SQL

Dominique Gonzalez
Université Lille3-Charles de Gaulle

SQL
par Dominique Gonzalez

Publié 5/10/2005

Table des matières

Chapitre 1. Pourquoi et comment ?

1.1. Pourquoi ce document ?

Ces pages ont pour origine un cours destiné aux étudiants de 2ème année de l'IUP IIES de l'université de Lille III-Charles de Gaulle, à Villeneuve d'Ascq, pour les années universitaires 2002-2003 et 2003-2004.

Elles ont ensuite été un peu remaniées et augmentées à l'occasion d'un cours destiné aux étudiants de 3ème année de la licence MIASHS.

Ces pages ne sont pas destinées à être un cours autonome : elles ne sont qu'un support de cours, et beaucoup de choses, qui sont transmises à l'oral pendant les cours, ne sont pas écrites.

L'environnement technique du cours est constitué de machines sous `linux`, les bases de données étant hébergées sur un serveur `PostgreSQL`, lui aussi sous `linux`. L'installation de ces logiciels ne sera pas abordée et ne fait pas partie du contenu du cours.

On n'abordera pas non plus la partie *analyse* (`Merise`, etc.) qui est pourtant foncdamentale dans le processus de création d'une base données. Cette partie sera vue ailleurs. L'objectif de ce cours est seulement la découverte de `SQL`.

1.2. Comment a-t-il été construit ?

Ce polycopié a été rédigé au format `DocBook` :

- Le texte source a été écrit au format XML avec `emacs`, en respectant la DTD de `DocBook`.
- Le code source a été compilé au format PDF avec `openjade` et au format HTML avec `xsltproc`.
- La version que vous avez devant les yeux a été compilée le 5/10/2005 .

1.3. Où trouver ce document ?

Ce document est disponible sous plusieurs formats sur le web:

- Un seul document HTML : http://www.grappa.univ-lille3.fr/polys/sql/sql.html (Lourd à charger, mais facile à sauvegarder ou à imprimer)
- Plusieurs pages HTML : http://www.grappa.univ-lille3.fr/polys/sql/index.html (Plus faciles à consulter)
- Une version HTML sans feuilles de style : http://www.grappa.univ-lille3.fr/polys/sql/book1.htm (Quelle idée ? Mais si vous y tenez...)
- PDF : http://www.grappa.univ-lille3.fr/polys/sql/sql.pdf

1.4. `QBullets`

Les petites images animées qui illustrent les liens de la version web de ce document proviennent de QBullets

[1]

1. http://www.matterform.com/

Chapitre 2. Qu'est-ce que SQL ?

2.1. Avant-propos

Ce premier chapitre est extrait d'un polycopié réalisé par Rémi Gilleron et Marc Tommasi dans le cadre de leur cours sur l' architecture client-serveur.

Je me suis contenté d'en modifier la mise en page.

Vous pouvez consulter l'original sur internet[1].

Ce polycopié était lui-même fortement inspiré de l'excellent ouvrage « *SQL 2* », de Christian Marée et Guy Ledant, chez Armand Colin.

2.2. Intérêt de SQL

Tous les systèmes de gestion de données utilisent SQL pour l'accès aux données ou pour communiquer avec un serveur de données. SQL (Standard Query Language) est né à la suite des travaux mathématiques de Codd, travaux qui ont fondé les bases de données relationnelles. SQL, défini d'abord chez IBM, a subi trois tentatives de normalisation en 86, 89 et 92 (SQL 2 ou SQL 92). Nous présentons trois raisons fondamentales qui justifient l'utilisation de SQL.

- D'une part, la structuration et la manipulation des données sont devenues très complexes. Pour une application de taille moyenne, la base de données contient fréquemment plus de trente tables fortement interconnectées. Il est donc hors de question de manipuler les données de façon algorithmique traditionnelle. Une requête SQL dans un langage logique simple remplace donc bien avantageusement plusieurs dizaines de lignes d'un langage de programmation tel C ou Cobol.
- D'autre part, l'architecture client-serveur est omniprésente. Tandis que la station client exécute le code de l'application en gérant, en particulier, l'interface graphique, le serveur optimise la manipulation et le contrôle des données. De plus, les applications doivent être portables et gérer des données virtuelles, c'est-à-dire émanant de n'importe quel serveur. Développer une application dans un environnement hétérogène n'est possible que parce que la communication entre l'applicatif client et le serveur est réalisée par des primitives SQL normalisées.
- Enfin, les applications à développer (même sur un PC) sont devenues de plus en plus complexes. Le profil du programmeur a fortement changé. Il doit maintenant traiter des données de plus en plus volumineuses, intégrer les techniques de manipulation des interfaces, maîtriser la logique événementielle et la programmation orientée objet, tout celà dans un contexte d'architecture client-serveur où se cotoient les systèmes d'exploitation et les protocoles de réseaux hétérogènes. L'accès et la manipulation des données ne sont que l'un des aspects de la conception et de la réalisation de programmes. On cherche donc à acquérir un environnement de développement performant qui prend en charge un grand nombre de tâches annexes. Des outils de développement sont apparus pour permettre au développeur de se concentrer sur l'application proprement dite : générateurs d'écrans, de rapports, de requêtes, d'aide à la conception de programme, de connection à des bases de données distantes via des réseaux. Dans tous ces outils, la simplicité et la standardisation de SQL font que SQL est utilisé chaque fois qu'une définition, une manipulation, ou un contrôle de données est nécessaire. SQL est donc un élément central entre les divers composants d'un environnement de développement dans une architecture client-serveur.

2.3. SQL dans l'architecture en couches des SGBD

Dans la phase d'analyse de systèmes d'information, on considère différents niveaux d'abstraction du système d'information : le niveau conceptuel, le niveau logique ou organisationnel et enfin, le niveau physique ou opérationnel.

Nous allons considérer ici différents niveaux de perception d'une base de données relationnelle. Un SGBD est fréquemment décrit par une structure en couches correspondant à des perceptions différentes des données, associées à des tâches différentes pour différents acteurs.

1. http://www.grappa.univ-lille3.fr/polys/frime

Pour plus de simplicité, nous distinguerons trois types d'acteurs : les administrateurs de la base de donnée, les développeurs et les utilisateurs. Au niveau externe, proche de l'utilisateur, la perception est totalement indépendante du matériel et des techniques mises en oeuvre, tandis qu'au niveau le plus intérieur, se trouvent les détails de l'organisation sur disque et en mémoire.

Le *schéma logique* est l'ensemble de toutes les données pertinentes, toutes applications confondues. Il est rendu conforme à un modèle de représentation des données, et est totalement indépendant de la technologie utilisée. Nous choisissons le modèle relationnel. Ce niveau a un inconvénient : toutes les données sont accessibles à tout le monde. Cet ensemble vaste de données est trop touffu. Il est préférable de montrer à l'utilisateur (et au programmeur) une vue plus simple des données.

On constitue ainsi des *schémas externes*. Par exemple, le gestionnaire du stock n'est concerné que par les données décrivant les articles en stock. S'il ne manipule que des bordereaux d'entrée, des bordereaux de sortie, et des fiches d'état du stock, ceux-ci constituent son schéma de perception externe.

Le *schéma interne* fournit une perception plus technique des données.

Enfin le *schéma physique* est dépendant du matériel et du logiciel de base.

Niveau externe

: Au niveau externe, les utilisateurs et développeurs d'application ont une perception limitée de la base de données. On parle de *vue*. Une vue peut être considérée comme une restriction du schéma logique à un type d'utilisateur. Ce niveau concerne les utilisateurs et les développeurs.

