nombres réels à virgule flottante.

Ils effectuent l'arrondi en respectant les mêmes règles que pour les types numériques classiques.

Atelier 6



Atelier 6

- Les fonctions d'arrondis



Durée : 10 minutes

SQL pour MySQL et PostgreSQL

Module 1 : Présentation de l'environnement - Diapo 1.85

Exercice n° 1 Les fonctions d'arrondis

Écrivez les requêtes permettant d'afficher :

- Les employés et leur salaire journalier (salaire / 20) arrondi à l'entier inférieur.
- Les employés et leur salaire journalier (salaire / 20) arrondi à l'entier supérieur.
- Les produits commercialisés, la valeur du stock, les unités en stock fois le prix unitaire, arrondie à la centaine près.
- Les produits commercialisés, la valeur du stock, les unités en stock fois le prix unitaire, arrondie à la dizaine inférieure.
- Les employés et leur revenu annuel (salaire*12 + commission) arrondi à la centaine près.

Les fonctions de dates

CURRENT_DATE, NOW(), STATEMENT_TIMESTAMP()

La fonction « **CURRENT_DATE** » permet de connaître la date et l'heure actuelle.



```
cours=# SELECT CURRENT_DATE, NOW(), STATEMENT_TIMESTAMP();
      date      |              now              | statement_timestamp
-----+-----+-----
 2016-03-06 | 2016-03-06 09:40:56.08735+01 | 2016-03-06 09:40:56.08735+01
(1 ligne)
```

CURRENT_TIMESTAMP

Les fonctions : « **CURTIME** », « **CURRENT_TIME** », « **CURRENT_TIME** », « **CURRENT_TIMESTAMP** », « **SYSDATE** », « **NOW** » permettent de connaître la date et l'heure relative à la plage horaire de la session.



```
cours=# SELECT CURRENT_TIME A, CURRENT_TIMESTAMP B, NOW() C;
      a      |              b              |              c
-----+-----+-----
 09:16:06.086+02 | 2014-09-19 09:16:06.086+02 | 2014-09-19 09:16:06.086+02
```

EXTRACT

La fonction « **EXTRACT** » permet d'extraire un élément depuis un élément de type date ou bien un intervalle de temps.

EXTRACT (FORMAT FROM EXPRESSION)

FORMAT

Le format peut être une des valeurs suivantes :

« **CENTURY** », « **DAY** », « **DECADE** », « **DOW** », « **DOY** », « **EPOCH** », « **HOURL** », « **ISODOW** », « **ISOYEAR** », « **MICROSECONDS** », « **MILLENNIUM** », « **MILLISECONDS** », « **MINUTE** », « **MONTH** », « **QUARTER** », « **SECOND** », « **TIMEZONE** », « **TIMEZONE_HOUR** », « **TIMEZONE_MINUTE** », « **WEEK** », « **YEAR** ».



```
cours=# SELECT EXTRACT (DAY FROM DATE_Eмбаuche) "Jour",
cours=#         EXTRACT (DOW FROM DATE_Eмбаuche) "de la semaine",
cours=#         EXTRACT (ISODOW FROM DATE_Eмбаuche) "de la semaine ISO",
cours=#         EXTRACT (DOY FROM DATE_Eмбаuche) "de l'année"
cours=# FROM EMPLOYES LIMIT 5;
 Jour | de la semaine | de la semaine ISO | de l'année
-----+-----+-----+-----
    8 |              5 |              5    |        67
    9 |              6 |              6    |       221
   26 |              5 |              5    |       299
    9 |              4 |              4    |        40
    7 |              6 |              6    |       250

cours=# SELECT EXTRACT (ISOYEAR FROM DATE_Eмбаuche) "Année",
cours=#         EXTRACT (QUARTER FROM DATE_Eмбаuche) "Trimestre",
cours=#         EXTRACT (MONTH FROM DATE_Eмбаuche) "Mois",
```

```
cours-#      EXTRACT (WEEK FROM DATE_EMBAUCHE) "Semaine"
cours-# FROM EMPLOYES LIMIT 5;
```

Année	Trimestre	Mois	Semaine
2002	1	3	10
2003	3	8	32
2001	4	10	43
1995	1	2	6
2002	3	9	36

Opérateurs

Opérateurs

Opérande / Opérande		DATE	TIMESTAMP	INTERVAL	Number
	Opérateur				
DATE	+	—	—	DATE	DATE
	-	DATE	DATE	DATE	DATE
TIMESTAMP	+	—	—	TIMESTAMP	—
	-	INTERVAL	INTERVAL	TIMESTAMP	TIMESTAMP
INTERVAL	+	DATE	TIMESTAMP	INTERVAL	—
	-	—	—	INTERVAL	—
	*	—	—	—	INTERVAL
	/	—	—	—	INTERVAL
Number	+	DATE	DATE	—	NA
	-	—	—	—	NA
	*	—	—	INTERVAL	NA
	/	—	—	—	NA

TSOFT - ORACLE 10g SQL et PL*SQL

Module 7 : Le traitement des dates - Diapo 7.5



Une expression de type date est une combinaison de noms de colonnes, de constantes et de fonctions de manipulation de date combinés au moyen des **opérateurs** addition « + », soustraction « - », multiplication « * » ou division « / ».

L'opération doit être lue de la manière suivante :

« OPERANDE » « OPERATEUR » « OPERANDE » = « RESULTAT »

Dans l'exemple ci-après, vous pouvez remarquer l'utilisation de l'opérateur de multiplication avec un type de donnée « **INTERVAL DAY TO SECOND** ».



```
cours=# SELECT DATE_EмбаUCHE + 1 A, DATE_EмбаUCHE - 1 B
cours=# FROM EMPLOYES LIMIT 3;
a      | b
-----+-----
2002-03-09 | 2002-03-07
2003-08-10 | 2003-08-08
2001-10-27 | 2001-10-25
```

Types intervalle

INTERVAL YEAR [(P)] TO MONTH

Il représente un intervalle de temps exprimé en années et en mois. C'est une valeur relative qui peut être utilisée pour incrémenter ou décrémenter une valeur absolue d'un type date.

« P » est un littéral entier entre 0 et 9 devant être utilisé pour spécifier le nombre de chiffres acceptés pour représenter les années (2 étant la valeur par défaut).

INTERVAL DAY [(P)] TO SECOND [(P)]

Il représente un intervalle de temps exprimé en jours, heures, minutes et secondes. C'est une valeur relative qui peut être utilisée pour incrémenter ou décrémenter une valeur absolue d'un type date.

« P » est un littéral entier entre 0 et 9 doit être utilisé pour spécifier le nombre de chiffres acceptés pour représenter les jours et les fractions de secondes (2 et 6 étant respectivement les valeurs par défaut).

PostgreSQL



```
cours=# SELECT DATE_EMBAUCHE A,
cours-#         DATE_EMBAUCHE + INTERVAL '8' MONTH B,
cours-#         DATE_EMBAUCHE - INTERVAL '2' YEAR C,
cours-#         DATE_EMBAUCHE + INTERVAL '10-1' D
cours-# FROM EMPLOYES LIMIT 5;
```

a	b	c	d
2002-03-08	2002-11-08 00:00:00	2000-03-08 00:00:00	2012-04-08 00:00:00
2003-08-09	2004-04-09 00:00:00	2001-08-09 00:00:00	2013-09-09 00:00:00
2001-10-26	2002-06-26 00:00:00	1999-10-26 00:00:00	2011-11-26 00:00:00
1995-02-09	1995-10-09 00:00:00	1993-02-09 00:00:00	2005-03-09 00:00:00
2002-09-07	2003-05-07 00:00:00	2000-09-07 00:00:00	2012-10-07 00:00:00

Atelier 7



Atelier 7

- Les zones horaires
- La manipulation des dates



Durée : 10 minutes

TSOFT - ORACLE 10g SQL et PL*SQL

Module 7 : Le traitement des dates - Diapo 7.11

Exercice n° 1 La manipulation des dates

Écrivez les requêtes permettant d'afficher :

- La date du prochain dimanche (à ce jour).
- Les dates du premier et du dernier jour du mois en cours.
- La date du premier jour du trimestre (format 'Q').
- Le nom, la date de fin de période d'essai (3 mois) et leur ancienneté à ce jour exprimé en mois pour tous les employés.
- Le nom et le jour de leur première paie (dernier jour du mois de leur embauche).

Les fonctions de conversion

Le langage SQL propose de nombreuses fonctions de conversion automatique entre les types de données. Bien que le moteur du SGBDR qui exécute chaque ordre SQL sache prendre en compte l'évaluation de certaines expressions qui utilisent des données de types différents, il est toujours préférable de programmer des expressions homogènes, dans lesquelles les conversions de types sont clairement indiquées par utilisation de fonctions de conversion.

Le tableau de l'image présente les fonctions de conversion entre les différents types de données que l'on va détailler.

CAST

La fonction « **CAST** », est un mécanisme de conversion d'un type de donnée en un autre type de donnée extrêmement souple et pratique. Cette fonction est connue des développeurs utilisant les langages orientés objet dans lesquels il est souvent nécessaire de transtyper un objet d'une classe en un objet d'une autre classe.

CAST(EXPRESSION AS NOM_TYPE)

EXPRESSION Une expression qui doit être convertie.

NOM_TYPE Le type de donnée cible.

```
SQL> SELECT CAST(DATE_NAISSANCE AS TIMESTAMP WITH TIME ZONE)
2 FROM EMPLOYES;
```

```
CAST(DATE_NAISSANCE AS TIMESTAMP WITH TIME ZONE)
```

```
-----
09/01/58 00:00:00,000000 +02:00
04/03/55 00:00:00,000000 +02:00
19/09/58 00:00:00,000000 +02:00
...
```



TO_CHAR / TO_DATE

La fonction « **TO_CHAR** » permet également de convertir une date, avec un certain format, en chaîne de caractères.

<i>Format</i>	<i>Description</i>
MM	Numéro du mois dans l'année
RM	Numéro du mois dans l'année en chiffres romains
MON	Le nom du mois abrégé sur trois lettres
MONTH	Le nom du mois écrit en entier
DDD	Numéro du jour dans l'année, de 1 à 366
DD	Numéro du jour dans le mois, de 1 à 31
D	Numéro du jour dans la semaine, de 1 à 7
DY	Le nom de la journée abrégé sur trois lettres
DAY	Le nom de la journée écrit en entier
YYYY	Année complète sur quatre chiffres
RR	Deux derniers chiffres de l'année de la date courante
Q	Le numéro du trimestre
WW	Numéro de la semaine dans l'année

IW	Semaine de l'année selon le standard ISO
W	Numéro de la semaine dans le mois
J	Calendrier Julien -jours écoulés depuis le 31 décembre 4713 av. J.-C
HH	Heure du jour, toujours de format 1-12
HH24	Heure du jour, sur 24 heures
MI	Minutes écoulées dans l'heure
SS	Secondes écoulées dans une minute
SSSS	Secondes écoulées depuis minuit, toujours 0-86399
AM, PM	Affiche AM ou PM selon qu'il s'agit du matin ou de l'après-midi
FM	Les valeurs sont renvoyées sans les caractères blanc avant ou après



```

cours=# SELECT TO_CHAR( NOW(), 'D DD DDD FMDAY FMDay' );
           to_char
-----
 6 19 262 FRIDAY Friday

cours=# SELECT TO_CHAR( NOW() - INTERVAL '1'
cours( #      MONTH, 'MM MON Mon MONTH Month' );
           to_char
-----
 08 AUG Aug AUGUST      August

cours=# SELECT TO_CHAR( NOW() - INTERVAL '1'
cours( #      MONTH, 'MM MON Mon FMMONTH FMMonth' );
           to_char
-----
 08 AUG Aug AUGUST August

cours=# SELECT TO_CHAR( NOW(), 'DD/MM/YYYY Q WW iW W HH:MM:SS SSSS' );
           to_char
-----
19/09/2014 3 38 43 3 12:09:37 44137

cours=# SELECT TO_CHAR( NOW(), 'DD/MM/YYYY HH24:MM:SS SSSS' );
           to_char
-----
19/09/2014 12:09:38 44138

cours=# SELECT TO_DATE( '19/09/2014 12:09', 'DD/MM/YYYY HH24:MM' );
           to_char
-----
19/09/2014 12:09

```

MAKE_DATE et MAKE_TIME

La fonction construit une date à partir d'une année un mois et un nombre de jours. La fonction « **MAKE_TIME** » permet de construire une heure.



```

cours=> select make_date(2015,6,10), make_time(12,20,00);
 make_date | make_time
-----+-----
 2015-06-10 | 12:20:00
(1 ligne)

```

DATE_TRUNC

La fonction est conceptuellement similaire à la fonction trunc pour les nombres. Les valeurs valides pour l'arrondi : microseconds, milliseconds, second, minute, hour, day, week, month, quarter, year, decade, century, millennium.



```
cours=> select date_trunc('day',now()), date_trunc('month',now()),
cours->          date_trunc('year',now());
```

date_trunc	date_trunc	date_trunc
2016-03-06 00:00:00+01	2016-03-01 00:00:00+01	2016-01-01 00:00:00+01

(1 ligne)

Atelier 8.1



Atelier 8.1

■ Les conversions



Durée : 10 minutes

TSOFT - ORACLE 10g SQL et PL*SQL

Module 8 : Les conversions SQL - Diapo 8.11

Exercice n° 1 Les conversions

Écrivez les requêtes permettant d'afficher :

- La date du jour formatée de la sorte :

Nous sommes le :

Vendredi 14 Juillet 2006

- L'heure du jour formatée de la sorte :

Il est : 13 heures et 07 minutes

- La date du jour, l'heure du jour et les secondes écoulées depuis minuit.
- La date dans trois ans et dix mois.

Les fonctions générales

CASE

L'instruction « **CASE** » permet de mettre en place une condition d'instruction conditionnelle « **IF..THEN..ELSE** » directement dans une requête. Le fonctionnement est similaire à la fonction « **DECODE** » avec plus de flexibilité.