Niveau logique

: Traduction dans le modèle relationnel du schéma conceptuel. On précise à ce niveau les tables, les relations entre tables, les contraintes d'intégrité, les vues et les droits par groupe d'utilisateurs. Ce niveau concerne l'administrateur et les développeurs.

Niveau interne

: On définit les index et tous les éléments informatiques susceptibles d'optimiser les ressources et les accès aux données. Ce niveau concerne l'administrateur.

Niveau physique

: On y précise tout ce qui dépend du matériel et du système d'exploitation. Ce niveau concerne l'administrateur.

Cette découpe en niveaux présente les avantages suivants :

- Les applications développées sont indépendantes du niveau interne. Tout changement de stratégie d'accès, ou d'organisation des données entraîne une modification au niveau interne, mais le schéma logique reste inchangé. Par exemple, une requête SQL précise le QUOI sans se préoccuper du COMMENT.
- La distinction externe/logique assure (en partie) l'indépendance entre les applications et le niveau logique. Par exemple on peut enrichir le schéma logique sans modifier les applications existantes pour toutes les vues non concernées par les modifications apportées au schéma logique.
- La distinction logique/interne permet de modifier les optimisations d'accès aux données. Par exemple, si une application a des performances insuffisantes, il est possible d'optimiser les accès (en introduisant de nouveaux index, par exemple) et d'augmenter les performances sans modifier l'application.
- La distinction interne/physique permet une meilleure portabilité car seule la partie physique est dépendante du matériel et du système d'exploitation.

2.4. SQL : Principes d'une Base de Données Relationnelle

Le terme *relationnel* provient de la définition mathématique d'algèbre relationnelle (Codd 70). Une relation est un ensemble de tuples (tuple est la généralisation de couple à un nombre quelconque d'éléments. Les couples, les triplets, les quadrupets, etc sont des tuples) de données, on peut alors définir des opérations algébriques sur les relations (voir la section Section 2.7). Nous parlerons dans la suite de table, et non pas de relation. Une *table* est un ensemble de tuples de données distincts deux à deux. Une table est constituée d'une *clef primaire* et de plusieurs *attributs* (ou colonnes) qui dépendent de cette clef. La clef primaire d'une table est un attribut ou un groupe d'attributs de la table qui détermine tous les autres de façon unique. Une

table possède toujours une et une seule clef primaire. Par contre, une table peut présenter plusieurs clefs candidates qui pourraient jouer ce rôle. Le *domaine d'un attribut* est l'ensemble des valeurs que peut prendre cet attribut. Le domaine est constitué d'un type, d'une longueur et de contraintes qui réduisent l'ensemble des valeurs permises. Une clef étrangère dans une table est une clef primaire ou candidate dans une autre table. Les *contraintes d'intégrité* font partie du schéma logique. Parmi celles-ci, on distingue :

- les *contraintes de domaine* qui restreignent l'ensemble des valeurs que peut prendre un attribut dans une table,
- les *contraintes d'intégrité d'entité* qui précisent qu'une table doit toujours avoir une clef primaire et
- les *contraintes d'intégrité référentielle* qui précisent les conditions dans lesquelles peuvent être ajoutés ou supprimés des enregistrements lorsqu'il existe des associations entre tables par l'intermédiaire de clefs étrangères.

Exemple 2-1. Contraintes d'intégrité

```
CLIENTS (cltnum, cltnom, cltpnom, cltloc, cltca, clttype)
COMMANDES (cmdnum, cmdclt, cmddate, cmdvnd)
LIGCOMMANDES (ldccmd, ldcart, ldcqte)
ARTICLES (artnum, artnom, artpv, artcoul)
```

Les identifiants sont en **gras**, les clefs étrangères en *italiques*. Une contrainte d'intégrité référentielle est, par exemple, l'obligation de la présence d'un client pour une comande. C'est-à-dire encore qu'à un enregistrement dans la table `COMMANDES` doit correspondre un enregistrement de la table `CLIENTS` tel que `COMMANDES.cmdclt=CLIENTS.cltnum`.

2.5. Architecture client-serveur et communication par SQL

Nous allons d'abord voir les différents types d'application possibles pour la gestion de données distantes en commençant par les trois formes les plus simples.

Monoposte

La base de données se trouve sur un poste et n'est pas accessible en réseau. Il faut, dans ce cas, penser à la notion de sécurité si plusieurs utilisateurs peuvent interroger la base (suppression accidentelle d'enregistrements).

Multiposte, basée sur des terminaux liés à un site central

C'est l'informatique multiposte traditionnelle. La gestion des données est centralisée. Les applications ont été écrites par le service informatique et seules ces applications peuvent interroger le serveur.

Multiposte, basée sur un serveur de fichiers

C'est la première forme (la plus simple) d'architecture client-serveur. Si l'applicatif sur un PC souhaite visualiser la liste des clients habitant Paris, tous les enregistrements du fichier CLIENT transitent sur le réseau, entre le serveur et le client, la sélection (des clients habitant Paris) est faite sur le PC. Le trafic sur le réseau est énorme et les performances se dégradent lorsque le nombre de clients augmente. Les serveurs de fichiers restent très utilisés comme serveurs d'images, de documents, d'archives.

De nouveaux besoins sont apparus :

- diminuer le trafic sur le réseau pour augmenter le nombre de postes sans nuire au fonctionnement,
- traiter des volumes de données de plus en plus grand,
- accéder de façon transparente à des données situées sur des serveurs différents,
- accéder aux données de façon ensembliste, même si les données sont distantes, afin de diminuer le travail du programmeur,
- adopter des interfaces graphiques de type `Windows` pour les applicatifs clients.

2.5.1. Bases de données en client-serveur

Dans une architecture client-serveur, un applicatif est constitué de trois parties : l'interface utilisateur, la logique des traitements et la gestion des données. Le client n'exécute que l'interface utilisateur et la logique des traitements, laissant au serveur de bases de données la gestion complète des manipulations de données.

- Le serveur de bases de données fournit des services aux processus clients. Les tâches qu'il doit prendre en compte sont : la gestion d'une mémoire cache, l'exécution de requêtes exprimées en SQL, exécuter des requêtes mémorisées, la gestion des transactions, la sécurité des données.
- Le client doit ouvrir une connection pour pouvoir profiter des services du serveur. Il peut ouvrir plusieurs connections simultanées sur plusieurs serveurs. Le client peut soumettre plusieurs requêtes simultanément au serveur et traiter les résultats de façon asynchrone.
- Communication entre le client et le serveur. Puisque l'application doit pouvoir se connecter à divers serveurs de façon transparente, le langage de communication SQL doit être compatible avec la syntaxe SQL de chaque serveur pressenti. Or, malgré les normes, les dialectes SQL sont nombreux et parfois source d'incompatibilité. La seule façon de permettre une communication plus large est d'adopter un langage SQL standardisé de communication. Une couche fonctionnelle du client traduit les requêtes du dialecte SQL client en SQL normalisé. La requête transformée est envoyée au serveur. Celui-ci traduit la requête dans le dialecte SQL-serveur et l'exécute. Le résultat de la requête suit le chemin inverse. Le langage de communication normalisé le plus fréquent est l'ODBC (Open DataBase Connectivity) de Microsoft. Signalons également IDAPI (Integrated Database Application Programming Interface) de Borland. De plus, ces programmes permettent d'interroger des bases de données autres que relationnelles.

2.5.2. Les serveurs de transactions

Une transaction correspond à une procédure SQL, i.e. un groupe d'instructions SQL. Il y a un seul échange requête/réponse pour la transaction. Le succès ou l'échec concerne l'ensemble des instructions SQL. Ces serveurs sont essentiellement utilisés pour l'informatique de production (ou opérationnelle) pour laquelle la rapidité des temps de réponse est importante sinon essentielle.

2.6. Structure générale du langage SQL

Les instructions essentielles SQL se répartissent en trois familles fonctionnellement distinctes et trois formes d'utilisation :

Selon la norme SQL 92, le *SQL interactif* permet d'exécuter une requête et d'en obtenir immédiatement une réponse. Le *SQL intégré* (ou module SQL) permet d'utiliser SQL dans un langage de troisième génération (C, Cobol, ...), les instructions permettant essentiellement de gérer les curseurs. Le *SQL dynamique* est une extension du SQL intégré qui permet d'exécuter des requêtes SQL non connues au moment de la compilation. Hors norme, SQL est utilisable sous la forme de librairies de fonctions API (exemple : ODBC). Nous nous limitons au SQL interactif.

Dans le SQL interactif, le *LDD* (Langage de Définition de données) permet la description de la structure de la base (tables, vues, index, attributs, ...). Le dictionnaire contient à tout moment le descriptif complet de la structure de données. Le *LMD* (Langage de Manipulation de Données) permet la manipulation des tables et des vues. Le *LCD* (Langage de Contrôle des Données) contient les primitives de gestion des transactions et des privilèges d'accès aux données. Le tableau ci-dessous vous donne les principales primitives SQL et leur classification. Nous nous intéresserons essentiellement au LMD et à la commande *SELECT*. Nous étudierons également quelques problèmes concernant les privilèges d'accès et les transactions.

Tableau 2-1. SQL interactif

LDD	LMD	LCD
CREATE	SELECT	GRANT
DROP	INSERT	REVOKE
ALTER	DELETE	CONNECT

LDD	LMD	LCD
	UPDATE	COMMIT
		ROLLBACK
		SET

2.7. SQL : un langage algébrique

Pour bien comprendre le langage SQL, nous allons brièvement exposer les principes sur lesquels repose ce langage (algèbre relationnelle).

Une table (relation) est un ensemble de tuples. On peut donc appliquer à une table les opérateurs algébriques usuels. Le résultat d'une opération ou requête est une nouvelle table qui est exploitable à son tour dans une nouvelle opération. Tous les opérateurs peuvent être dérivés de cinq primitives de base : la PROJECTION, la SÉLECTION, l'UNION, la DIFFÉRENCE et le PRODUIT. L'opérateur de JOINTURE qui peut être déduit des cinq primitives de base est cependant fondamental. (algèbre relationnelle).

La PROJECTION

permet de ne conserver que les attributs intéressants d'une table (sélection verticale). De plus, la projection élimine les répétitions de tuples résultant de cette sélection.

Exemple 2-2. Projection

```
CLIENTS2 = PROJECT CLIENTS OVER (cltnom, cltloc)
```

La SÉLECTION

permet de ne conserver que les tuples qui respectent une condition définie sur les valeurs des attributs (sélection horizontale).

Exemple 2-3. Sélection

```
BONSCLIENTS = SELECT CLIENTS WHERE cltca>10000
```

L'UNION

réalise l'union de plusieurs tables.

Exemple 2-4. Union

```
CLIENTS3 = SELECT CLIENTS WHERE cltloc = 'PARIS'
          UNION
           SELECT BONSCLIENTS WHERE cltloc='BRUXELLES'}
```

La DIFFÉRENCE

consiste à prendre les tuples appartenant à une table mais pas à une autre.

Exemple 2-5. Différence

```
CLIENTS4 = SELECT CLIENTS WHERE cltloc = 'BRUXELLES' EXCEPT BONSCLIENTS
```

Le PRODUIT

réalise la juxtaposition de tous les tuples de la première table avec chaque tuple de la seconde. Celà signifie que, si les deux tables ont respectivement M et N tuples, la table résultante aura $M \times N$ tuples. Cette opération présente peu d'intérêt mais combinée avec une sélection, on obtient une opération fondamentale : la JOINTURE.

La JOINTURE

N'est possible que sur deux tables possédant un attribut de domaine commun. Elle consiste à juxtaposer les tuples dont la valeur d'un attribut est égal dans les deux tables. C'est une primitive dérivée car elle peut être définie à l'aide des primitives précédentes (exercice laissé au lecteur).

Exemple 2-6. Jointure

```
CMDCLIENTS = COMMANDES JOIN CLIENTS ON cmdclt=cltnum}
```

Les primitives peuvent être combinées pour constituer des requêtes plus élaborées. La séquence d'opérateurs permettant de réaliser une requête élaborée devient assez vite complexe. Le langage SQL permet (heureusement) d'exprimer globalement une requête sans faire apparaître les tables et les primitives intermédiaires. Ce sera le moteur SQL qui sera chargé d'optimiser la requête.

Exemple 2-7. Requête

Une requête SQL qui permet de dresser la liste des noms des clients qui ont acheté des articles de moins de 200F est :

```
SELECT DISTINCT cltnom
  FROM clients, commandes, ligcommandes, articles
 WHERE cltnum=cmdclt
   AND artnum=lcdart
   AND cmdnum=ldccmd
   AND artpv < 200
```

Une combinaison de primitives permettant d'exécuter cette requête est :

```
TEMP1 = SELECT ARTICLES WHERE artpv < 200
TEMP2 = PROJECT TEMP1 OVER (artnum)
TEMP3 = PROJECT CLIENTS OVER (cltnum,cltnom)
TEMP4 = PROJECT COMMANDES OVER (cmdnum, cmdclt)
TEMP5 = PROJECT LIGCOMMANDES OVER (ldccmd,ldcart)
TEMP6 = TEMP2 JOIN TEMP5 ON artnum=ldcart
TEMP7 = PROJECT TEMP6 OVER (ldccmd)
TEMP8 = TEMP3 JOIN TEMP4 ON cltnum=cmdclt
TEMP9 = PROJECT TEMP8 OVER (cltnom, cmdnum)
TEMP10 = TEMP7 JOIN TEMP9 ON ldccmd=cmdnum
RESULTAT = PROJECT TEMP10 OVER (cltnom)}
```

Une telle séquence n'est, en général, pas unique. La séquence fournie est une des plus efficaces (le lecteur peut s'exercer à en trouver d'autres).

Chapitre 3. Présentation de PostgreSQL

3.1. Définition de PostgreSQL

Ce texte provient de la documentation PostgreSQL 7.4.8[1].

PostgreSQL est un système de gestion de bases de données relationnelles objet (ORDBMS) basé sur POSTGRES, Version 4.2, développé à l'université de Californie au département des sciences informatiques de Berkeley. POSTGRES a lancé de nombreux concepts rendus ensuite disponibles dans plusieurs systèmes de bases de données commerciales.

PostgreSQL est un descendant open-source du code original de Berkeley. Il supporte SQL92 et SQL99 tout en offrant de nombreuses fonctionnalités modernes :

- requêtes complexes ;
- clés étrangères ;
- déclencheurs (triggers) ;
- vues ;
- intégrité des transactions ;
- contrôle des accès concurrents (MVCC ou multiversion concurrency control).

De plus, PostgreSQL peut être étendu de plusieurs façons par l'utilisateur, par exemple en ajoutant de nouveaux

- types de données ;
- fonctions ;
- opérateurs ;
- fonctions d'agrégat ;
- méthodes d'indexage ;
- langages de procédure.

Et grâce à sa licence libérale, PostgreSQL peut être utilisé, modifié et distribué par tout le monde gratuitement quelque soit le but visé, qu'il soit privé, commercial ou académique.

3.2. Bref historique de PostgreSQL

Ce texte provient de la documentation PostgreSQL 7.4.8[2].