La première syntaxe de cette fonction est :



```
cours=> SELECT NOM, PRENOM, FONCTION,
cours->          CASE FONCTION
cours->          WHEN 'Vice-Président' THEN
cours->          SALAIRE*1.1
cours->          WHEN 'Chef des ventes' THEN
cours->          SALAIRE*1.2
cours->          WHEN 'Représentant(e)' THEN
cours->          SALAIRE*1.1 + COMMISSION
cours->          ELSE
cours->          SALAIRE*1.1
cours->          END "Salaire"
cours-> FROM EMPLOYES LIMIT 3;
```

nom	prenom	fonction	Salaire
Giroux	Jean-Claude	Président	165000.000
Fuller	Andrew	Vice-Président	105600.000
Brasseur	Hervé	Vice-Président	161700.000

(3 lignes)

La deuxième syntaxe de cette fonction est :



```
cours=# SELECT NOM, FONCTION, SALAIRE,
cours=#          CASE
cours=#          WHEN FONCTION = 'Assistante commerciale'
cours=#          THEN '10%'
cours=#          WHEN FONCTION = 'Représentant(e)' AND
cours=#          SALAIRE < 2600
cours=#          THEN '30%'
cours=#          WHEN FONCTION = 'Représentant(e)' AND
cours=#          SALAIRE < 3200
cours=#          THEN '20%'
cours=#          ELSE
cours=#          'Pas d''augmentation'
cours=#          END "Salaire"
cours=# FROM EMPLOYES LIMIT 3;
```

nom	fonction	salaire	Salaire
Giroux	Président	150000.00	Pas d'augmentation
Fuller	Vice-Président	96000.00	Pas d'augmentation
Brasseur	Vice-Président	147000.00	Pas d'augmentation

Atelier 8.2

Atelier 8.2

■ Les fonctions générales



Durée : 15 minutes

TSOFT - ORACLE 10g SQL et PL*SQL

Module 8 : Les conversions SQL - Diapo 8.16

Exercice n° 1 Les fonctions générales

Écrivez les requêtes permettant d'afficher :

- Le nom, le prénom, le salaire et la commission formatée de la sorte :

NOM	PRENOM	SALAIRE	Commission
-----	-----	-----	-----
Fuller	Andrew	10000	Pas de commission
Buchanan	Steven	8000	Pas de commission
Peacock	Margaret	2856	250
Leverling	Janet	3500	1000
Davolio	Nancy	3135	1500
Dodsworth	Anne	2180	0
King	Robert	2356	800
Suyama	Michael	2534	600
Callahan	Laura	2000	Pas de commission

- Le nom du produit, la plus grande valeur entre la valeur des produits en stock et la valeur des produits commandés pour tous les produits disponibles. La valeur du stock ou de la commande est calculée en multipliant la plus grande valeur du stock ou de la commande par le prix unitaire. Toutes les valeurs des produits commandés doivent être affichées avec une valeur négative.

NOM_PRODUIT	Valeur Stock
-----	-----
Raclette Courdavault	21.725,00€
Chai	3.510,00€
Chang	-3.800,00€
...	

- La société, l'adresse et le numéro de fax des fournisseurs. S'il n'y a pas de numéro de fax renseigné, affichez le numéro de téléphone.

Les fonctions d'agrégat

Les fonctions "verticales" ou les fonctions d'agrégat, sont utilisées pour le calcul cumulatif des valeurs par rapport à un regroupement ou pour l'ensemble des lignes de la requête.

SUM

La fonction « **SUM** » calcule la somme des expressions arguments pour l'ensemble des lignes correspondantes.



```
cours=# SELECT SUM(SALAIRE),SUM(COMMISSION) FROM EMPLOYES ;
```

sum	sum
1185440.00	157690.00

AVG

La fonction « **AVG** » calcule la moyenne des expressions arguments pour l'ensemble des lignes correspondantes.



```
cours=# SELECT AVG(COMMISSION),AVG(CASE WHEN COMMISSION IS NULL
cours( #      THEN 0 ELSE COMMISSION END) FROM EMPLOYES ;
```

avg	avg
1609.0816326530612245	1420.6306306306306

Attention



La fonction « **AVG** » est influencée par les valeurs « **NULL** », la somme est calculée pour l'ensemble des lignes mais le nombre des lignes pris en compte est seulement celui pour la quelle la valeur EXPRESSION est « **NOT NULL** ».

MIN

La fonction « **MIN** » calcule la plus petite des valeurs pour les expressions arguments pour l'ensemble des lignes correspondantes.

MAX

La fonction « **MAX** » calcule la plus grande des valeurs pour les expressions arguments pour l'ensemble des lignes correspondantes.



```
cours=# SELECT MAX(SALAIRE) MAX_SAL,
cours=#      MAX(COMMISSION) MAX_COM,
cours=#      MAX(AGE) MAX_DATE,
cours=#      MAX(NOM) MAX_NOM
cours=# FROM EMPLOYES ;
```

max_sal	max_com	max_date	max_nom
150000.00	16480.00	1991-07-31	Zonca

VARIANCE

La fonction « **VARIANCE** » calcule la variance de toutes les valeurs pour l'ensemble des lignes correspondantes.

STDDEV

La fonction « **STDDEV** » calcule l'écart type des valeurs pour l'ensemble des lignes correspondantes.

COUNT

La fonction « **COUNT** » calcule le nombre des valeurs non NULL des expressions arguments pour l'ensemble des lignes correspondantes.



```
cours=# SELECT COUNT(*),COUNT(FONCTION), COUNT(DISTINCT FONCTION),
cours=#          COUNT(COMMISSION) FROM EMPLOYES;
count | count | count | count
-----+-----+-----+-----
    111 |    111 |      5 |    98
```

Dans l'exemple, vous pouvez distinguer quatre utilisations de la fonction COUNT pour le calcul du nombre :

- des lignes distinctes de la table EMPLOYES,
- des valeurs non « **NULL** » de la colonne FONCTION, sans tenir compte des doublons,
- des valeurs non « **NULL** » et distinctes de la colonne FONCTION.
- des valeurs non « **NULL** » de la colonne COMMISSION.

FONCTION	SUM(SALAIRE)
Assistante commerciale	2000
Chef des ventes	8000
Représentant (e)	16561
Vice-Président	10000

Module 9 : Groupement des données - Diapo 9.6

```
SELECT [ALL | DISTINCT]{*,[EXPRESSION1 [AS] ALIAS1[,...]}
FROM NOM_TABLE
WHERE PREDICAT
GROUP BY [NOM_COLONNE1|EXPRESSION1],
         [NOM_COLONNE2|EXPRESSION2][,...]
ORDER BY [NOM_COLONNE1|POSITION1] [ASC|DESC][,...] ;
```

3-27

1	1	56	2
1	2	13	1
2	2	133	4
2	3	20	1
2	7	20	1
2	9	40	1
3	2	126	2
3	7	15	1
4	2	60	1
4	3	20	1
4	6		1
4	7	104	2
4	8	31	1
4	9	40	1
5	4	108	2
...			

Dans l'exemple précédent, vous pouvez voir que la requête ne retourne qu'une seule ligne qui rassemble l'ensemble des lignes de la table qui respecteront les conditions de la clause « **WHERE** ».



```
cours=# SELECT NOM,FONCTION,SUM(SALAIRE+COMMISSION)
cours=# FROM EMPLOYES;
ERREUR: la colonne « employees.nom » doit apparaître dans la clause
GROUP BY ou doit être utilisé dans une fonction d'agrégat
LIGNE 1 : SELECT NOM,FONCTION,SUM(SALAIRE+COMMISSION)
          ^
```



Attention

Toute requête qui utilise des fonctions "verticales" sur un groupe défini, doit afficher, dans les expressions qui ne sont pas des arguments des fonctions "verticales" seulement les colonnes contenues dans la clause « **GROUP BY** ». Les colonnes affichables, en dehors des fonctions "verticales", sont celles qui ont une valeur unique dans le groupe.

Une colonne composant d'une expression critère d'un groupe doit, pour pouvoir être utilisée dans les expressions destinées à l'affichage, être employée avec la même expression de la clause « **GROUP BY** ».

```
cours=# SELECT EXTRACT(YEAR FROM DATE_COMMANDE) "Année",
cours=#         EXTRACT(MONTH FROM DATE_COMMANDE) "Mois",
cours=#         SUM(PORT) "Port"
cours=# FROM COMMANDES
cours=# GROUP BY EXTRACT(YEAR FROM DATE_COMMANDE),
cours=#         EXTRACT(MONTH FROM DATE_COMMANDE) ;
 Année | Mois | Port
-----+-----+-----
 2011  |    6 | 69621.70
 2010  |    7 | 49321.80
 2010  |   10 | 55018.80
 2010  |    3 | 49768.40
 2011  |    2 | 66132.70
 2010  |    1 | 49332.80
 2010  |   12 | 47898.20
 2010  |    4 | 46904.00
```

2010		5		50921.50
2010		8		53680.40
2010		6		49397.90
2010		2		46707.10
2011		1		65722.30
2011		3		73703.40
2010		9		47753.60
2011		5		67417.30
2010		11		49608.20
2011		4		68543.40

La sélection de groupe

La sélection de groupe

```
SELECT FONCTION, SUM(SALAIRE)
FROM EMPLOYES
GROUP BY FONCTION
HAVING SUM(SALAIRE) >= 10000;
```



FONCTION	SALAIRE

FONCTION	SUM(SALAIRE)
Représentant (e)	16561
Vice-Président	10000

TSOFT - ORACLE 10g SQL et PL*SQL

Module 9 : Groupement des données - Diapo 9.7

Les sélections dans une requête sans groupe sont effectuées dans la clause « **WHERE** ». Dans cette clause le prédicat (l'ensemble des critères de sélection) est exécuté pour chaque enregistrement de la table, le niveau de détail, le résultat de la requête étant formé par les lignes qui vérifient le prédicat.

Les requêtes groupées peuvent être sélectionnées à l'aide de la clause « **HAVING** », pour spécifier le prédicat sur groupe.

La syntaxe de l'instruction « **SELECT** » :

```
SELECT [ALL | DISTINCT]{*,[EXPRESSION1 [AS] ALIAS1[,...]}
FROM NOM_TABLE
WHERE PREDICAT
GROUP BY [NOM_COLONNE1|EXPRESSION1][,...]
HAVING PREDICAT
ORDER BY [NOM_COLONNE1|EXPRESSION1] [ASC|DESC][,...] ;
```



```
cours=> SELECT NO_EMPLOYE,  
cours->      EXTRACT(YEAR FROM DATE_COMMANDE) "Année",  
cours->      SUM(PORT) "Port"  
cours-> FROM COMMANDES  
cours-> GROUP BY NO_EMPLOYE,  
cours->      EXTRACT(YEAR FROM DATE_COMMANDE)  
cours-> HAVING SUM(PORT) > 18000  
cours-> ORDER BY SUM(PORT) DESC;
```

no_employe	Année	Port
84	2010	29958.10
63	2010	29113.40
79	2010	27924.30
111	2010	19954.70
79	2011	19722.20
39	2010	18917.40

84	2011	18826.20
7	2010	18805.60
105	2010	18751.00
63	2011	18593.80
29	2010	18015.90

(11 lignes)

. Dans l'exemple précédent, vous pouvez remarquer que le groupe est formé par les deux critères précisés dans la clause « **GROUP BY** », le numéro d'employé (NO_EMPLOYE) et l'année de la commande. Oracle exécute les clauses dans un ordre bien défini :

1. Sélectionne les lignes conformément à la clause « **WHERE** ».
2. Groupe les lignes conformément à la clause « **GROUP BY** ».
3. Calcule les résultats des fonctions d'agrégat pour chaque groupe.
4. Élimine les groupes conformément à la clause « **HAVING** ».
5. Ordonne les groupes conformément à la clause « **ORDER BY** ».

L'ordre d'exécution est important, car il affecte directement les performances des requêtes. En général, plus le nombre d'enregistrements éliminés par une clause « **WHERE** » est grand, plus l'exécution de la requête est rapide. Ce gain en performances provient de la réduction du nombre de lignes devant être traitées durant l'opération « **GROUP BY** ».

Lorsqu'une requête inclut une clause « **HAVING** », il est préférable de la remplacer par une clause « **WHERE** ». Toutefois, cette substitution est généralement possible seulement lorsque la clause « **HAVING** » est utilisée pour éliminer des groupes basés sur la colonne de groupement. Prenez l'exemple précédent : NO_EMPLOYE peut être utilisé aussi bien dans la clause « **WHERE** » que dans la clause « **HAVING** » cependant la requête s'exécute plus vite s'il est utilisé dans la clause « **WHERE** » étant donné que le nombre des lignes à regrouper est moins important.

Attention



Les expressions utilisées dans la clause « **HAVING** » peuvent contenir seulement des colonnes et expressions contenues dans la clause « **GROUP BY** » ou des fonctions "verticales" qui respectent la même syntaxe que les expressions de l'affichage.

Une requête peut contenir à la fois une clause « **WHERE** » et une clause « **HAVING** ». Dans ce cas, la clause « **WHERE** » doit précéder la clause « **GROUP BY** » et la clause « **HAVING** » doit lui succéder.

Sachez que vous pouvez utiliser un alias de colonne dans une clause « **ORDER BY** », mais pas dans une autre clause « **WHERE** », « **GROUP BY** » ou « **HAVING** ».

Transformer les tables en XML

Les fonctions suivantes transforment le contenu de tables relationnelles en valeurs XML.

```
table_to_xml(tbl regclass, nulls boolean,
             tableforest boolean, targetns text)
query_to_xml(query text, nulls boolean,
             tableforest boolean, targetns text)
cursor_to_xml(cursor refcursor, count int,
             nulls boolean, tableforest boolean, targetns text)
```

```
cours=> select table_to_xml('employes', true, false, '');
               table_to_xml
```

```
-----
<employes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">+
+
<row>+
+
  <no_employe>37</no_employe>+
  <rend_compte xsi:nil="true"/>+
  <nom>Giroux</nom>+
  <prenom>Jean-Claude</prenom>+
  <fonction>Président</fonction>+
  <titre>M.</titre>+
  <date_naissance>1979-04-28</date_naissance>+
  <date_embauche>2002-03-08</date_embauche>+
  <salaire>150000.00</salaire>+
  <commission xsi:nil="true"/>+
  <pays xsi:nil="true"/>+
  <region xsi:nil="true"/>+
</row>+
+
<row>+
+
  <no_employe>14</no_employe>+
  <rend_compte>37</rend_compte>+
  <nom>Fuller</nom>+
  <prenom>Andrew</prenom>+
  <fonction>Vice-Président</fonction>+
  <titre>M.</titre>+
  <date_naissance>1984-06-27</date_naissance>+
  <date_embauche>2003-08-09</date_embauche>+
  <salaire>96000.00</salaire>+
  <commission xsi:nil="true"/>+
  <pays xsi:nil="true"/>+
  <region>Amériques</region>+
</row>+
+
...
```

```
cours=> create table prodXML as
cours-> select nom_produit, societe fournisseur,
cours->           nom_categorie categorie, prix_unitaire,
cours->           unites_stock stock, unites_commandees commandees
cours->           from produits join fournisseurs on
cours->           ( produits.no_fournisseur = fournisseurs.no_fournisseur)
```



```

cours->      join categories on
cours->      (produits.code_categorie =categories.code_categorie
cours(>      and produits.code_categorie in (1,3));
SELECT 33
cours=> select table_to_xml('prodXML', true, false, '');
               table_to_xml

-----
<prodxml xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">+
<row>                                                +
  <nom_produit>Tea</nom_produit>                    +
  <fournisseur>Formaggi Fortini s.r.l.</fournisseur> +
  <categorie>Boissons</categorie>                  +
  <prix_unitaire>2.00</prix_unitaire>               +
  <stock>50</stock>                                +
  <commandees>20</commandees>                      +
</row>                                              +
                                                    +
<row>                                                +
  <nom_produit>Chai</nom_produit>                   +
  <fournisseur>Exotic Liquids</fournisseur>         +
...