Le système de bases de données relationnel objet PostgreSQL est issu de POSTGRES, programme écrit à l'université de Californie à Berkeley. Avec plus d'une dizaine d'années de développement, PostgreSQL est la plus avancée des bases de données libres.

3.2.1. Le projet POSTGRES de Berkeley

Le projet POSTGRES, mené par le professeur Michael Stonebraker, était sponsorisé par l'agence des projets avancés de la Défense (DARPA, acronyme de Advanced Research Projects Agency), le bureau des recherches de l'armée (ARO, acronyme de Army Research Office), le NSF (acronyme de National Science Foundation) ainsi que ESL, Inc. L'implémentation de POSTGRES a commencé en 1986. Les concepts initiaux du système ont été présenté dans The design of POSTGRES[3] et la définition du modèle de données initial est apparu dans The POSTGRES data model[4]. Le concept du système de règles à ce moment était décrit dans The design of

1. http://traduc.postgresqlfr.org/pgsql-fr/preface.html#INTRO-WHATIS
2. http://traduc.postgresqlfr.org/pgsql-fr/history.html
3. http://s2k-ftp.cs.berkeley.edu:8000/postgres/papers/ERL-M85-95.pdf
4. http://s2k-ftp.cs.berkeley.edu:8000/postgres/papers/ERL-M87-13.pdf

the POSTGRES rules system[5]. L'architecture du gestionnaire de stockage était détaillée dans The design of the POSTGRES storage system[6].

Postgres a connu plusieurs versions majeures depuis. La première « démo » devint opérationnelle en 1987 et fut présenté en 1988 à la conférence ACM-SIGMOD. La version 1, décrite dans The implementation of POSTGRES[7], fut livrée à quelques utilisateurs externes en juin 1989. En réponse à une critique du premier mécanisme de règles (A commentary on the POSTGRES rules system[8]), celui-ci fut réécrit (On Rules, Procedures, Caching and Views in Database Systems[9]) dans la version 2, présentée en juin 1990. La version 3 apparut en 1991. Elle ajoutait le support de plusieurs gestionnaires de stockage, un exécuteur de requêtes amélioré et un gestionnaire de règles réécrit. La plupart des versions suivantes jusqu'à Postgres95 (voir Section 3.2.2) portèrent sur la portabilité et la fiabilité.

POSTGRES fut utilisé pour réaliser différentes applications de recherche et de production. Par exemple : un système d'analyse de données financières, un programme de suivi des performances d'un moteur à réaction, une base de données de suivi d'astéroïdes, une base de données médicale et plusieurs systèmes d'informations géographiques. POSTGRES a aussi été utilisé comme outil de formation dans plusieurs universités. Enfin, Illustra Information Technologies (devenu Informix[10], maintenant détenu par IBM[11]) a repris le code et l'a commercialisé. Fin 1992, POSTGRES devint le gestionnaire de données principal du projet de calcul scientifique Sequoia 2000[12] fin 1992.

La taille de la communauté d'utilisateurs doubla pratiquement durant l'année 1993. Il devint de plus en plus évident que la maintenance du code et le support nécessitaient de plus en plus de temps et d'énergie, qui auraient dûs être employés à des recherches sur les bases de données. Afin de réduire le poids du support, le projet POSTGRES de Berkeley se termina officiellement avec la version 4.2.

3.2.2. Postgres95

En 1994, Andrew Yu et Jolly Chen ajoutèrent un interpréteur de langage SQL à POSTGRES. Sous un nouveau nom, Postgres95 fut par la suite publié sur le Web, afin de devenir un descendant libre (open-source) du code source initial de POSTGRES, version Berkeley.

Le code de Postgres95 était complètement compatible avec le C ANSI et 25% moins gros. De nombreux changements internes amélioraient les performances et la maintenabilité. Les versions 1.0.x de Postgres95 étaient 30 à 50% plus rapides que POSTGRES, version 4.2, pour le test Wisconsin Benchmark. Mises à part les corrections de bogues, les principales améliorations étaient :

- Le langage PostQUEL était remplacé par SQL (exécuté côté serveur). Les requêtes imbriquées ne furent pas supportées avant PostgreSQL (voir plus loin) mais elles pouvaient être imitées dans Postgres95 avec des fonctions SQL définies par l'utilisateur. Les agrégats furent reprogrammés, l'utilisation de la clause GROUP BY ajouté.
- En plus du moniteur de programme, un nouveau programme (psql) permettait d'exécuter des requêtes SQL interactives, en utilisant GNU Readline.
- Une nouvelle bibliothèque cliente, libpgtcl, supportait les programmes écrits en Tcl. Un shell exemple, pgtclsh, fournissait de nouvelles commandes Tcl pour interfacer les applications Tcl avec le serveur Postgres95.
- L'interface pour les gros objets était révisée. Les objets de grande taille Inversion étaient le seul mécanisme pour stocker de tels objets, le système de fichiers Inversion étant supprimé.
- Le système de règles de niveau instance était supprimé. Les règles étaient toujours disponibles en tant que règles de réécriture.
- Un bref tutoriel présentant les possibilités SQL ainsi que celles spécifiques à Postgres95 était distribué avec le code source.
- La version GNU de make (au lieu de la version BSD) était utilisée pour la construction. Par ailleurs, Postgres95 pouvait être compilé avec GCC sans correctif (l'alignement des nombres doubles était corrigé).

5. http://traduc.postgresqlfr.org/pgsql-fr/biblio.html#STON87A
6. http://s2k-ftp.cs.berkeley.edu:8000/postgres/papers/ERL-M87-06.pdf
7. http://s2k-ftp.cs.berkeley.edu:8000/postgres/papers/ERL-M90-34.pdf
8. http://s2k-ftp.cs.berkeley.edu:8000/postgres/papers/ERL-M89-82.pdf
9. http://s2k-ftp.cs.berkeley.edu:8000/postgres/papers/ERL-M90-36.pdf
10. http://www.informix.com/
11. http://www.ibm.com/
12. http://meteora.ucsd.edu/s2k/s2k_home.html

3.2.3. PostgreSQL

En 1996, il devint évident que le nom « Postgres95 » vieillissait mal. Nous avons choisi un nouveau nom, PostgreSQL, pour mettre en avant la relation entre le POSTGRES original et les capacités SQL des versions plus récentes. En même temps, nous avons positionné le numéro de version à 6.0, pour reprendre la numérotation originale du projet POSTGRES de Berkeley.

Durant le développement de Postgres95, un effort particulier avait été fourni pour identifier et comprendre les problèmes existants dans le code. Avec PostgreSQL, la priorité put être mise sur l'augmentation des caractéristiques et des possibilités, même si le travail a continué dans tous les domaines.

3.3. PostgreSQL 8.0 est là !

Ce texte est l'article de presse annonçant la sortie de PostgreSQL 8.0.

NewYork : le 19 Janvier 2005 - Le groupe de Développement Global de PostgreSQL vient de produire la version 8.0 du système de gestion de bases de données PostgreSQL, confortant sa place de système de bases de données open source le plus sophistiqué du monde. Cette version, en proposant des fonctionnalités qui n'étaient jusqu'alors présentes que dans les systèmes de bases de données propriétaires les plus coûteux, devrait largement favoriser l'adoption de PostgreSQL par les utilisateurs et les éditeurs de solutions informatiques.

En plus des améliorations significatives sur l'extensibilité, les fonctionnalités et la performance, PostgreSQL 8.0 fait la preuve de l'efficacité sans égal du développement open source. Plus d'une douzaine de compagnies, incluant Red Hat, Fujitsu, Afilias, Software Research Associates Inc., 2nd Quadrant et Command Prompt Inc., ainsi que des centaines de développeurs individuels ont contribué à l'ajout des fonctionnalités majeures de la 8.0, dans des proportions qui dépassent toutes les versions précédentes.