```

query_to_xml exécute la requête dont le texte est passé par le paramètre query et transforme le résultat

```

cours=> select query_to_xmlschema(
cours'>      'select nom_produit, societe fournisseur,
cours'>      nom_categorie categorie,prix_unitaire,
cours'>      unites_stock stock, unites_commandees commandees
cours'>      from produits join fournisseurs on
cours'>      ( produits.no_fournisseur =
cours'>      fournisseurs.no_fournisseur)
cours'>      join categories on
cours'>      (produits.code_categorie =categories.code_categorie
cours'>      and produits.code_categorie in (1,3))', true, false, '');
               query_to_xmlschema

-----
<xsd:schema                                          +
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">    +
                                                    +
  <xsd:simpleType name="VARCHAR">                  +
    <xsd:restriction base="xsd:string">            +
      </xsd:restriction>                          +
    </xsd:simpleType>                              +
                                                    +
  <xsd:simpleType name="NUMERIC">                   +
    </xsd:simpleType>                              +
                                                    +
  <xsd:complexType name="RowType">                  +
    <xsd:sequence>                                  +
      <xsd:element name="nom_produit" type="VARCHAR" nillable="true"></xsd:element> +
      <xsd:element name="fournisseur" type="VARCHAR" nillable="true"></xsd:element> +
      <xsd:element name="categorie" type="VARCHAR" nillable="true"></xsd:element> +
      <xsd:element name="prix_unitaire" type="NUMERIC" nillable="true"></xsd:element>+
      <xsd:element name="stock" type="NUMERIC" nillable="true"></xsd:element>    +
      <xsd:element name="commandees" type="NUMERIC" nillable="true"></xsd:element> +
    </xsd:sequence>                                +
  </xsd:complexType>                              +
                                                    +
  <xsd:complexType name="TableType">              +

```

```

    <xsd:sequence>
      <xsd:element name="row" type="RowType" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:element name="table" type="TableType"/>

</xsd:schema>
+
+
+
+
+
+
+

CREATE TABLE prixcarburants (id serial PRIMARY KEY, station xml);

INSERT INTO prixcarburants(station) VALUES
(' <pdv id="1000001" latitude="4620114" longitude="519791" cp="01000"
pop="R">
  <adresse>596 AVENUE DE TREVOUX</adresse>
  <ville>SAINT-DENIS-LÈS-BOURG</ville>
  <ouverture debut="01:00:00" fin="01:00:00" saufjour="" />
  <services>
    <service>Automate CB</service>
    <service>Vente de gaz domestique</service>
    <service>Station de gonflage</service>
  </services>
  <prix nom="Gazole" id="1" maj="2016-08-26T14:08:04"
valeur="1085"/>
  <prix nom="SP95" id="2" maj="2016-08-26T14:08:04"
valeur="1269"/>
  <prix nom="SP98" id="6" maj="2016-08-26T14:08:04"
valeur="1289"/>
  <rupture id="5" nom="E10" debut="2015-02-09T16:02:00" fin="2016-
09-01T00:09:29"/>
</pdv>');
INSERT INTO prixcarburants(station) VALUES
(' <pdv id="1000002" latitude="4621842" longitude="522767" cp="01000"
pop="R">
  <adresse>16 Avenue de Marboz</adresse>
  <ville>BOURG-EN-BRESSE</ville>
  <ouverture debut="01:00:00" fin="01:00:00" saufjour="" />
  <services>
    <service>Automate CB</service>
    <service>Vente de gaz domestique</service>
  </services>
  <prix nom="Gazole" id="1" maj="2016-08-24T09:08:47"
valeur="1099"/>
  <prix nom="SP95" id="2" maj="2016-08-24T09:08:47"
valeur="1269"/>
  <prix nom="SP98" id="6" maj="2016-08-24T09:08:48"
valeur="1299"/>
  <rupture id="3" nom="E85" debut="2009-11-03T12:11:00" fin="2016-
09-01T00:09:29"/>
</pdv>');
...

```


Traiter du XML

Les fonctions suivantes transforment le contenu de tables relationnelles en valeurs XML.

xpath(xpath, xml [, nsarray])

La fonction xpath évalue l'expression XPath avec un document XML bien formé. En particulier, il doit avoir un seul élément de nœud racine.



```
cours=> SELECT xpath('//adresse/text()',station) adresse,
cours->          xpath('/pdv/@cp',station) cp,
cours->          xpath('//ville/text()',station) ville
cours-> FROM prixcarburants;
```

adresse	cp	ville
{ "596 AVENUE DE TREVOUX" }	{ 01000 }	{ SAINT-DENIS-LÈS-BOURG }
{ "16 Avenue de Marboz" }	{ 01000 }	{ BOURG-EN-BRESSE }
{ "20 Avenue du Maréchal Juin" }	{ 01000 }	{ Bourg-en-Bresse }
{ "642 Avenue de Trévoux" }	{ 01000 }	{ SAINT-DENIS-LÈS-BOURG }
{ "1 Boulevard John Kennedy" }	{ 01000 }	{ BOURG-EN-BRESSE }
{ "Avenue Amédée Mercier" }	{ 01000 }	{ Bourg-en-Bresse }
{ "Bd Charles de Gaulle" }	{ 01000 }	{ BOURG-EN-BRESSE }
{ "56 Rue du Stand" }	{ 01000 }	{ Bourg-en-Bresse }
{ "Boulevard Charles de Gaulle" }	{ 01000 }	{ BOURG-EN-BRESSE }
{ "642, AVENUE DE TREVOUX" }	{ 01000 }	{ "ST DENIS LES BOURG" }
{ "BOULEVARD CHARLES DE GAULLE" }	{ 01000 }	{ "BOURG EN BRESSE" }
{ "LE GRAND RIVOLET" }	{ 01090 }	{ MONTCEAUX }
{ "ROUTE DE DORTAN" }	{ 01100 }	{ Arbent }
{ "Rue Brillat-Savarin" }	{ 01100 }	{ OYONNAX }
{ "174 Cours de Verdun" }	{ 01100 }	{ Oyonnax }
...		

```
cours=> SELECT xpath('/pdv/@id',station) ville,
cours->          xpath('/pdv/services/*',station) services
cours-> FROM prixcarburants;
```

ville	services
{ 1000001 }	{ "<service>Automate CB</service>","<service>Vente de gaz domestique</service>","<service>Station de gonflage</service>" }
{ 1000002 }	{ "<service>Automate CB</service>","<service>Vente de gaz domestique</service>" }
{ 1000004 }	{ "<service>Boutique alimentaire</service>","<service>Boutique non alimentaire</service>","<service>Vente de fioul domestique</service>","<service>GPL</service>","<service>Carburant qualité supérieure</service>","<service>Station de gonflage</service>","<service>Piste poids lourds</service>","<service>Toilettes publiques</service>","<service>Relais colis</service>","<service>Vente de gaz domestique</service>","<service>Location de véhicule</service>" }
{ 1000005 }	{ "<service>Vente de gaz domestique</service>","<service>Automate CB</service>","<service>Boutique alimentaire</service>","<service>Boutique non alimentaire</service>","<service>Carburant qualité supérieure</service>","<service>Lavage multi-programmes</service>","<service>Station de gonflage</service>","<service>Piste poids lourds</service>","<service>Station de lavage</service>" }
{ 1000006 }	{ "<service>Vente de gaz domestique</service>" }
{ 1000007 }	{ "<service>Vente de gaz domestique</service>","<service>Automate CB</service>","<service>Vente de fioul domestique</service>","<service>Station de lavage</service>","<service>Lavage multi-programmes</service>" }

4

Les requêtes multi-tables

L'opérateur CROSS JOIN

L'opérateur « **CROSS JOIN** » est un produit cartésien ; il donne le même résultat que celui d'une requête sans condition.

La syntaxe est la suivante :



```
cours=> SELECT COUNT(*) FROM PRODUITS;
count
-----
      120
(1 ligne)

cours=> SELECT COUNT(*) FROM CATEGORIES ;
count
-----
       10
(1 ligne)

cours=> SELECT COUNT(*)FROM PRODUITS CROSS JOIN CATEGORIES;
count
-----
     1200
(1 ligne)
```

L'opérateur « **NATURAL JOIN** » effectue la jointure entre deux tables en se servant des colonnes des deux tables qui portent le même nom.

```
cours=> SELECT COUNT(*) FROM CLIENTS NATURAL JOIN COMMANDES;
count
-----
    13462
(1 ligne)

cours=> SELECT COUNT(*) FROM DETAILS_COMMANDES NATURAL JOIN
PRODUITS;
ERREUR:  les JOIN/USING types numeric et character varying ne
peuvent pas correspondre
```

Dans l'exemple précédent, la requête joint les tables DETAILS_COMMANDE et PRODUITS à l'aide de l'opérateur « **NATURAL JOIN** ». Vous pouvez remarquer que la jointure donne un message d'erreur de nombre invalide. En effet la colonne QUANTITE est prise en compte pour la jointure ; son nom est identique dans les deux tables, mais le type de la colonne est différent.

L'opérateur JOIN USING

L'opérateur « **JOIN USING** » effectue la jointure entre deux tables en se servant des colonnes spécifiées respectant la syntaxe suivante :



```
cours=> SELECT CLIENTS.SOCIETE, FOURNISSEURS.SOCIETE
cours-> FROM CLIENTS JOIN FOURNISSEURS USING(VILLE);
```

societe	societe
Consolidated Holdings	Exotic Liquids
Eastern Connection	Exotic Liquids
Familia Arquibaldo	Refrescos Americanas LTDA
Lehmanns Marktstand	Plutzer Lebensmittelgroßmärkte AG
Mère Paillard	Ma Maison
North/South	Exotic Liquids
Paris spécialités	Aux joyeux ecclésiastiques
Queen Cozinha	Refrescos Americanas LTDA
Seven Seas Imports	Exotic Liquids
Spécialités du monde	Aux joyeux ecclésiastiques
Tradição Hipermercados	Refrescos Americanas LTDA
Alfreds Futterkiste	Heli Süßwaren GmbH Co. KG
Around the Horn	Exotic Liquids
B's Beverages	Exotic Liquids
Comércio Mineiro	Refrescos Americanas LTDA

(15 lignes)

La requête précédente affiche les clients qui sont localisés dans une ville d'un fournisseur ; la deuxième requête est la traduction dans l'ancienne syntaxe.



```
cours=#SELECT COUNT(*) FROM DETAILS_COMMANDES NATURAL JOIN PRODUITS;
ERREUR: les JOIN/USING types numeric et character varying ne
peuvent pas correspondre
cours=# SELECT NOM,
cours=# NOM_PRODUIT,
cours=# SUM(DETAILS_COMMANDES.PRIX_UNITAIRE*
cours=# DETAILS_COMMANDES.QUANTITE) CA
cours=# FROM EMPLOYES NATURAL JOIN
cours=# COMMANDES NATURAL JOIN
cours=# DETAILS_COMMANDES JOIN
cours=# PRODUITS USING( REF_PRODUIT)
cours=# WHERE EXTRACT ( YEAR FROM DATE_COMMANDE) = 2011
cours=# GROUP BY NOM, NOM_PRODUIT
cours=# ORDER BY NOM, NOM_PRODUIT;
```

nom	nom_produit	ca
Alvarez	Alice Mutton	85680.00
Alvarez	Amandes	25005.60
Alvarez	Aniseed Syrup	40452.12
Alvarez	Beer	59998.56
Alvarez	Boston Crab Meat	35214.48
Alvarez	Boysenberry Spread	169359.36
Alvarez	Brownie Mix	81699.84
Alvarez	Cajun Seasoning	93844.80

...

L'opérateur JOIN ON

L'opérateur « **JOIN ON** » effectue la jointure entre deux tables en se servant des conditions spécifiées respectant la syntaxe suivante :



```
cours=# SELECT A.NOM_PRODUIT, B.QUANTITE, B.PRIX_UNITAIRE
cours=# FROM PRODUITS A JOIN DETAILS_COMMANDES B
cours=# ON ( A.REF_PRODUIT = B.REF_PRODUIT) LIMIT 10;
```

nom_produit	quantite	prix_unitaire
Chai	57	72.60
Chai	145	72.60
Chai	55	72.60
Chai	147	72.60
Chai	188	72.60
Chai	84	72.60
Chai	159	72.60
Chai	76	72.60
Chai	30	72.60
Chai	150	72.60

Dans l'exemple précédent, la requête joint les tables DETAILS_COMMANDE et PRODUITS à l'aide de l'opérateur « **JOIN ON** ».

L'opérateur « **JOIN ON** » effectue la jointure entre deux tables en se servant des conditions spécifiées respectant la syntaxe suivante :



```
cours=# SELECT NOM || ' ' || PRENOM "Vendeur", SOCIETE "Client",
cours=# EXTRACT ( YEAR FROM DATE_COMMANDE) "Année", PORT "Port"
cours=# FROM CLIENTS A JOIN COMMANDES B
cours=# ON ( A.CODE_CLIENT = B.CODE_CLIENT )
cours=# JOIN EMPLOYES C
cours=# ON ( B.NO_EMPLOYE = C.NO_EMPLOYE )
cours=# AND DATE_COMMANDE > '25/06/2011'
cours=# AND PORT > 98;
```

Vendeur	Client	Année	Port
Griner Florence	Princesa Isabel Vinhos	2011	98.60
Gregoire Renée	Bon app'	2011	98.80
Coutou Myriam	Old World Delicatessen	2011	98.50
Piroddi Nathalie	Godos Cocina Típica	2011	99.90
Herve Didier	Laughing Bacchus Wine Cellars	2011	99.70
Silberreiss Albert	Split Rail Beer Ale	2011	99.00

Atelier 10.1



Atelier 10.1

■ Les équijointures



Durée : 15 minutes

PostgreSQL prise en main

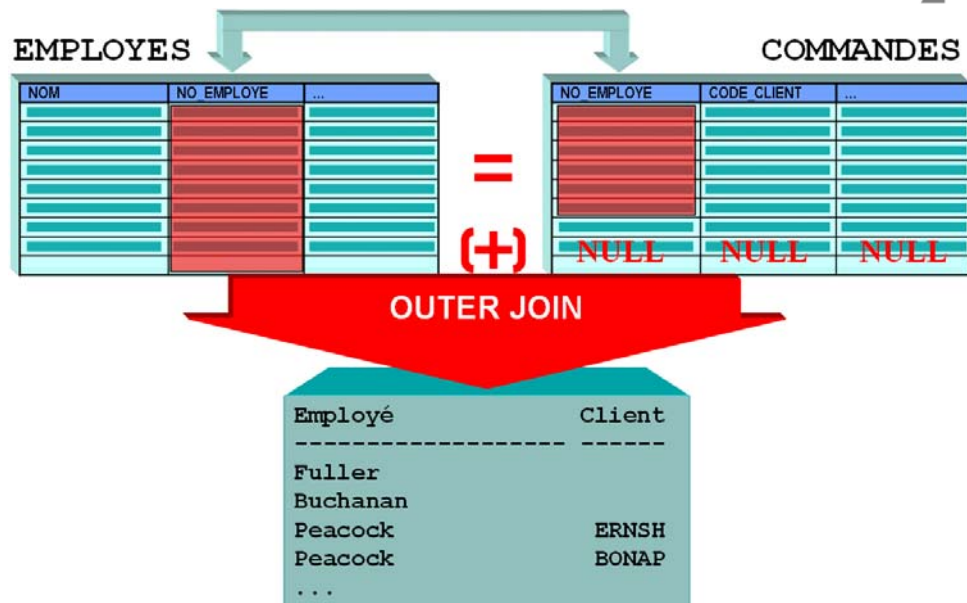
Module 1 : Présentation de l'environnement - Diapo 1.136

Exercice n° 1 Les équijointures

Écrivez les requêtes, compatible avec la norme ANSI/ISO SQL : 1999, permettant d'afficher :

- Le nom, le prénom et la société cliente, la date de la commande et les frais de port pour les employés qui ont effectué une vente pour les clients de Paris.
- La société cliente, le nombre des produits commandés, la ville et le pays qui ont commandé plus de vingt cinq produits.
- Le nom de la catégorie du produit, la société fournisseur et le nom du produit, uniquement pour les produits des catégories 1, 4 et 7.
- La société cliente, la société fournisseur et leur ville pour les clients qui sont localisés dans une ville d'un fournisseur (Il s'agit d'une jointure entre la table CLIENTS et FOURNISSEURS).
- Les sociétés clientes qui ont commandé le produit 'Chai'.