M. Takyuki Nakazawa, Directeur du groupe base de données open source s'exprime en ces termes : « Nous sommes persuadés que ces fonctionnalités professionnelles attireront beaucoup de nouveaux utilisateurs vers PostgreSQL. » . « Fujitsu est fier du soutien apporté aux contributions de PostgreSQL et de son travail avec la communauté PostgreSQL. Nous souhaitons aider PostgreSQL à devenir le SGBD dominant. »

Les nouvelles fonctionnalités comprennent :

- Support natif de Windows : PostgreSQL fonctionne maintenant de manière native avec les systèmes Windows et ne nécessite aucune couche d'émulation. Ceci améliore de façon considérable les performances du logiciel par rapport à la version précédente et constitue une alternative très séduisante aux logiciels de base de données propriétaires pour les fournisseurs indépendants de logiciels, les utilisateurs professionnels et les développeurs individuels sous Windows.
- Points de retournement : Cette fonction standard de SQL permet des retours sur des points spécifiques d'une transaction sur la base de données sans que l'ensemble de l'opération soit annulée. Ceci est un avantage pour les développeurs d'applications métiers qui ont besoin de pouvoir effectuer des transactions complexes utilisant des reprises sur erreur.
- Récupération par rapport à un point dans le temps : Cette fonctionnalité permet la restauration compléte des données à partir d'une sauvegarde automatique et continue des transactions effectuées. Ceci est une alternative longtemps attendue aux sauvegardes horaires ou journalières pour les services utilisant PostgreSQL pour des données critiques.
- Espaces de tables : Cruciaux pour les administrateurs de systèmes de data warehouse de multiples-gigaoctets, les espaces de tables permettent le placement des grandes tables et index sur leurs propres disques ou grappes, afin d'améliorer les performances.
- Améliorations de la gestion de la mémoire et des entrées sorties : L'utilisation des disques et de la mémoire a été optimisée au travers de l'emploi de l'algorithme de Cache à Remplacement Adaptatif, d'un nouveau système d'écriture en tâche de fond et d'une nouvelle fonctionnalité permettant d'effectuer des vacuum différés. Ceci résulte en une charge plus prévisible ainsi qu'une performance plus cohérente lors des pics de charge.

Javier Soltero, architecte en chef chez Hyperic LLC, a dit « PostgreSQL 8.0 nous apporte un haut niveau de simultanéité d'accès et le débit nécessaire pour notre produit de surveillance HQ. PostgreSQL 8.0 disposant du support natif de Windows, nous pouvons maintenant inclure PostgreSQL dans nos produits et bénéficier de l'extensibilité et de la performance d'ores et déjà prouvées de PostgreSQL. Sa licence nous permet de l'inclure dans notre distribution sans souci de redevances commerciales. »

En plus de toutes les fonctionnalités incluses dans cette version, PostgreSQL a été étendu par le développement accéléré de composants additifs et optionnels tout au long de l'année passée. L'outil de réplication Slony-I et l'utilitaire de pooling/brokering de connection pgPool sont d'ores et déjà utilisés pour des ensemble de serveurs haute disponibilité. Plusieurs langages de procédures stockées ont été ajoutés ou largement étendus comme PL/Java, PL/PHP et PL/Perl. Les sources de données Npgsql et PGsqlClient .NET ont été améliorées afin de fournir un support aux nouveaux utilisateurs de Windows.

Pour obtenir la liste complète et la description de toutes les nouvelles fonctionnalités de la 8.0, référez-vous à notre dossier de presse[13].

À propos de PostgreSQL : PostgreSQL est le fruit du travail collectif de centaines de développeurs, s'appuyant sur un développement entamé il y a presque vingt ans à l'Université de Californie à Berkeley. Avec son support de longue date des fonctionnalités nécessaires dans l'entreprise telles que les transactions, les procédures et fonctions stockées, les triggers, les sous-requêtes, PostgreSQL est utilisé dans les branches métiers ou les agences gouvernementales les plus exigentes. PostgreSQL est distribué sous licence BSD, permettant l'utilisation et la distribution sans rétribution pour les utilisations commerciales et non commerciales.

3.4. Fonctionnalités supplémentaires dans PostgreSQL 8.0

Ce texte provient de Neodante's blog[14].

En plus des fonctionnalités majeures décrites ci-dessus, d'autres fonctionnalités trés intéressantes pour les utilisateurs de PostgreSQL ont été soit ajoutées soit améliorées.

- Fonctions : les fonctions PostgreSQL disposent maintenant de la possibilité de placer une variable à l'intérieur d'une chaîne, permettant ainsi de limiter les erreurs dues à l'imbrication des apostrophes. Par ailleurs, grâce aux nouveaux points de sauvegarde (Savepoints), les fonctions PL/pgSQL sont capables d'une gestion limitée des exceptions grâce à l'utilisation de la clause EXCEPTION.
- Conception de base de données : l'administrateur de la base de données peut maintenant changer le type d'une colonne existante en utilisant ALTER TABLE. La déclaration de colonnes utilisant des types de données composites peut aussi être utilisée, comme il est prévu dans le standard SQL. Enfin, tous les objets de la base supportent CHANGE OWNER.
- Import/Export de données : La commande COPY permet maintenant l'utilisation du format de fichier texte très répandu CSV (valeurs séparées par des virgules). Ceci rend les imports et les exports vers d'autres logiciels plus faciles à réaliser.
- Amélioration de l'optimiseur de requêtes : le planificateur et l'exécuteur de requêtes utiliseront maintenant les index pour des types de données compatibles, réduisant grandement le recours aux transtypages de valeurs constantes. Cette version inclut également des améliorations dans l'utilisation des index, une optimisation des clauses OR, un échantillonnage amélioré pour ANALYZE, un TRUNCATE plus rapide, une meilleure génération des plans de requêtes préparées, plus un grand nombre d'améliorations mineures, trop nombreuses pour être citées.
- Journalisation : les administrateurs peuvent désormais configurer la rotation des fichiers de journalisation de PostgreSQL au travers du fichier postgresql.conf. Le fichier de journalisation de PostgreSQL est encore plus configurable que précédemment, grâce à l'inclusion de préfixes de lignes dont le format aura été défini par l'administrateur, le traçage de types de requêtes SQL particuliers et le traçage des déconnexions.
- Sauvegarde : l'utilitaire de sauvegarde portable, pg_dump, a été en grande partie réécrit. Cette version ajoute de nouvelles possibilités comme les heures de début et de fin, ainsi que l'élimination des problèmes résiduels concernant la portabilité et la dépendance entre les fichiers de sauvegardes.

Bien sûr, d'autres changements et améliorations sont inclus dans cette version. Référez-vous à la page Notes de version[15] pour une liste complète.

13. http://www.postgresql.org/about/press
14. http://blogs.developpeur.org/neodante/archive/2005/01/22/4058.aspx
15. http://www.postgresql.org/docs/8.0/static/release.htmll

3.5. D'autres informations

On pourra aller auusi consulter la rubrique PostgreSQL[16] sur Wikipedia[17].

Il existe un site en français[18] consacré entièrement à PostgreSQL.

16. http://fr.wikipedia.org/wiki/PostgreSQL
17. http://fr.wikipedia.org/
18. http://www.postgresqlfr.org/

Chapitre 4. Une base *jouet* pour découvrir SQL

Vous trouverez les réponses des exercices au Chapitre 14.