L'opérateur OUTER JOIN



PostgreSQL prise en main

Module 1 :Présentation de l'environnement - Diapo 1.137

L'opérateur « **OUTER JOIN ON** » effectue une jointure externe entre deux tables en se servant des conditions spécifiées respectant la syntaxe suivante :



```
cours=> select count(*)from clients;
count
-----
      91
(1 ligne)

cours=> select count(*)from fournisseurs;
count
-----
      29
(1 ligne)

cours=> select count(*)
cours-> from clients cl join fournisseurs fr
cours->      using(pays,ville);
count
-----
      15
(1 ligne)

cours=> select count(*)
cours-> from clients cl left join fournisseurs fr using(pays,ville);
count
-----
      91
(1 ligne)

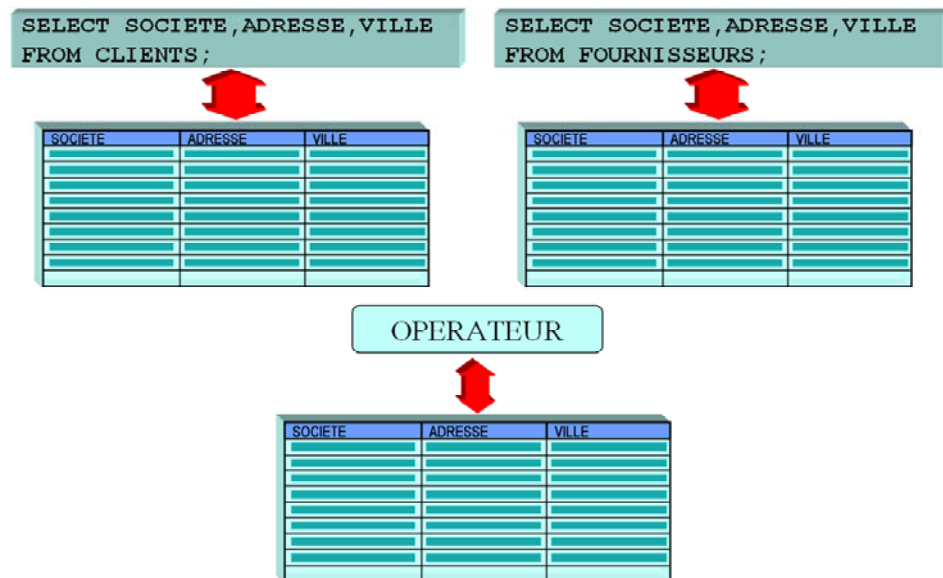
cours=> select count(*)
```



```
cours-> from clients cl right join fournisseurs fr
cours-> using(pays,ville);
count
-----
      38
(1 ligne)

cours=> select count(*)
cours-> from clients cl full join fournisseurs fr
cours->      using(pays,ville);
count
-----
     114
(1 ligne)
```

Les opérateurs ensemblistes



PostgreSQL prise en main

Module 1 : Présentation de l'environnement - Diapo 1.133

Il est parfois nécessaire de combiner des informations de même type à partir de plusieurs tables. Un exemple classique est la fusion de plusieurs listes de mailing en vue d'un envoi en masse de publicité. Les conditions d'envoi suivantes doivent généralement pouvoir être spécifiées :

- à toutes les personnes dans les deux listes (en évitant d'envoyer la lettre deux fois à une même personne) ;
- seulement aux personnes qui se trouvent dans les deux listes ;
- seulement aux personnes qui se trouvent dans une des deux listes.

Dans Oracle, ces trois conditions sont définies à l'aide des opérateurs :

- « **UNION** »
- « **INTERSECT** »
- « **MINUS** »

La syntaxe de l'instruction « **SELECT** » :

```
SELECT {*, [EXPRESSION1 [AS] ALIAS1[, ...]]} FROM NOM_TABLE
WHERE PREDICAT
GROUP BY [NOM_COLONNE1 | EXPRESSION1] [, ...]
HAVING PREDICAT
```

OPERATEUR [ALL | DISTINCT]

```
SELECT {*, [EXPRESSION1 [AS] ALIAS1[, ...]]} FROM NOM_TABLE
WHERE PREDICAT
GROUP BY [NOM_COLONNE1 | EXPRESSION1] [, ...]
HAVING PREDICAT
```

```
ORDER BY [POSITION1] [ASC | DESC] [, ...] ;
```

Dans une requête utilisant des opérateurs ensemblistes :

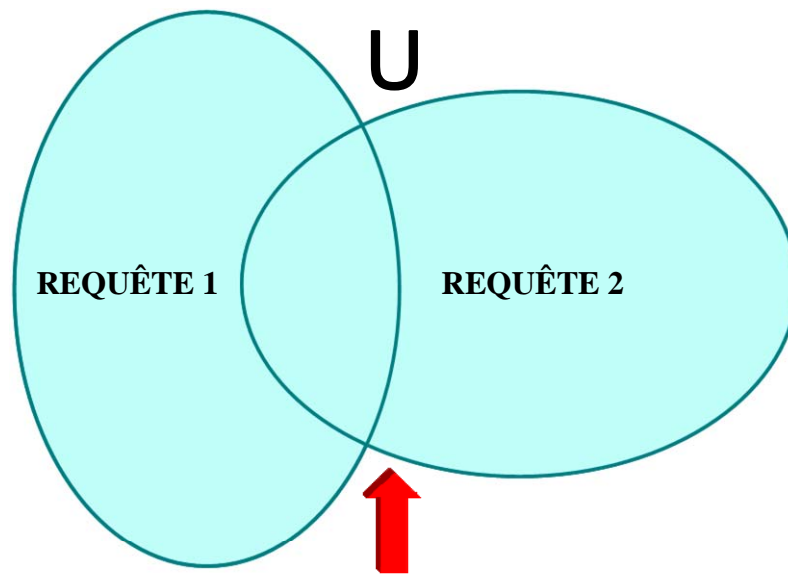
- Tous les ordres « **SELECT** » doivent avoir le même nombre de colonnes sélectionnées, et leurs types doivent être compatibles. Les conversions éventuelles doivent être faites à l'intérieur de l'ordre « **SELECT** » à l'aide des fonctions de conversion « **TO_CHAR** », « **TO_DATE** », etc.
- Aucun attribut ne peut être de type « **LONG** », « **BLOB** », « **CLOB** », « **BFILE** ».
- Les doublons sont éliminés, « **DISTINCT** » est implicite.
- Les noms des colonnes où alias sont ceux du premier ordre « **SELECT** ».
- La largeur de chaque colonne est donnée par la plus grande de tous ordres « **SELECT** » confondus.
- Si une clause « **ORDER BY** » est utilisée, elle doit faire référence au numéro de la colonne et non à son nom, car le nom peut être différent dans chacun des ordres « **SELECT** ».

Combinaison de plusieurs opérateurs ensemblistes

On peut utiliser, dans une même requête, plusieurs opérateurs « **UNION** », « **INTERSECT** » ou « **MINUS** », combinés avec des opérations de projection, de sélection ou de jointure. Dans ce cas, la requête est évaluée en combinant les deux premiers ordres « **SELECT** » à partir de la gauche avec le premier opérateur ensembliste, puis en combinant le résultat avec le troisième ordre « **SELECT** », etc.

Comme dans une expression arithmétique, il est possible de modifier l'ordre d'évaluation en utilisant des parenthèses.

L'opérateur UNION



Résultat = REQUÊTE_1 U REQUÊTE_2

PostgreSQL prise en main

Module 1 : Présentation de l'environnement - Diapo 1.134

L'opérateur d'union « **UNION** » entre deux requêtes permet de retrouver l'ensemble des lignes des deux requêtes de départ. Les attributs de même rang des requêtes de départ doivent être compatibles, c'est-à-dire définis de même type.



```
cours=> SELECT SOCIETE,VILLE,'Client' FROM CLIENTS
cours-> UNION
cours-> SELECT SOCIETE,VILLE,'Fournisseur' FROM FOURNISSEURS;
```

societe	ville	?column?
Que Delícia	Rio de Janeiro	Client
Hanari Carnes	Rio de Janeiro	Client
HILARIÓN-Abastos	San Cristóbal	Client
Alfreds Futterkiste	Berlin	Client
Chop-suey Chinese	Bern	Client
Folies gourmandes	Lille	Client
Tradição Hipermercados	São Paulo	Client
Princesa Isabel Vinhos	Lisboa	Client
Split Rail Beer Ale	Lander	Client
North/South	London	Client
Seven Seas Imports	London	Client
Karkki Oy	Lappeenranta	Fournisseur
New England Seafood Cannery	Boston	Fournisseur
Mère Paillarde	Montréal	Client
Aux joyeux ecclésiastiques	Paris	Fournisseur
Paris spécialités	Paris	Client
Wartian Herkku	Oulu	Client
...		

Dans l'exemple précédent, la requête affiche l'ensemble des tiers de l'entreprise, aussi bien des clients que des fournisseurs.

Note

Les noms des colonnes sont ceux de la première requête ainsi que les alias qui sont utilisées dans les ordres de tri.

Il faut se rappeler que la clause « **ORDER BY** » ne peut figurer qu'une fois en fin du bloque SQL, car elle opère sur le résultat concaténé des différents « **SELECT** ».



```
cours=> SELECT SOCIETE,VILLE,'Client' "Cli/Four" FROM CLIENTS
cours-> UNION
cours-> SELECT SOCIETE,VILLE,'Fournisseur' FROM FOURNISSEURS
cours-> ORDER BY VILLE, SOCIETE;
```

societe	ville	Cli/Four
Drachenblut Delikatessen	Aachen	Client
Rattlesnake Canyon Grocery	Albuquerque	Client
Old World Delicatessen	Anchorage	Client
Grandma Kelly's Homestead	Ann Arbor	Fournisseur
Gai pâturage	Annecy	Fournisseur
Vaffeljernet	Århus	Client
Galería del gastrónomo	Barcelona	Client
LILA-Supermercado	Barquisimeto	Client
Bigfoot Breweries	Bend	Fournisseur
Magazzini Alimentari Riuniti	Bergamo	Client
Alfreds Futterkiste	Berlin	Client
Heli Süßwaren GmbH Co. KG	Berlin	Fournisseur
Chop-suey Chinese	Bern	Client
Save-a-lot Markets	Boise	Client
New England Seafood Cannery	Boston	Fournisseur
Folk och få HB	Bräcke	Client
Königlich Essen	Brandenburg	Client
Maison Dewey	Bruxelles	Client
Cactus Comidas para llevar	Buenos Aires	Client
...		

Attention

L'opérateur d'union « **UNION** » entre deux requêtes permet de concaténer tous les types de données sans aucun contrôle de la pertinence de cet assemblage.

En d'autres termes on peut mélanger 'les choux' et 'les carottes' ; les informations sont affichées ensemble sans aucun contrôle.



```
cours=> SELECT SOCIETE,VILLE,'Client' "Cli/Four" FROM CLIENTS
cours-> UNION
cours-> SELECT NOM,FONCTION,'Employé' FROM EMPLOYES
cours-> UNION
cours-> SELECT NOM_PRODUIT,QUANTITE,'Produit' FROM PRODUITS;
```

societe	ville	Cli/Four
Les Comptoirs - Olive Oil	36 boîtes	Produit
HILARIÓN-Abastos	San Cristóbal	Client
Tradição Hipermercados	São Paulo	Client

Scones	24 paquets de 4 pièces	Produit
Tarte au sucre	48 tartes	Produit
Chai	10 boîtes x 20 sacs	Produit
Geitost	1 carton (500 g)	Produit
Bazart	Représentant(e)	Employé
Wimmers gute Semmelknödel	20 sacs x 4 pièces	Produit
Mère Paillarde	Montréal	Client
Paris spécialités	Paris	Client
Di Clemente	Représentant(e)	Employé
Maurousset	Représentant(e)	Employé
Marmalade	30 boîtes cadeau	Produit
Grandma's Boysenberry Spread	12 pots (8 onces)	Produit
Original Frankfurter grüne Soße	12 boîtes	Produit
Steeleye Stout	24 bouteilles (1 litre)	Produit
Spegesild	4 boîtes (250 g)	Produit
Gorgonzola Telino	12 cartons (100 g)	Produit
Weiss	Représentant(e)	Employé
Montesinos	Représentant(e)	Employé
...		



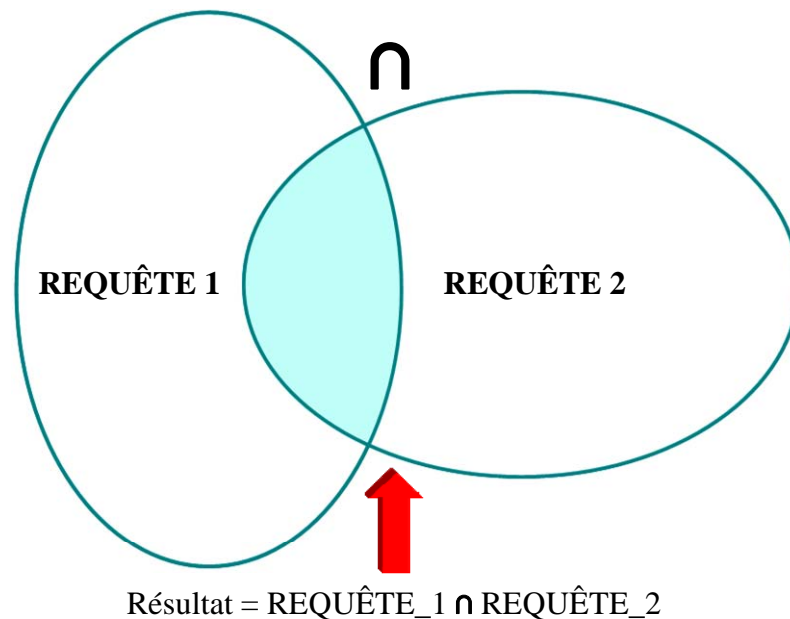
Attention

L'opérateur « **UNION** » comporte, comme l'ordre « **SELECT**, » la possibilité d'utiliser les options « **ALL** » ou « **DISTINCT** ».