4.1. Les tables

Tableau 4-1. La table UN

a	b	c
x	m	2
x	n	1
y	m	4
z	p	1

Tableau 4-2. La table DEUX

d	e
x	8
y	4
x	1

4.2. Quelques requêtes

Calculer *à la main* le résultat des requêtes suivantes.

1. `SELECT * FROM un ;`
2. `SELECT a FROM un ;`
3. `SELECT a FROM un WHERE c=1 ;`
4. `SELECT a FROM un WHERE c=1 OR c=2 ;`
5. `SELECT DISTINCT a FROM un WHERE c=1 OR c=2 ;`
6. `SELECT a FROM un ORDER BY b ;`
7. `SELECT a,e FROM un,deux ;`
8. `SELECT a,e FROM un,deux WHERE c=e ;`

Chapitre 5. La base EXEMPLE

5.1. La base EXEMPLE

Elle est constituée de deux tables : EMP et DEPT.

Cette base représente d'une manière simpliste (et remarquablement incomplète) le personnel d'une entreprise. Elle n'est pas destinée à être réaliste, mais seulement à être simple à comprendre pour une première approche de SQL. On verra même plus tard que certaines informations de la base font référence à des tables que nous ne connaissons pas.

La table EMP contient les informations sur les employés.

La table DEPT recense les différents départements de l'entreprise.

5.2. La table EMP

Tableau 5-1. La table EMP

noemp	nom	prenom	embauche	no-supr	titre	nodept	salaire	tx_commission
1	Patamob	Adhémar	03/26/2000		Président	50	50000.00	
2	Zeublouze	Agathe	04/15/2000	1	Dir. Distrib	41	35000.00	
3	Kuzbidon	Alex	05/05/2000	1	Dir. Vente	31	34000.00	
4	Locale	Anasthasie	05/25/2000	1	Dir. Finance	10	36000.00	
5	Teutmaronne	Armand	06/14/2000	1	Dir. Administr	50	36000.00	
6	Zoudanlkou	Debbie	07/04/2000	2	Chef Entrepôt	41	25000.00	
7	Rivenbusse	Elsa	07/24/2000	2	Chef Entrepôt	42	24000.00	
8	Ardelpic	Helmut	08/13/2000	2	Chef Entrepôt	43	23000.00	
9	Peursconla	Humphrey	09/02/2000	2	Chef Entrepôt	44	22000.00	
10	Vrante	Héléna	09/22/2000	2	Chef Entrepôt	45	21000.00	
11	Enfaillite	Mélusine	10/12/2000	3	Représentant	31	25000.00	10.00
12	Eurktumeme	Odile	11/01/2000	3	Représentant	32	26000.00	12.50
13	Hotdeugou	Olaf	11/21/2000	3	Représentant	33	27000.00	10.00
14	Odlavieille	Pacôme	12/11/2000	3	Représentant	34	25500.00	15.00
15	Amartakaldire	Quentin	12/31/2000	3	Représentant	35	23000.00	17.50
16	Traibien	Samira	12/31/2000	6	Secrétaire	41	15000.00	
17	Fonfec	Sophie	01/10/2001	6	Secrétaire	41	14000.00	
18	Fairant	Teddy	01/20/2001	7	Secrétaire	42	13000.00	
19	Blaireur	Terry	02/09/2001	7	Secrétaire	42	13000.00	
20	Ajerre	Tex	02/09/2001	8	Secrétaire	43	13000.00	
21	Chmonfisse	Thierry	02/19/2001	8	Secrétaire	43	12000.00	
22	Phototetedemort	Thomas	02/19/2001	9	Secrétaire	44	22500.00	
23	Kaécouté	Xavier	03/01/2001	9	Secrétaire	34	11500.00	
24	Adrouille-Toultan	Yves	03/11/2001	10	Secrétaire	45	11000.00	
25	Anchier	Yvon	03/21/2001	10	Secrétaire	45	10000.00	

Détails sur les différents champs :

`noemp`

L'identifiant de la personne. Chaque employé a un `noemp`, tous les `noemp` sont différents. C'est une clef primaire (*primary key*).

`nom`

Le nom de la personnne.

`prenom`

Le prénom de la personnne.

`embauche`

Sa date d'embauche.

`nosupr`

Le `noemp` de son supérieur hiérarchique.

Par exemple *Sophie Fonfec* (dont le `noemp` est 17) a 6 pour `nosupr`, ce qui signifie que le `noemp` de son supérieur hiérarchique est 6, c'est-à-dire que son supérieur hiérarchique est *Debbie Zoudanlkou* (dont le propre supérieur hiérarchique est *Agathe Zeublouze*, etc.).

`titre`

La fonction de l'employé (directeur, secrétaire, représentant, etc.).

`nodept`

Le numéro du département (dans l'entreprise) dont dépend l'employé. Ce numéro fait référence à la clef primaire de la table `DEPT` (voir Section 5.3). C'est une clef étrangère (*foreign key*).

`salaire`

Salaire mensuel de l'employé.

`tx_commission`

Taux de commission, exprimé en pourcentage.

5.3. La table `DEPT`

Tableau 5-2. La table `DEPT`

nodept	nom	noregion
10	Finance	1
20	Atelier	2
30	Atelier	3
31	Vente	1
32	Vente	2
33	Vente	3
34	Vente	4
35	Vente	5
41	Distribution	1
42	Distribution	2
43	Distribution	3
44	Distribution	4
45	Distribution	5
50	Administration	1

Détails sur les différents champs :

`nodept`

L'identifiant du département. Chaque département a un `nodept`, tous les `nodept` sont différents. C'est une clef primaire (*primary key*).

`nom`

Le nom du département. On peut remarquer que des départements différents peuvent avoir le même nom.

`noregion`

Référence à une région. Ce numéro fait référence à la clef primaire d'une table qui ne sera pas décrite ici[1]. C'est une clef étrangère (*foreign key*).

1. On rappelle ce qui est dit en tête de ce chapitre : *Elle n'est pas destinée à être réaliste, mais seulement à être simple à comprendre pour une première approche de SQL. On verra même plus tard que certaines informations de la base font référence à des tables que nous ne connaissons pas.* (Section 5.1)

Chapitre 6. La commande SELECT, syntaxe de base

Vous trouverez les réponses des exercices au Chapitre 15.

6.1. SELECT

La commande de recherche est le verbe SELECT .

L'étude de la commande SELECT et des différentes clauses et fonctions est faite, pour commencer, en considérant une seule table.

Par la suite, l'utilisation de plusieurs tables dans le même SELECT, sera prise en compte.

La syntaxe de base est :

```
  SELECT colonnes d'une ou plusieurs tables séparées par « , »
    FROM tables séparées par « , »
   WHERE conditions logiques séparées par « AND » ou « OR »
ORDER BY colonnes séparées par « , »
```

Le caractère « * » permet de demander l'affichage de toutes les colonnes d'une table.

1. Afficher toutes les informations concernant les employés.
2. Afficher toutes les informations concernant les départements.

A la place de « * » on peut donner la liste des colonnes souhaitées, dans l'ordre souhaité, en écrivant la requête sur une ou plusieurs lignes. L'écriture sur plusieurs lignes est conseillée en vue de rendre plus simple la relecture et la modification de la requête. Il s'agit de la projection vue Section 2.7.

3. Afficher le nom, la date d'embauche, le numéro du supérieur, le numéro de département et le salaire de tous les employés.

Le titre des colonnes obtenues par un SELECT sont les noms des champs.

On peut cependant changer ces titres en utilisant AS. La commande

```
SELECT nom AS Employe FROM emp ;
```

produira le même effet que

```
SELECT nom FROM emp ;
```

à la différence que la colonne ne sera pas intitulé nom mais Employe.