Dans le cas de l'option « **DISTINCT** », l'option par défaut, les enregistrements en double sont éliminés ; c'est pour éliminer les doublons qu'Oracle effectue un tri des enregistrements.

Comme l'opérateur « **UNION** » c'est en effet « **UNION DISTINCT** » il est préférable d'utiliser « **UNION ALL** » chaque fois qu'il n'est pas nécessaire d'éliminer les doublons.

L'opérateur INTERSECT



PostgreSQL prise en main

Module 1 : Présentation de l'environnement - Diapo 1.135

L'opérateur d'intersection entre deux requêtes permet de retrouver le résultat composé des lignes qui appartiennent simultanément aux deux requêtes de départ.

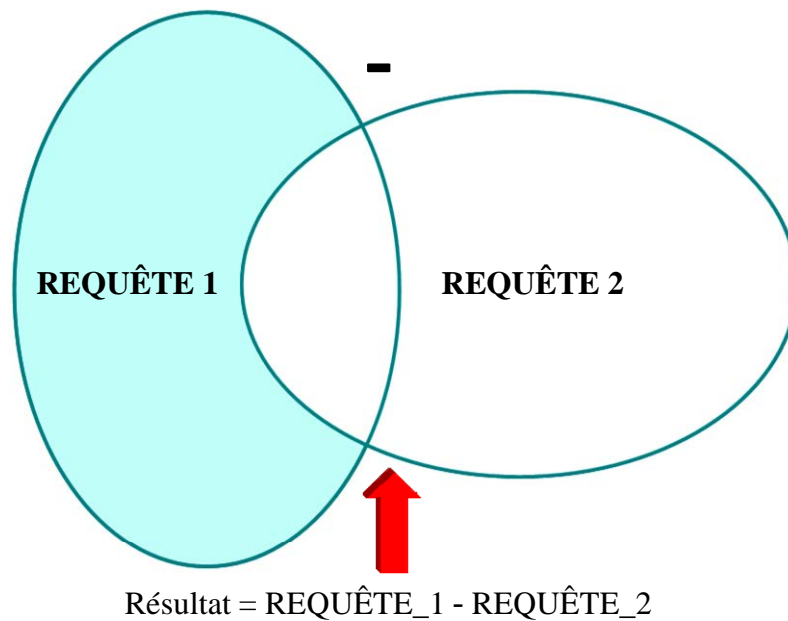


```
cours=> SELECT VILLE, NOM_PRODUIT
cours-> FROM CLIENTS NATURAL JOIN COMMANDES
cours->      JOIN DETAILS_COMMANDES USING(NO_COMMANDE)
cours->      JOIN PRODUITS USING(REF_PRODUIT)
cours-> INTERSECT
cours-> SELECT VILLE, NOM_PRODUIT
cours-> FROM PRODUITS NATURAL JOIN FOURNISSEURS;
```

ville	nom_produit
Frankfurt a.M.	Original Frankfurter grüne Soße
Montréal	Pears
Paris	Beer
Berlin	NuNuCa Nuß-Nougat-Creme
Berlin	Schoggi Schokolade
Frankfurt a.M.	Rössle Sauerkraut
Frankfurt a.M.	Rhönbräu Klosterbier
Paris	Côte de Blaye
São Paulo	Guaraná Fantástica
Montréal	Pâté chinois
London	Chang
Frankfurt a.M.	Thüringer Rostbratwurst
...	

La première requête retrouve la ville de résidence des clients et les noms des produits commandés. La deuxième requête retrouve la ville de résidence des fournisseurs et les noms des tous les produits commandés. L'intersection des deux requêtes affiche les villes des clients et le nom du produit pour les produits achetés par les clients qui habitent dans la même ville que le fournisseur.

L'opérateur DIFFERENCE



PostgreSQL prise en main

Module 1 : Présentation de l'environnement - Diapo 1.136

L'opérateur différence entre deux requêtes permet de retrouver le résultat composé des lignes qui appartiennent à la première requête et qui n'appartiennent pas à la deuxième requête. L'opérateur différence est le seul opérateur ensembliste non commutatif.



```
cours=> SELECT NO_EMPLOYE FROM EMPLOYES
cours-> EXCEPT
cours-> SELECT NO_EMPLOYE FROM COMMANDES;
no_employe
-----
21
24
104
86
75
14
89
18
37
57
44
95
30
64
27
11
33
23
109
...
```


Atelier 12.1



Atelier 12.1

- Les opérateurs ensemblistes



PostgreSQL prise en main

Durée : 15 minutes

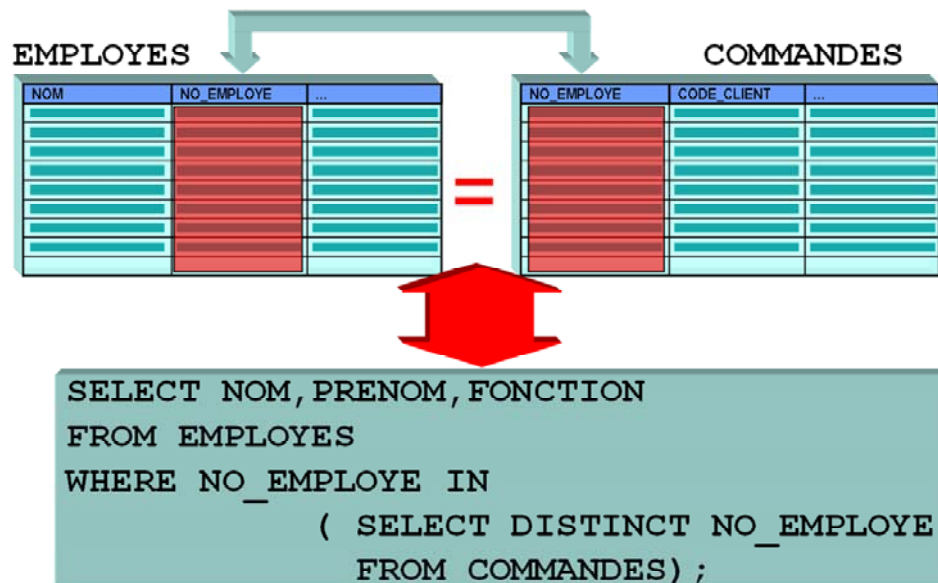
Module 1 : Présentation de l'environnement - Diapo 1.137

Exercice n° 1 Les opérateurs ensemblistes

Écrivez les requêtes permettant d'afficher :

- Pour un mailing, il faut trouver l'ensemble des tiers de l'entreprise (les sociétés clientes ou fournisseurs) ainsi que leur adresse et ville de résidence.
- Toutes les commandes qui comportent en même temps des produits de catégorie 1 du fournisseur 1 et produits de catégorie 2 du fournisseur 2.
- Les produits qu'on ne commande qu'à Paris.
- Les sociétés clientes qui ont commandé le produit 'Chai' mais également qui ont commandé plus de vingt cinq produits.

Les sous-requêtes



PostgreSQL prise en main

Module 1 : Présentation de l'environnement - Diapo 1.138

La jointure peut aussi être exprimée d'une manière plus procédurale avec des blocs imbriqués reliés par l'opérateur « **IN** ». On dit alors que la requête, dont le résultat sert de valeur de référence dans le prédicat, est une requête imbriquée ou une sous-requête.

Il est possible d'imbriquer plusieurs requêtes, le résultat de chaque requête imbriquée servant de valeur de référence dans la condition de sélection de la requête de niveau supérieur, appelée requête principale.

Il existe en fait plusieurs types de requêtes imbriquées, suivant les valeurs retournées, la dépendance ou non de la requête principale ou l'emplacement de la sous-requête.

Typologie des sous-requêtes :

- Sous-requête renvoyant une seule ligne
- Sous-requête renvoyant plusieurs lignes
- Sous-requête renvoyant plusieurs expressions
- Sous-requête synchronisée
- Sous interrogation dans la clause « **FROM** »

Une sous-requête peut être exécutée une seule fois pour toute ligne évaluée de la requête mère. Mais si la sous-requête est corrélée, elle s'exécute pour chaque ligne de la requête mère du fait que son contexte d'évaluation est susceptible de changer à chaque ligne.

Attention

Suivant le type de résultat qu'une sous-requête offre, on peut la placer dans les différentes clauses, à l'exception des clauses « **GROUP BY** » et « **ORDER BY** ».

Par définition, une sous-requête ne peut pas contenir de clause « **ORDER BY** », car elle ne produit pas un résultat destiné à l'affichage.



Sous-requête monolignes

```
SELECT NOM, PRENOM, FONCTION
FROM EMPLOYES
WHERE SALAIRE >
```

```
( SELECT AVG (SALAIRE)
  FROM EMPLOYES
    ) ;
```

Le salaire moyen des tous les employés.

Une sous-requête de ce type s'utilise lorsque la valeur de référence de la condition de sélection doit être unique.

La sous-requête est entièrement évaluée avant la requête principale. Le résultat est identique à celui obtenu en exécutant dans une première étape la sous-requête pour obtenir la valeur de référence et en utilisant cette valeur dans la seconde étape pour exécuter la requête principale.



```
cours=> SELECT NOM_PRODUIT FROM PRODUITS
cours-> WHERE UNITES_STOCK = (SELECT MAX(UNITES_STOCK)
cours(>                                FROM PRODUITS) ;
```

```
nom_produit
-----
Potato Chips
```

```
SQL> SELECT COUNT( REF_PRODUIT)
      2 FROM  COMMANDES NATURAL JOIN DETAILS_COMMANDES
      3 WHERE NO_COMMANDE = 224975;
```

```
COUNT(REF_PRODUIT)
-----
                    52
```

```
cours=> SELECT SOCIETE, NO_COMMANDE, COUNT( REF_PRODUIT)
cours-> FROM CLIENTS NATURAL JOIN COMMANDES JOIN DETAILS_COMMANDES
cours->      USING(NO_COMMANDE) GROUP BY SOCIETE, NO_COMMANDE
cours-> HAVING COUNT( REF_PRODUIT) >=( SELECT COUNT( REF_PRODUIT)
cours(>      FROM  COMMANDES NATURAL JOIN DETAILS_COMMANDES
cours(>      WHERE  NO_COMMANDE = 224975);
```

```
      societe                | no_commande | count
-----+-----+-----
Berglunds snabbköp         |      216798 |      52
```

Blondel père et fils	224968	52
Die Wandernde Kuh	216810	52
Du monde entier	216814	52
Furia Bacalhau e Frutos do Mar	225524	52
Hungry Owl All-Night Grocers	220267	52
Laughing Bacchus Wine Cellars	224974	52
LINO-Delicateses	224963	52
Rancho grande	215629	52
Rattlesnake Canyon Grocery	228419	52
The Big Cheese	224975	52
White Clover Markets	220279	52

Dans l'exemple précédent, la première requête affiche le nombre des produits pour la commande numéro 224975. La deuxième requête affiche tous les clients et les numéros des commandes qui ont un nombre égal ou supérieur de produits.

Richter Supermarkt	Genève	Suisse
Toms Spezialitäten	Münster	Allemagne
Die Wandernde Kuh	Stuttgart	Allemagne
Wolski Zajazd	Warszawa	Pologne
Alfreds Futterkiste	Berlin	Allemagne
Blauer See Delikatessen	Mannheim	Allemagne
Chop-suey Chinese	Bern	Suisse

Dans l'exemple précédent vous pouvez observer la liste des clients des employés qui n'ont pas de supérieur hiérarchique.

Attention



La négation de l'opérateur « **IN** », à savoir « **NOT IN** », doit être utilisée avec prudence car elle retourne « **FALSE** » si une des valeurs ramenées par la sous-interrogation est « **NULL** ».

Il est préférable de s'assurer qu'aucune des valeurs retournées par la sous-requête n'est « **NULL** ».



```
cours=> SELECT DISTINCT REND_COMPTE FROM EMPLOYES;
rend_compte
```

```
-----
14
23
11
24
18
86
37
95
33
```

```
cours=> SELECT NOM || ' ' || PRENOM EMPLOYE, NO_EMPLOYE, REND_COMPTE
cours-> FROM EMPLOYES WHERE NO_EMPLOYE NOT IN
cours-> ( SELECT REND_COMPTE FROM EMPLOYES );
employe | no_employe | rend_compte
-----+-----+-----
(0 ligne)
```

```
cours=> SELECT NOM || ' ' || PRENOM EMPLOYE, NO_EMPLOYE, REND_COMPTE
cours-> FROM EMPLOYES WHERE NO_EMPLOYE NOT IN
cours-> ( SELECT REND_COMPTE FROM EMPLOYES
cours(> WHERE REND_COMPTE IS NOT NULL );
employe | no_employe | rend_compte
-----+-----+-----
Besse José | 1 | 86
Destenay Agnès | 2 | 95
Letertre Sylvie | 3 | 33
Kremser Arnaud | 4 | 11
Lamarre Eric | 5 | 95
Cleret Doris | 6 | 23
Poidatz Benoît | 7 | 23
Messelier Philippe | 8 | 11
```

Gardeil Henri	9	24
Capharsie Gérard	10	23
Perny Sylvie	12	24
Courty Jean-Louis	13	86
Suyama Michael	15	33
Malejac Yannick	16	33
Blard Jean-Benoît	17	11
Pagani Hector	19	95
Gerard Sylvie	20	24
Poupard Claudette	21	
Piroddi Nathalie	22	33
Chaussende Maurice	25	23
Hanriot Catherine	26	23

```

cours=> SELECT SOCIETE, VILLE, PAYS FROM CLIENTS
cours-> WHERE CODE_CLIENT IN ( SELECT CODE_CLIENT
cours(>                                FROM EMPLOYES E,COMMANDES C
cours(>                                WHERE E.NO_EMPLOYE = C.NO_EMPLOYE
cours(>                                AND C.DATE_COMMANDE > '25/06/2011'
cours(>                                AND E.PAYS = 'France');

```

societe	ville	pays
-----+-----+-----		
France restauration	Nantes	France
Bon app'	Marseille	France
La maison d'Asie	Toulouse	France
Blondel père et fils	Strasbourg	France

Les opérateurs ANY et ALL



L'opérateur ANY

L'opérateur « **ANY** » compare une expression à chaque valeur de la liste des valeurs ramenée par la sous-requête, la condition sera vraie si elle est vraie pour au moins une des valeurs renvoyées par la sous-requête.

L'opérateur « **= ANY** » est équivalent à l'opérateur « **IN** ».



```

cours=> SELECT SOCIETE, VILLE, PAYS FROM CLIENTS
cours-> WHERE CODE_CLIENT IN ( SELECT CODE_CLIENT
cours(>                               FROM EMPLOYES E,COMMANDES C
cours(>                               WHERE E.NO_EMPLOYE = C.NO_EMPLOYE
cours(>                               AND C.DATE_COMMANDE > '25/06/2011'
cours(>                               AND E.PAYS = 'France');

```

societe	ville	pays
France restauration	Nantes	France
Bon app'	Marseille	France
La maison d'Asie	Toulouse	France
Blondel père et fils	Strasbourg	France

(4 lignes)

```

cours=> SELECT SOCIETE, VILLE, PAYS FROM CLIENTS
cours-> WHERE CODE_CLIENT=ANY( SELECT CODE_CLIENT
cours(>                               FROM EMPLOYES E,COMMANDES C
cours(>                               WHERE E.NO_EMPLOYE = C.NO_EMPLOYE
cours(>                               AND C.DATE_COMMANDE > '25/06/2011'
cours(>                               AND E.PAYS = 'France');

```

societe	ville	pays

France restauration	Nantes	France
Bon app'	Marseille	France
La maison d'Asie	Toulouse	France
Blondel père et fils	Strasbourg	France

(4 lignes)

L'opérateur « **< ANY** » signifie que l'expression est inférieure à au moins une des valeurs donc inférieure au maximum des valeurs de la liste.



```
cours=> SELECT NOM || ' ' || PRENOM EMPLOYE, SALAIRE FROM EMPLOYES
cours-> WHERE SALAIRE > ANY ( SELECT AVG(SALAIRE)
cours(> FROM EMPLOYES
cours(> WHERE REND_COMPTE IS NULL);
```

employe	salaire
Giroux Jean-Claude	150000.00
Fuller Andrew	96000.00
Brasseur Hervé	147000.00
Leger Pierre	19000.00
Splingart Lydia	16000.00

(5 lignes)

```
cours=> SELECT NOM_PRODUIT, UNITES_COMMANDEES FROM PRODUITS
cours-> WHERE UNITES_COMMANDEES >ANY
cours-> ( SELECT AVG(UNITES_COMMANDEES)*5
cours(> FROM PRODUITS GROUP BY CODE_CATEGORIE
cours(> HAVING AVG(UNITES_COMMANDEES) IS NOT NULL);
```

nom_produit	unites_commandees
Vegetable Soup	100
Chicken Soup	90
Aniseed Syrup	70
Maxilaku	60
Louisiana Hot Spiced Okra	100
Gorgonzola Telino	70
Røgede sild	70
Chocolade	70
Wimmers gute Semmelknödel	80
Green Tea	100

(10 lignes)

L'opérateur « **> ANY** » signifie que l'expression est supérieure à au moins une des valeurs donc supérieure au minimum.

L'opérateur ALL

L'opérateur « **ALL** » compare une expression à chaque valeur de la liste des valeurs ramenée par la sous-requête ; la condition sera vraie si elle est vraie pour chacune des valeurs renvoyées par la sous-requête.

L'opérateur « **< ALL** » signifie que l'expression est inférieure au minimum et « **> ALL** » signifie que l'expression est supérieure au maximum.



```
cours=> SELECT NOM_PRODUIT, UNITES_STOCK FROM PRODUITS
cours-> WHERE UNITES_STOCK > ALL ( SELECT UNITES_STOCK FROM PRODUITS
cours(> WHERE CODE_CATEGORIE = 2);
```

nom_produit	unites_stock
-------------	--------------

Boston Crab Meat	123
Rhönbräu Klosterbier	125
Green Tea	125
Potato Chips	200

Dans l'exemple précédent la requête affiche les produits pour lesquels la quantité du stock est supérieure a toutes les quantités des produits de la catégorie 2.



```
cours=> SELECT CODE_CLIENT,NO_EMPLOYE,
cours->      EXTRACT(YEAR FROM DATE_COMMANDE) ANNEE,
cours->      SUM(PORT)
cours-> FROM COMMANDES
cours-> WHERE CODE_CLIENT = 'HANAR'
cours->      AND NO_EMPLOYE != ALL (SELECT NO_EMPLOYE FROM EMPLOYES
cours(>      WHERE DATE_EMBAUCHE < '01/05/1992')
cours->      GROUP BY CODE_CLIENT,NO_EMPLOYE,
cours->      EXTRACT(YEAR FROM DATE_COMMANDE);
code_client | no_employe | annee | sum
-----+-----+-----+-----
HANAR      |          105 | 2010 | 2595.70
HANAR      |          105 | 2011 | 1375.70
HANAR      |           7 | 2010 | 1601.80
HANAR      |           7 | 2011 | 1021.90
(4 lignes)
```

Dans l'exemple précédent, la requête affiche les commandes pour le client 'HANAR' vendues par un employé embauché avant '01/05/1992'.

L'opérateur « **NOT IN** » est équivalent à l'opérateur « **!= ALL** ».

Sous-requête synchronisée

```
SELECT CODE_CATEGORIE,
       NOM_PRODUIT,
       UNITES_STOCK,
       PRIX_UNITAIRE
FROM PRODUITS A
WHERE UNITES_STOCK >
      ( SELECT AVG(UNITES_STOCK)
        FROM PRODUITS B
        WHERE B.CODE_CATEGORIE =
              A.CODE_CATEGORIE );
```

Corrélation, même colonne

PostgreSQL prise en main

Module 1 : Présentation de l'environnement - Diapo 1.143

Oracle autorise également le traitement d'une sous-requête faisant référence à une colonne de la table de l'interrogation principale. Le traitement est plus complexe dans ce cas, car il faut évaluer la sous-requête pour chaque ligne traitée par la requête principale. On dit alors que la sous-requête est synchronisée avec la requête principale. La sous-requête est évaluée pour **chaque ligne** de la requête principale.



```
cours=> SELECT CODE_CATEGORIE "Cat", NOM_PRODUIT,
cours->         UNITES_STOCK "Stock", PRIX_UNITAIRE "Prix"
cours-> FROM PRODUITS P1
cours-> WHERE UNITES_STOCK > ( SELECT AVG(UNITES_STOCK)*2
cours(>         FROM PRODUITS P2
cours(>         WHERE P2.CODE_CATEGORIE = P1.CODE_CATEGORIE);
```

Cat	nom_produit	Stock	Prix
2	Grandma's Boysenberry Spread	120	125.00
4	Queso Manchego La Pastora	86	190.00
3	NuNuCa Nuß-Nougat-Creme	76	70.00
8	Boston Crab Meat	123	92.00
4	Raclette Courdavault	79	275.00
4	Geitost	112	13.00
3	Valkoinen suklaa	65	81.00
2	Sirop d'érable	113	143.00
1	Rhönbräu Klosterbier	125	39.00
2	Syrup	100	7.50
9	Boysenberry Spread	100	18.75
1	Green Tea	125	2.00
7	Potato Chips	200	0.49
...			

Dans l'exemple précédent la synchronisation entre la requête principale et la sous-requête est indiquée ici par l'utilisation, dans la sous-requête, de la colonne CODE_CATEGORIE de la table PRODUITS de la requête principale.

L'opérateur EXISTS

Une des formes particulière de la sous-requête synchronisée est celle testant l'existence de lignes de valeurs répondant à telle ou telle condition.

L'opérateur « **EXISTS** » permet de construire un prédicat évalué à « **TRUE** » si la sous-requête renvoie au moins une ligne.



```
cours=> SELECT SOCIETE, REF_PRODUIT, SUM(PORT)
cours-> FROM CLIENTS CL, COMMANDES CO, DETAILS_COMMANDES DC
cours-> WHERE CL.CODE_CLIENT = CO.CODE_CLIENT
cours-> AND CO.NO_COMMANDE = DC.NO_COMMANDE
cours-> AND CO.DATE_COMMANDE > '28/06/2011'
cours-> AND EXISTS ( SELECT * FROM PRODUITS PR, FOURNISSEURS FR
cours(> WHERE PR.NO_FOURNISSEUR = FR.NO_FOURNISSEUR
cours(> AND DC.REF_PRODUIT = PR.REF_PRODUIT
cours(> AND FR.VILLE = CL.VILLE)
cours-> GROUP BY SOCIETE,REF_PRODUIT;
```

societe	ref_produit	sum
Alfreds Futterkiste	25	64.80
Mère Paillard	54	76.80
Queen Cozinha	24	81.40
Alfreds Futterkiste	26	120.40
Mère Paillard	109	63.00
Comércio Mineiro	24	85.10
Alfreds Futterkiste	104	55.60
North/South	3	93.30

(8 lignes)

Dans l'exemple précédent, la requête affiche les clients, numéro de commande, référence produit et les frais de port pour les produits achetés par les clients qui habitent dans la même ville que le fournisseur.

Il est à noter que la projection totale (*) de la sous-requête est sans signification, puisque seul compte le fait que la sous-requête renvoie ou non une ligne. La projection peut donc être une constante quelconque, par exemple :



```
cours=> SELECT SOCIETE, REF_PRODUIT, SUM(PORT)
cours-> FROM CLIENTS CL, COMMANDES CO, DETAILS_COMMANDES DC
cours-> WHERE CL.CODE_CLIENT = CO.CODE_CLIENT
cours-> AND CO.NO_COMMANDE = DC.NO_COMMANDE
cours-> AND CO.DATE_COMMANDE > '25/06/2011'
cours-> AND EXISTS ( SELECT 'constante'
cours(> FROM PRODUITS PR, FOURNISSEURS FR
cours(> WHERE PR.NO_FOURNISSEUR = FR.NO_FOURNISSEUR
cours(> AND DC.REF_PRODUIT = PR.REF_PRODUIT
cours(> AND FR.VILLE = CL.VILLE)
cours-> AND EXISTS ( SELECT 'constante'
cours(> FROM EMPLOYES EM, COMMANDES CO1
cours(> WHERE EM.NO_EMPLOIE = CO1.NO_EMPLOIE
cours(> AND REGION LIKE
cours(> (SELECT REGION || '%' FROM EMPLOYES
cours(> WHERE FONCTION = 'Vice-Président'
cours(> AND NOM = 'Brasseur')
cours(> AND CO.NO_COMMANDE = CO1.NO_COMMANDE)
cours-> GROUP BY SOCIETE, REF_PRODUIT;
```

societe	ref_produit	sum
---------	-------------	-----

Lehmanns Marktstand	77	57.00
Alfreds Futterkiste	25	64.80
Lehmanns Marktstand	29	57.00
Alfreds Futterkiste	26	120.40
Around the Horn	3	148.80
Lehmanns Marktstand	64	131.20
Lehmanns Marktstand	75	57.00
Around the Horn	2	120.70
North/South	3	93.30
Around the Horn	1	52.10
Alfreds Futterkiste	104	55.60

```

cours=> SELECT NOM||' '||PRENOM EMPLOYE
cours-> FROM EMPLOYES
cours-> WHERE NO_EMPLOYE IN
cours-> ( SELECT NO_EMPLOYE
cours(> FROM COMMANDES CO1, CLIENTS CL, DETAILS_COMMANDES DC
cours(> WHERE CO1.CODE_CLIENT = CL.CODE_CLIENT
cours(> AND CO1.NO_COMMANDE = DC.NO_COMMANDE
cours(> AND PORT >
cours(> ( SELECT AVG(PORT)*1.38 FROM COMMANDES CO2
cours(> WHERE CO1.CODE_CLIENT = CO2.CODE_CLIENT
cours(> AND EXTRACT(YEAR FROM CO1.DATE_COMMANDE) =
cours(> EXTRACT(YEAR FROM CO2.DATE_COMMANDE))
cours(> AND EXISTS
cours(> ( SELECT 'constante'
cours(> FROM PRODUITS PR1
cours(> WHERE UNITES_STOCK >
cours(> ( SELECT AVG(UNITES_STOCK)*2
cours(> FROM PRODUITS PR2
cours(> WHERE PR1.NO_FOURNISSEUR =
cours(> PR2.NO_FOURNISSEUR)
cours(> AND PR1.REF_PRODUIT = DC.REF_PRODUIT));
      employe
-----
Kremser Arnaud
Poidatz Benoît
Gregoire Renée
Mure Guy
Urbaniak Isabelle
King Robert
Marielle Michel
Viry Yvan
Regner Charles

```


La clause WITH

La clause « **WITH** » permet d'assigner à une ou plusieurs sous-requêtes un alias afin de pouvoir l'utiliser à différents endroits dans la requête principale ou dans les sous-requêtes utilisées dans celle-ci.

La syntaxe de la clause est la suivante :

```
WITH alias_sous-requête AS ( sous-requête )[,...]
SELECT ...
```

L'exemple suivant affiche les sociétés qui ont la plus grande somme des frais de port par année et par trimestre.



```
cours=> WITH SAT_P AS
cours->      ( SELECT SOCIETE,
cours(>          EXTRACT(YEAR FROM DATE_COMMANDE) ANNEE,
cours(>          EXTRACT(MONTH FROM DATE_COMMANDE) MOIS,
cours(>          SUM(PORT) PORT
cours(>      FROM COMMANDES CO1, CLIENTS CL
cours(>      WHERE CO1.CODE_CLIENT = CL.CODE_CLIENT
cours(>      GROUP BY SOCIETE, ANNEE, MOIS )
cours-> SELECT SOCIETE, ANNEE, MOIS, TO_CHAR(PORT,'9999D00') PORT
cours-> FROM SAT_P
cours-> WHERE PORT IN ( SELECT MAX(PORT) FROM SAT_P
cours(>          GROUP BY ANNEE, MOIS )
cours-> ORDER BY ANNEE, MOIS;
```

societe	annee	mois	port
Paris spécialités	2010	1	1265,10
Ernst Handel	2010	2	1207,60
Spécialités du monde	2010	3	1417,10
Océano Atlántico Ltda.	2010	4	1396,60
LILA-Supermercado	2010	5	1296,50
Frankenversand	2010	6	1369,50
Wolski Zajazd	2010	7	1395,80
Consolidated Holdings	2010	8	1333,70
Drachenblut Delikatessen	2010	9	1391,10
Folk och få HB	2010	10	1485,20
La corne d'abondance	2010	11	1538,00
LINO-Delicateses	2010	12	1405,80
Folies gourmandes	2011	1	1929,80
QUICK-Stop	2011	2	1790,70
Queen Cozinha	2011	3	1794,50
Hungry Coyote Import Store	2011	4	1528,80
Bólido Comidas preparadas	2011	5	1780,50
Cactus Comidas para llevar	2011	6	1796,80

5

La mise à jour des données

Insertion d'une ligne

La commande « **INSERT** » permet d'insérer une ligne dans une table en spécifiant les valeurs à insérer par la syntaxe :

```
INSERT INTO NOM_TABLE [(COLONNE_1[,...])]
VALUES (EXPRESSION_1[,...]);
```

NOM_TABLE	La table dans laquelle la requête insère un enregistrement et seulement un enregistrement.
COLONNE_N	La liste des noms de colonnes de la table qui font l'objet d'une insertion ; elle est optionnelle. Toute colonne qui ne se trouve pas dans la liste reçoit la valeur « NULL ». En l'absence d'une liste de colonnes, des valeurs doivent être spécifiées pour toutes les colonnes de la table dans l'ordre défini lors de la création de la table.
EXPRESSION_N	L'expression doit être évaluée avec succès pour chacune des colonnes de la table. Les valeurs possibles sont : une constante, le résultat de l'expression, la valeur nulle « NULL ».