Si vous voulez utiliser un titre de colonne qui contient autre chose que des lettres non accentuées (des espaces, des ponctuations, des accents, etc.) il faut l'entourer par des guillemets (« " »). On écrira ainsi :

```
SELECT nom AS "Nom de l'employé" FROM emp ;
```

6.2. Suppression des doublons

Il peut être utile de supprimer les doublons, d'où utilisation de la clause DISTINCT. Elle a pour effet de n'afficher qu'une seule fois les lignes qui seraient semblables à l'affichage.

On l'utilise sous la forme SELECT DISTINCT ....

4. Afficher le titre de tous les employés.

5. Afficher les différentes valeurs des titres des employés.

6.3. Restriction

Les clauses de restriction s'écrivent derrière `WHERE`. Elle permettent de sélectionner les lignes à afficher. Il s'agit de la sélectiontelle que vue Section 2.7.

6. Afficher le nom, le numéro d'employé et le numéro du département des employés dont le titre est « `Secrétaire` ».
7. Afficher le nom et le numéro de département dont le numéro de departement est supérieur à `40`.

6.4. Restriction en comparant les colonnes entre elles

La restriction peut mettre en jeu la comparaison de deux ou plusieurs colonnes entre elles. Il suffit de les appeler par leurs noms.

8. Afficher le nom et le prénom des employés dont le nom est alphabétiquement antérieur au prénom.
9. Afficher le nom, le salaire et le numéro du département des employés dont le titre est « `Représentant` », le numéro de département est `35` et le salaire est supérieur à `20000`.
10. Afficher le nom, le titre et le salaire des employés dont le titre est « `Représentant` » ou dont le titre est « `Président` ».
11. Afficher le nom, le titre, le numéro de département, le salaire des employés du département `34`, dont le titre est « `Représentant` » ou « `Secrétaire` ».
12. Afficher le nom, le titre, le numéro de département, le salaire des employés dont le titre est Représentant, ou dont le titre est Secrétaire dans le département numéro `34`.
13. Afficher le nom, et le salaire des employés dont le salaire est compris entre `20000` et `30000`.
14. Afficher le nom, le titre, le numéro de département des employés dont le titre appartient à la liste : « `Représentant`, `Secrétaire` ».

6.5. Négation, recherche approchée

Les opérateurs arithmétiques sont niés avec « `!` » : par exemple « non égal » s'écrit « `!=` » (ou « `<>` »).

Les autres opérateurs sont niés par « `NOT` » : par exemple « pas dans » s'écrit « `NOT IN` ».

Les caractères *jokers* sont « `%` » pour une chaîne et « _ » pour un caractère. On les utilise avec l'opérateur `LIKE`.

15. Afficher le nom des employés commençant par la lettre « `H` ».
16. Afficher le nom des employés se terminant par la lettre « `n` ».
17. Afficher le nom des employés contenant la lettre « `u` » en 3ème position.
18. Afficher le salaire et le nom des employés du service `41` classés par salaire croissant.
19. Afficher le salaire et le nom des employés du service `41` classés par salaire décroissant.
20. Afficher le titre, le salaire et le nom des employés classés par titre croissant et par salaire décroissant.

6.6. Valeurs non renseignées

En SQL il existe une valeur correspondant à la valeur *vide*. C'est l'équivalent de la valeur Null de certains langages de programmation (comme java ou python). C'est la valeur qu'ont les champs non renseignés. Elle s'appelle également NULL. Cette valeur n'est pas prise en compte dans les classements.

21. Afficher le taux de commission, le salaire et le nom des employés classés par taux de commission croissante.

Si on veut utiliser les valeurs non renseignées dans une restriction ce sera avec les clauses IS NULL ou IS NOT NULL.

22. Afficher le nom, le salaire, le taux de commission et le titre des employés dont le taux de commission n'est pas renseigné.
23. Afficher le nom, le salaire, le taux de commission et le titre des employés dont le taux de commission est renseigné.
24. Afficher le nom, le salaire, le taux de commission, le titre des employés dont le taux de commission est inférieur à 15.

6.7. Expressions arithmétiques

Des expressions arithmétiques[1] peuvent être utilisées après : SELECT, WHERE, et ORDER BY.

25. Afficher le nom, le salaire, le taux de commission et la commission d employés dont le taux de commission n'est pas nul.
26. Afficher le nom, le salaire, le taux de commission, la commission des employés dont le taux de commission n'est pas nul, classé par taux de commission croissant.

6.8. Éviter les valeurs NULL

Le remplacement *automatique* des valeurs non renseignées (NULL) se fait par l'utilisation de la fonction COALESCE.

```
COALESCE( arg1, arg2, ...)
```

où :

- arg1 = colonne qui peut être non renseignée
- arg2 = valeur ou nom d'une autre colonne de substitution
- ...

Le résultat renvoyé est la première valeur non NULL.

27. Afficher le nom, la rémunération totale des employés.
28. Afficher le nom, la rémunération totale, en tenant compte des taux de commission non renseignés des employés.

1. C'est-à-dire des calculs sur les nombres.

6.9. Arrondis

La fonction `ROUND(arg1, arg2)` permet de faire l'arrondi mathématique de `arg1` avec `arg2` décimales.

La fonction `TRUNC(arg1, arg2)` permet de tronquer `arg1` avec `arg2` décimales.

29. Afficher le nom, le salaire et le salaire horaire arrondi à deux décimales des employés du département `35`. Renommer les colonnes.

6.10. Concaténation

L'opérateur de concaténation de chaînes de caractères est « `||` » (2 caractères « pipe »).

30. Afficher le nom et le prénom (concaténés) des employés. Renommer les colonnes.

6.11. Opérateur conditionnel

La fonction de codification du `SQL` standard n'existe pas en `PostgreSQL` :

```
DECODE (arg1, arg20,arg21, arg30,arg31, arg40)
```

Mais on dispose de la structure `CASE` qui donne le même résultat (et avec plus de possibilités) : 1

```
CASE WHEN cond1 THEN arg1 WHEN cond2 THEN arg2 ... ELSE arg3 END
```

- si la condition `cond1` est vérifiée, alors le résultat est `arg1`,
- sinon, si la condition `cond2` est vérifiée, alors le résultat est `arg2`,
- ...
- sinon le résultat est `arg3`.

31. Afficher le nom, le numéro de département, le numéro de département décodé des départements.

    Ou, pour simplifier : pour le département de numéro 10, afficher « Vu », pour le département de numéro 50, afficher « Pas vu », pour les autres départements, afficher « En cours »....

32. Afficher le prénom si le département est `41` sinon le nom des employés, le numéro de département, et le salaire des employés classé par numéro de département.

6.12. Chaînes de caractères

La fonction extraction de chaîne de caractères `SUBSTR(arg1, arg2, arg3)` permet d'extraire de `arg1` à partir de la position `arg2`, les `arg3` caractères. Si `arg3` n'est pas précisé, on obtient toute la fin de la chaîne de caractères.

La fonction de retour du rang d'une chaîne dans une autre chaîne, `STRPOS (arg1, arg2)` permet de retourner le rang de la chaîne `arg2` dans la chaîne `arg1`.

Les fonctions `UPPER(arg1)` et `LOWER(arg1)` permettent respectivement de forcer à la majuscule ou à la minuscule.

La fonction `LENGTH(arg1)` permet d'obtenir le nombre de caractères d'une chaîne de caractères.

33. Afficher les 5 premières lettres du nom des employés.
34. Afficher le nom et le rang de la lettre « `r` » à partir de la 3ème lettre dans le nom des employés.
35. Afficher le nom, le nom en majuscule et le nom en minuscule de l'employé dont le nom est `Vrante`.
36. Afficher le nom et le nombre de caractères du nom des employés.