La requête suivante permet d'insérer une ligne dans la table **CATEGORIES** en spécifiant les valeurs à insérer sous forme des constantes.

Les opérations de mise à jour des données doivent tenir compte des contraintes des tables. Dans l'exemple suivant vous pouvez constater la violation de la contrainte « **PRIMARY KEY** » définie sur la colonne **CODE_CATEGORIE**.



```
cours=> INSERT INTO CATEGORIES ( CODE_CATEGORIE,NOM_CATEGORIE,
cours(> DESCRIPTION )
cours-> VALUES( 11, 'Viandes et Poissons',' Viandes et Poissons ') ;
INSERT 0 1
cours=> SELECT CODE_CATEGORIE, NOM_CATEGORIE FROM CATEGORIES;
```

code_categorie	nom_categorie
1	Boissons
2	Condiments
3	Desserts
4	Produits laitiers
5	Pâtes et céréales
6	Viandes
7	Produits secs
8	Poissons et fruits de mer
9	Conserves
10	Viande en conserve
11	Viandes et Poissons

```
(11 lignes)

cours=> INSERT INTO CATEGORIES ( CODE_CATEGORIE,NOM_CATEGORIE,
cours(> DESCRIPTION )
cours-> VALUES( 11, 'Viandes et Poissons',' Viandes et Poissons ') ;
ERREUR: la valeur d'une clé dupliquée rompt la contrainte unique «
categories_pk »
DÉTAIL : La clé « (code_categorie)=(11) » existe déjà.
```

Attention



Si vous tentez d'insérer une valeur qui dépasse la largeur d'une colonne de type caractère ou l'étendue d'une colonne de type numérique, vous obtenez un message d'erreur. Vous devez respecter les contraintes définies pour vos colonnes.

L'expression « **DEFAULT** » permet de définir une valeur par défaut pour la colonne, qui sera prise en compte si aucune valeur n'est spécifiée dans une commande « **INSERT** ». Elle est spécifiée à la création de la table et peut être une constante, une pseudocolonne « **USER** », « **SYSDATE** » ou tout simplement une expression.

Il est également possible d'insérer pour une colonne une valeur « **NULL** » de manière explicite.



```
cours=> ALTER TABLE EMPLOYES
cours->     ALTER COLUMN DATE_EMPAUCHE SET DEFAULT '2014-01-01';
ALTER TABLE
cours=> INSERT INTO EMPLOYES ( NO_EMPLOYE, NOM, PRENOM, FONCTION,
cours(>                                TITRE, DATE_NAISSANCE,
cours(>                                DATE_EMPAUCHE, SALAIRE, COMMISSION )
cours-> VALUES                                ( 200,'BIZOI', 'Razvan', 'Formateur',
cours(>                                'M.','1965-02-03', DEFAULT, 10000, NULL);
INSERT 0 1
cours=> SELECT NO_EMPLOYE,DATE_EMPAUCHE, COMMISSION
cours-> FROM EMPLOYES WHERE NO_EMPLOYE = 200;
no_employe | date_embauche | commission
-----+-----+-----
          200 | 2014-01-01    |
(1 ligne)
```

La requête précédente effectue l'insertion d'une ligne dans la table EMPLOYES en spécifiant les valeurs à insérer sous forme de constantes, ainsi que l'expression « **DEFAULT** » pour la colonne DATE_EMPAUCHE et précise de manière explicite la valeur « **NULL** » pour la colonne COMMISSION.

La commande « **INSERT** » permet d'insérer des données qui ont été sélectionnées dans une ou plusieurs tables.



```
INSERT INTO CLIENTS (CODE_CLIENT,SOCIETE,ADRESSE,VILLE,
                     CODE_POSTAL,PAYS,TELEPHONE)
SELECT UPPER(SUBSTR(REPLACE(SOCIETE,' ',''),1,5)),
        SOCIETE,ADRESSE,VILLE,CODE_POSTAL,PAYS, TELEPHONE
FROM FOURNISSEURS
WHERE PAYS = 'France';
```

Dans l'exemple précédent, les données extraites de la table FOURNISSEURS sont insérées dans la table CLIENTS. Notez que la clause « **WHERE** » de l'instruction « **SELECT** » peut extraire une ou plusieurs lignes. Vous remarquerez que vous n'êtes pas tenu d'insérer telles qu'elles les valeurs sélectionnées ; vous pouvez les modifier en utilisant des fonctions de chaîne, de date, ou numériques. Les valeurs insérées représentent le résultat de ces fonctions.

Modification des données

La commande « **UPDATE** » modifie les valeurs d'une ou de plusieurs colonnes, dans une ou plusieurs lignes existantes d'une table.



```
cours=# UPDATE EMPLOYES
cours-# SET SALAIRE      = SALAIRE*1.1,
cours-#      COMMISSION  = COMMISSION*1.2 ;
UPDATE 112
```

Dans l'exemple précédent les salaires sont augmentés de 10% et les commissions de 20% pour l'ensemble des enregistrements de la table EMPLOYES.

Comme vous pouvez le constater, l'expression peut faire référence aux anciennes valeurs des colonnes de la ligne.



```
cours=> SELECT CODE_CLIENT, SOCIETE, ADRESSE
cours-> FROM CLIENTS WHERE CODE_CLIENT = 'BLONP';
code_client |      societe      |      adresse
-----+-----+-----
BLONP      | Blondel père et fils | 24, place Kléber
(1 ligne)

cours=> UPDATE CLIENTS
cours-> SET ADRESSE = '104, rue Mélanie'
cours-> WHERE CODE_CLIENT = 'BLONP';
UPDATE 1
cours=> SELECT CODE_CLIENT, SOCIETE, ADRESSE
cours-> FROM CLIENTS WHERE CODE_CLIENT = 'BLONP';
code_client |      societe      |      adresse
-----+-----+-----
BLONP      | Blondel père et fils | 104, rue Mélanie
(1 ligne)
```

Dans l'exemple précédent, la modification porte seulement sur le client 'BLONP' qui est le seul enregistrement de la table CLIENTS qui respecte la clause « **WHERE** ».



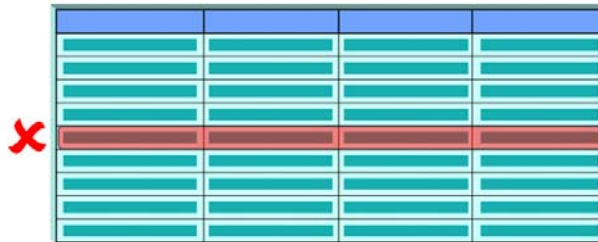
```
cours=# SELECT NOM, SALAIRE FROM EMPLOYES
cours-# WHERE NOM LIKE 'Peacock';
nom | salaire
-----+-----
Peacock | 6710.00

cours=# UPDATE EMPLOYES
cours-# SET SALAIRE = ( SELECT AVG(SALAIRE)
cours-#      FROM EMPLOYES
cours-#      WHERE FONCTION LIKE 'Rep%' )
cours-# WHERE NOM LIKE 'Peacock';
UPDATE 1
cours=# SELECT NOM, SALAIRE FROM EMPLOYES
cours-# WHERE NOM LIKE 'Peacock';
nom | salaire
-----+-----
Peacock | 8284.67
```

Suppression des données

Suppression des données

```
DELETE DETAILS_COMMANDES
WHERE NO_COMMANDE = 11077;
```



L'instruction « **DELETE** » supprime une ou plusieurs lignes d'une table.

DELETE FROM NOM_TABLE [WHERE PREDICAT];

NOM_TABLE Table dans laquelle la requête supprime un ou plusieurs enregistrements suivant la clause WHERE.

WHERE Clause agissant de façon analogue à la clause WHERE de l'ordre « **SELECT** » et qui permet d'indiquer les lignes concernées par la suppression.

```
cours=> DELETE FROM DETAILS_COMMANDES WHERE NO_COMMANDE = 11077;
DELETE 25
```

Dans l'exemple précédent, les détails de la commandes 11077 sont effacés.

Note

Dans une commande « **DELETE** » en l'absence de clause « **WHERE** », l'ensemble des enregistrements de la table sont supprimés.


```
cours=> DELETE FROM DETAILS_COMMANDES ;
DELETE 2155
```

Dans l'exemple précédent, tous les enregistrements de la table DETAILS_COMMANDES sont effacés.

Contraintes d'intégrité

Contraintes d'intégrité

```
DELETE COMMANDES  
WHERE NO_EMPLOYE = 3;
```



```
DELETE COMMANDES  
*  
ERREUR à la ligne 1 :  
ORA-02292: violation de contrainte  
(STAGIAIRE.FK_DETAILS__COMMANDES_COMMANDE)  
d'intégrité - enregistrement fils existant
```

TSOFT - ORACLE 10g SQL et PL*SQL

Module 13 : Mise à jour des données - Diapo 13.14

Une requête de modification du contenu de la base de données « **INSERT** », « **UPDATE** » ou « **DELETE** », ne sera exécutée que si le résultat respecte toutes les contraintes d'intégrité définies sur cette base.



```
cours=# UPDATE EMPLOYES SET DATE_NAISSANCE = NULL  
cours=# WHERE NO_EMPLOYE = 2;  
ERREUR: une valeur NULL viole la contrainte NOT NULL de la colonne  
« date_naissance »  
DÉTAIL : La ligne en échec contient (2, 95, Destenay, Agnès,  
Représentant(e), Mme, null, 2001-04-04, 10890.00, 948.00, Danemark,  
Europe du Nord)
```

Dans l'exemple précédent la contrainte d'intégrité « **NOT NULL** » interdit la mise à jour de la colonne DATE_NAISSANCE.



```
cours=# DELETE FROM EMPLOYES WHERE NO_EMPLOYE = 2;  
ERREUR: UPDATE ou DELETE sur la table « employes » viole la  
contrainte de clé étrangère  
« comm_empl_fk » de la table « commandes »  
DÉTAIL : La clé (no_employe)=(2) est toujours référencée à partir de  
la table « commandes ».
```

Dans l'exemple précédent la contrainte d'intégrité référentielle interdit la suppression de l'enregistrement.

Atelier 12

Atelier 13

- La mise à jour des données
- Les mise à jour évoluées



Durée : 60 minutes

TSOFT - ORACLE 10g SQL et PL*SQL

Module 13 : Mise à jour des données - Diapo 13.20

Exercice n° 1 La mise à jour des données

Insérez une nouvelle catégorie de produits nommée « Légumes et fruits » tout en respectant les contraintes d'insertion et mise à jour de la table CATEGORIES, à savoir que le CODE_CATEGORIE doit être unique et que les colonnes NOM_CATEGORIE et DESCRIPTION doivent être renseignées. Affichez l'enregistrement inséré et validez la transaction.

Le fournisseur 'Nouvelle-Orléans Cajun Delights' est racheté par le fournisseur 'Grandma Kelly's Homestead'.

Créez un nouveau fournisseur qui s'appelle « Kelly » avec les mêmes coordonnées que le fournisseur 'Grandma Kelly's Homestead'.

Tous les produits livrés anciennement par les fournisseurs 'Nouvelle-Orléans Cajun Delights' et 'Grandma Kelly's Homestead' seront distribués par le nouveau fournisseur.

Effacez les deux anciens fournisseurs.

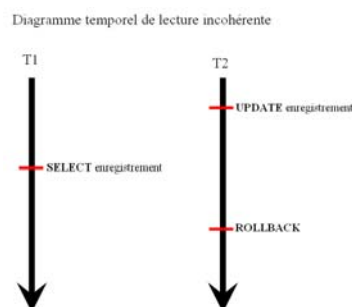
Affichez les produits livrés par le nouveau fournisseur et exécutez la commande suivante « **COMMIT ;** ». (La gestion des transactions fait l'objet du module suivant)

L'isolation

Une transaction peut s'isoler des autres transactions. C'est obligatoire dans les systèmes de base de données à utilisateurs multiples pour maintenir l'uniformité de données. La norme SQL-92 définit quatre niveaux d'isolation pour les transactions, s'étendant d'une uniformité très faible des données à une uniformité très forte. Pourquoi n'emploierait-on pas le niveau le plus fort pour toutes les transactions ? C'est une question de ressource. Plus le niveau d'isolation est fort, plus le verrouillage des ressources est intense. De plus, cela réduit le nombre d'utilisateurs pouvant accéder aux données simultanément. Comme vous pourrez le voir plus loin, le réglage du juste niveau est un compromis entre l'uniformité et la simultanéité.

Chacun de ces niveaux d'isolation peut produire certains effets secondaires connus sous le nom de **DIRTY READ** (lecture incohérente), **FUZZY READ** (lecture non répétitive) et **PHANTOM READ** (lecture fantôme). Seules les transactions avec un niveau d'isolation de type **SERIALIZABLE** sont immunisées.

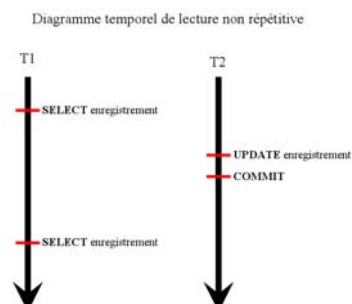
Lecture incohérente



La transaction T2 met à jour un enregistrement sans poser de verrous, celui-ci est disponible directement pour les autres transactions (exemple T1). Cette lecture peut être erronée, particulièrement si la transaction T1 annule l'effet de sa modification avec la commande « **ROLLBACK** ».

La lecture faite par la transaction T1 est fausse, on la nomme **DIRTY READ** (lecture incohérente).

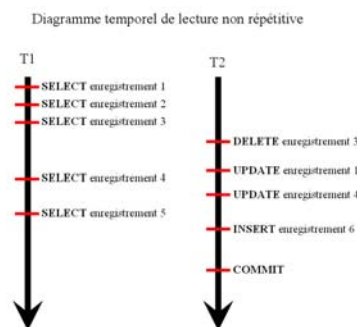
Lecture non répétitive



La transaction T1 consulte un enregistrement, (pour contrôler sa disponibilité par exemple) et dans la suite de la transaction le consulte à nouveau (pour le mettre à jour). Si une seconde transaction T2 modifie l'enregistrement entre les deux lectures, la transaction T1 va lire deux fois le même enregistrement, mais obtenir des valeurs différentes.

La lecture faite par la transaction est une lecture non répétitive **FUZZY READ**.

Lecture fantôme



La transaction T1 lit un ensemble d'enregistrements répondant à un critère donné à des fins de décompte ou d'inventaire. Parallèlement, la transaction T2 modifie les données. Au départ, il y a cinq enregistrements satisfaisant le critère. La transaction T1 lira en fin de compte quatre enregistrements, de façon tout à fait incohérente, puisque l'enregistrement 3 est supprimé, les enregistrements 4 et 1 sont modifiés et un nouvel enregistrement 6 apparaît.

La lecture faite par la transaction est une lecture non répétitive **PHANTOM READ**.

Les niveaux d'isolation

Le niveau d'isolation indique le comportement de la transaction par rapport aux autres transactions concurrentes. Plus le niveau d'isolation est faible, plus les autres transactions peuvent agir sur les données concernées par la première.

READ UNCOMMITTED

Le plus faible niveau restrictif permet à une transaction de lire des données qui ont été changées, mais pas encore validées.

READ COMMITTED

C'est le paramètre par défaut pour Oracle. Il assure que chaque requête dans une transaction lit seulement les données validées.

REPEATABLE READ

Ce niveau permet à une transaction de lire les mêmes données plusieurs fois avec la garantie qu'elle recevra les mêmes résultats à chaque fois. Vous pouvez le réaliser en plaçant des verrous sur les données qui sont lues, pour vous assurer qu'aucune autre transaction ne les modifiera pendant la durée de la transaction considérée.

SERIALIZABLE

Avec ce niveau le plus restrictif, une transaction ne prend en compte que les données validées avant le démarrage de la transaction, ainsi que les changements effectués par la transaction.

Le choix du niveau correct pour vos transactions est important. Bien que les transactions avec un niveau d'isolation de type **SERIALIZABLE** assurent une protection complète, elles affectent également la simultanéité en raison de la nature des verrous placés sur les données. C'est la nature de votre application qui détermine le meilleur niveau.

La valeur par défaut peut être adéquate, mais vous aurez peut-être besoin de la changer pour supporter votre application. Oracle vous permet de paramétrer l'isolation au niveau de la transaction ou de la session selon vos besoins. Pour définir le niveau d'isolation d'une transaction, employez l'instruction « **SET TRANSACTION** » au démarrage de votre transaction.

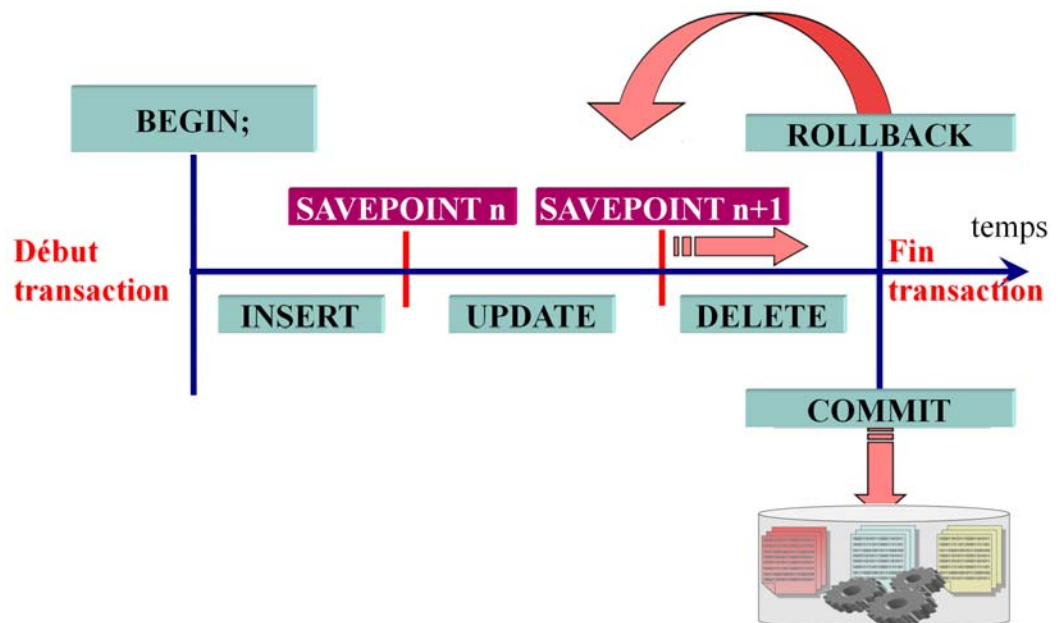
```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
SET TRANSACTION READ ONLY;
```

Début et fin de transaction



PostgreSQL prise en main

Module 1 : Présentation de l'environnement - Diapo 1.164

Une transaction commence à l'ouverture de la session ou à la fin de la précédente transaction. La toute première transaction débute au lancement du programme. Il n'existe pas d'ordre implicite de début de transaction.

```
BEGIN [ WORK | TRANSACTION ]
      [ ISOLATION LEVEL
        { SERIALIZABLE | REPEATABLE READ |
          READ COMMITTED | READ UNCOMMITTED }
        READ WRITE | READ ONLY [ NOT ] DEFERRABLE ] ;
```

Les points de repère « **SAVEPOINT** » sont des points de contrôle utilisés dans les transactions pour annuler partiellement l'une d'elles. Dans ce cas, un savepoint est défini par un identifiant et peut être référencé dans la clause « **ROLLBACK** ».



```
cours=> BEGIN TRANSACTION;
BEGIN
cours=> INSERT INTO CATEGORIES
cours-> ( CODE_CATEGORIE, NOM_CATEGORIE, DESCRIPTION )
cours-> VALUES ( 11,'Légumes et fruits','Légumes et fruits frais');
INSERT 0 1
cours=> SAVEPOINT POINT_REPERE_1;
SAVEPOINT
cours=> INSERT INTO FOURNISSEURS (NO_FOURNISSEUR, SOCIETE, ADRESSE,
cours(> VILLE, CODE_POSTAL, PAYS, TELEPHONE, FAX) VALUES (
cours-> 30,'Légumes de Strasbourg','104, rue Mélanie','Strasbourg'
cours(> ,67200,'France','03.88.83.00.68','03.88.83.00.62');
INSERT 0 1
cours=> SAVEPOINT POINT_REPERE_2;
SAVEPOINT
cours=> UPDATE PRODUITS SET CODE_CATEGORIE = 11
cours-> WHERE CODE_CATEGORIE = 2;
```

```

UPDATE 22
cours=> SAVEPOINT POINT_REPERE_3;
SAVEPOINT
cours=> UPDATE PRODUITS SET NO_FOURNISSEUR = 30
cours-> WHERE NO_FOURNISSEUR = 2;
UPDATE 7
cours=> SELECT NOM_PRODUIT, NO_FOURNISSEUR, CODE_CATEGORIE
cours-> FROM PRODUITS
cours-> WHERE NO_FOURNISSEUR = 30 AND
cours-> CODE_CATEGORIE = 11;

```

nom_produit	no_fournisseur	code_categorie
Chef Anton's Cajun Seasoning	30	11
Louisiana Hot Spiced Okra	30	11
Chef Anton's Gumbo Mix	30	11
Louisiana Fiery Hot Pepper Sauce	30	11

(4 lignes)

```

cours=> ROLLBACK TO POINT_REPERE_2;
ROLLBACK
cours=> SELECT NOM_PRODUIT, NO_FOURNISSEUR, CODE_CATEGORIE
cours-> FROM PRODUITS
cours-> WHERE NO_FOURNISSEUR = 2 AND
cours-> CODE_CATEGORIE = 2;

```

nom_produit	no_fournisseur	code_categorie
Chef Anton's Cajun Seasoning	2	2
Louisiana Hot Spiced Okra	2	2
Chef Anton's Gumbo Mix	2	2
Louisiana Fiery Hot Pepper Sauce	2	2

(4 lignes)

```

cours=> ROLLBACK TO POINT_REPERE_3;
ERREUR: aucun point de sauvegarde

```

!=(différent de)2-17, 4-19, 8-3
 %(symbole pourcentage)2-18
 %ROWTYPE.....7-6
 %TYPE.....7-6
 '(apostrophe)2-5
 -(tiret).....2-7, 2-9, 3-16
 *(étoile).....2-2, 2-7, 3-16
 .(point)2-5, 2-7
 /(barre oblique)2-7, 3-16
 ^=(différent de)8-3
 _ (tiret bas).....2-18
 ||(concaténation).....2-6
 ~=(différent de).....8-3
 +(plus).....2-7, 2-9, 3-16
 <(inférieur à).....4-19, 8-3
 =(inférieur ou égal).....4-19, 8-3
 <>(différent de).....2-17, 4-19, 8-3
 =(égal).....4-19, 8-3
 >(supérieur à).....4-19, 8-3
 >=(supérieur ou égal).....4-19, 8-3

A

ABS3-11
 AFTER10-2, 10-3, 10-6
 ALIAS2-2
 ALL2-2, 2-4, 4-8, 4-19, 4-23
 ALTER
 TABLE6-13
 ALTER TABLE
 DROP COLUMN6-14
 AND2-26
 ANY4-22
 AS2-2, 3-19
 ASC2-13
 ASCII.....3-8
 AUTONOMOUS_TRANSACTION9-16
 AVG3-25

B

BEFORE.....10-2, 10-3, 10-6
 BEGIN7-2, 9-6
 BETWEEN2-26, 8-3

C

Caractère générique2-18
 CASE2-10, 3-23, 8-5
 CAST3-19
 CEIL3-11
 CHAR3-8

CHECK.....6-6, 6-8
 CHR3-8
 Clause NOT NULL.....1-7
 Clés étrangères.....1-7
 Clés primaires1-7
 COALESCE.....2-10
 Colonnes1-6
 COMMIT9-15
 CONSTANT7-5
 Constante chaîne de caractère.....2-5
 Constante numérique2-5
 CONSTRAINT6-6
 Contrainte d'intégrité.....5-6
 Contrainte unique.....1-7
 COUNT3-26
 CREATE
 FUNCTION7-3
 INDEX6-19
 TRIGGER10-2
 USER6-23, 6-25
 CREATE TABLE
 AS SELECT6-5
 Syntaxe6-2
 CROSS JOIN4-2
 CURRENT OF9-21
 CURRENT_DATE3-14
 CURRENT_TIMESTAMP3-14
 CURSOR9-5, 9-14

D

DECLARE.....7-2, 9-5
 DECODE3-23
 DEFAULT5-3, 6-2, 6-13
 DELETE5-5, 5-6, 9-21
 DESC2-13
 DIFERENCE4-14
 DISTINCT2-2, 4-8, 4-9
 DROP
 FUNCTION7-9

E

END7-2
 ERROR
 ORA-010029-15
 ORA-014026-17
 ORA-300069-16
 EXCEPT4-14
 EXISTS.....4-27
 EXIT8-7
 EXTRACT.....3-14

F

FETCH 9-8, 9-15
 FLOOR 3-11
 Fonction
 Fonctions de groupe 3-25
 FOR 8-9, 9-9
 FOR UPDATE 9-14
 FOR UPDATE OF 9-19
 FOREIGN KEY 6-6
 FROM 2-2, 4-29
 FULL 4-6

G

GRANT 1-12
 GROUP BY 3-27, 3-28, 3-31, 6-16

H

HAVING 3-30, 3-31

I

IF-THEN-ELSE 8-3
 IN OUT 9-25
 IN_ 2-26, 4-19, 8-3, 9-24
 Index B-tree 6-19
 INITCAP 3-4
 INSERT 5-2, 5-3, 5-6, 6-2, 6-16
 INSTEAD OF 10-2
 INTERSECT 4-13
 INTO 7-7, 9-8
 IS NOT NULL 2-31
 IS NULL 2-23, 8-3

J

JOIN ON 4-4
 JOIN USING 4-3

L

LCD 1-9
 LDD 1-9, 1-12
 Le groupe 3-27
 LEFT 4-6
 LENGTH 3-8
 LID 1-9
 Lignes 1-6
 LIKE 2-18, 8-3
 LIMIT 2-11
 LMD 1-9, 2-2
 LONG 4-9
 LOOP 8-7, 9-9
 LOWER 3-4
 LPAD 3-4

M

MAKE_DATE 3-20
 MAKE_TIME 3-20
 Marqueur de position 2-18
 MAX 3-25
 MIN 3-25
 MOD 3-11

N

NATURAL JOIN 4-2
 NEW 10-8, 10-10
 NOT 2-30
 NOT BETWEEN 2-31
 NOT IN 2-30, 4-24
 NOT NULL 3-25, 5-6, 6-2, 6-6, 6-7, 6-13, 6-16, 7-5
 NOW 3-14
 NOWAIT 9-14
 NULL 2-13, 3-25, 3-26, 5-2, 5-3, 6-7, 6-13
 NULL FIRST 2-14
 NULL LAST 2-14

O

OF 9-14, 9-19
 OLD 10-8, 10-10
 OPEN 9-6
 OPEN..FOR 9-22, 9-24
 Opérateur
 AND 2-27
 BETWEEN 2-26, 8-3
 IN_ 2-26, 8-3
 IS NULL 2-23, 8-3
 LIKE 2-18, 8-3
 Opérateur de concaténation 2-6
 Opérateurs arithmétique 2-7
 Opérateurs de type DATE 2-9
 Opérateurs ensembliste 4-8
 Opérateurs logiques 2-18, 2-23, 2-26, 2-27
 ORDER BY 2-13, 3-31, 4-9, 4-16
 OUT 9-24
 OUTER JOIN 4-6

P

PL/SQL 1-11, 1-12
 POSITION 2-13
 POWER 3-11
 PREDICAT 2-16
 PRIMARY KEY 5-2, 6-6, 6-9, 6-11
 Pseudocolonne 2-9

R

RAISE EXCEPTION 10-4
 REF CURSOR 9-22
 REFERENCES 6-6, 6-11
 REGEXP_REPLACE 3-6

REPLACE3-6, 7-3, 10-2
RETURN 7-3, 9-5
REVOKE..... 1-12
RIGHT 4-6
ROLLBACK..... 9-15
ROUND..... 3-11
RPAD 3-4

S

SELECT
 Curseur 9-22
 FOR UPDATE..... 9-16
 INTO..... 7-7
 Le groupe..... 3-27
 Opérateurs ensembliste 4-8
 Projection..... 2-2
 Restriction..... 2-16
 Sélection 2-16
 Sous-requête 5-3
 Tri du résultat..... 2-13
SET 5-4
Sous-requête synchronisé 4-26
Sous-requêtes..... 4-16
SQL..... 1-2, 1-9, 1-11, 1-12, 2-2
START WITH 6-16
STDDEV 3-26
STRPOS 3-6
SUBSTRING..... 3-5
SUM 3-25
SYSDATE 2-9

T

TO_CHAR..... 3-19

TO_DATE 3-19
TRIM 3-4
TRUNC..... 3-11
TYPE
 REF CURSOR..... 9-22
Type de donnée
 CHARACTER 6-3
 INTERVAL DAY TO SECOND..... 3-17
 INTERVAL YEAR TO MONTH..... 3-17
 NUMBER 6-3
 VARCHAR..... 6-3
Types de données..... 1-6

U

UNION 4-10
UNIQUE..... 2-2, 6-6, 6-11
UPDATE 5-4, 5-6, 6-16, 9-21
USING 9-24

V

VALUES 5-2
VARIANCE..... 3-25

W

WAIT..... 9-14
WHEN 2-10, 3-23, 8-7, 10-2, 10-10
WHERE 2-16, 3-28, 3-30, 3-31, 5-4, 5-5
WHERE CURRENT OF 9-21
WHILE 8-8
WITH..... 4-30
WITH CHECK OPTION..... 6-17