6.13. Opérations ensemblistes

Utilisation des opérateurs ensemblistes : `UNION`, `INTERSECT`, `EXCEPT`.

`UNION` seul prend les données distinctes uniquement.

Si on veut toutes les données : `UNION ALL`.

37. Afficher le nom, le type de rémunération (salaire ou commission) et le montant de ce salaire ou de cette commission, des employés dont le `TX_COMMISSION` n'est pas vide.
38. Afficher les numéros de département communs à la table `EMP` et à la table `DEPT`.
39. Afficher les numéros de département auxquels aucun employé n'est affecté.

Chapitre 7. Les jointures et les sous-requêtes

Vous trouverez les réponses des exercices au Chapitre 16.

7.1. Jointures

Dans la table `EMP` la colonne `nodept` contient des valeurs qui se trouvent dans la colonne `nodept` de la table `DEPT` et inversement.

Logiquement chaque ligne de `EMP` *correspond* à une ligne de `DEPT` et, chaque ligne `DEPT` *correspond* à une ou plusieurs lignes de `EMP`.

Cette correspondance se fait par l'opérateur égalité (« = ») entre les valeurs des deux colonnes des deux tables : c'est une équi-jointure.

Après `FROM` les deux tables sont citées, et le critère d'équi-jointure sert de restriction. Sans cette restriction c'est un produit cartésien de `EMP` et de `DEPT` qui serait obtenu.

1. Rechercher le prénom des employés et le numéro de la région de leur département.

Derrière `SELECT` il peut y avoir nécessité de préfixer la colonne par le nom de la table. En effet si la colonne apparaît dans plusieurs tables il y a ambiguïté.

Pour faciliter l'écriture des requêtes, les tables citées derrière `FROM` peuvent être renommées temporairement, et l'alias peut être utilisé pour préfixer les colonnes.

2. Rechercher le numéro du département, le nom du département, le nom des employés classés par numéro de département (renommer les tables utilisées).

Si le critère de jointure est omis, le résultat est un produit cartésien inutilisable le plus souvent.

3. Rechercher le nom de l'employé `Amartakaldire` et le nom de son département, ne pas utiliser de critère de jointure (volontairement).

7.2. Auto-jointures

La possibilité de renommer temporairement une table dans une requête permet de faire la jointure d'une table sur elle-même, c'est à dire l'auto-jointure.

4. Rechercher le nom et le salaire des employés qui gagnent plus que leur patron, et le nom et le salaire de leur patron.
5. Rechercher le nom et le salaire des employés qui gagnent plus que `Amartakaldire`, le nom et le salaire de `Amartakaldire`.

7.3. Sous-requêtes

Le résultat d'une requête peut servir dans une clause de restriction d'une autre requête, on parlera alors de sous-requête imbriquée.

La recherche suivante peut se faire de deux manières.

6. Rechercher le nom et le titre des employés qui ont le même titre que `Amartakaldire`.

Dans ce qui précède, la sous-requête retourne une seule valeur, l'opérateur égalité peut être utilisé. Si ce n'est pas le cas il faut utiliser les clauses `ANY` ou `ALL`.

7. Rechercher le nom, le salaire et le numéro de département des employés qui gagnent plus qu'un seul employé du département `31`, classés par numéro de département et salaire.
8. Rechercher le nom, le salaire et le numéro de département des employés qui gagnent plus que tous les employés du service `31`, classés par numéro de département et salaire.

En fait : « `IN` » est équivalent à « `= ANY` », tandis que « `NOT IN` » est équivalent à « `!= ALL` ».

9. Rechercher le nom et le titre des employés du service `31` qui ont un titre que l'on trouve dans le service `32`.
10. Rechercher le nom et le titre des employés du service `31` qui ont un titre l'on ne trouve pas dans le service `32`.
11. Rechercher le nom, le titre et le salaire des employés qui ont le même titre et le même salaire que `Fairant`.

Renommer une table et utiliser une sous-requête permet de synchroniser une sous-requête avec la requête principale.

12. Rechercher le numéro de département, le nom et le salaire des employés qui gagnent plus que la moyenne de leur département, classés par département.

L'opérateur `EXISTS` permet de retourner des lignes si la sous-requête en retourne.

13. Rechercher le numéro d'employé, le nom, le prénom des employés pour lesquel il existe au moins un `Représentant` dans leur département.
14. Rechercher le numéro de département, le nom du département dans lesquels il n'y a personne et montrer que le département ne contient pas d'employé.

7.4. `LEFT JOIN`, `RIGHT JOIN`

Dans la table `DEPT` il y a des lignes avec un numéro de département qui ne *correspondent* à aucune ligne de `EMP`.

Cette ligne est à l'extérieur de la jointure entre les deux tables.

Si on souhaite, malgré tout, obtenir dans le résultat de la jointure ces lignes *extra* on utilise un `LEFT JOIN`.

Pour bien comprendre le sens et la syntaxe de `LEFT JOIN` il peut être utile de revenir sur la syntaxe des requêtes. Il faut savoir que la requête

```
SELECT ... FROM a,b,c WHERE a.x=b.y AND b.z=c.t ...
```

peut s'écrire également

```
SELECT ... FROM a JOIN b ON a.x=b.y JOIN c ON b.z=c.t ...
```

Il s'agit de la syntaxe originale des jointures en `SQL`.

Pour écrire un `LEFT JOIN`, la syntaxe est la même, en remplaçant bien entendu `JOIN` par `LEFT JOIN`.

Qu'est-ce que cela va changer ? On se souvient que pour une jointure *normale* on ne prend que les enregistrements de chaque table qu'on peut relier par la condition de jointure. Avec un `LEFT JOIN` on prendra en plus les enregistrements de la table écrite à gauche (car `LEFT`) de l'expression `LEFT JOIN` et qui ne sont reliés à aucun enregistrement de celle de droite.

Il existe également `RIGHT JOIN` qui fonctionne de manière tout à fait symétrique.

15. Rechercher le numéro de département, le nom du département, le nom des employés en tenant compte des numéro de département dans lesquels il y a personne classé par numéro de département.

Chapitre 8. Créer votre propre base

8.1. Créer la base

Utiliser d'abord la commande :

```
CREATE DATABASE base ;
```

où `base` est un nom à votre choix (qui ne doit pas déjà exister).

Connectez vous ensuite à votre base

```
\c base
```

Puis tapez (ou recopiez) les commandes suivantes.

8.2. Créer les tables

8.2.1. DEPT

```
CREATE TABLE dept
            (nodept NUMERIC(2) NOT NULL
                    CONSTRAINT dept_nodept_pk PRIMARY KEY,
             nom VARCHAR(25) ,
             noregion NUMERIC(1) NOT NULL
            )
;
```

8.2.2. EMP

```
CREATE TABLE emp
            (noemp NUMERIC(7) NOT NULL
                   CONSTRAINT emp_noemp_pk PRIMARY KEY,
             nom VARCHAR(25) ,
             prenom VARCHAR(25) ,
             embauche DATE,
             nosupr NUMERIC(7) ,
             titre  VARCHAR(25) ,
             nodept NUMERIC(2) NOT NULL ,
             salaire NUMERIC(11, 2) ,
             tx_commission NUMERIC(4, 2)
                           CONSTRAINT emp_tx_commission_ck
                                      CHECK (tx_commission BETWEEN 10 AND 20) ,
             CONSTRAINT emp_nodept_fk
                        FOREIGN KEY (nodept) REFERENCES dept(nodept)
            )
;
```

8.3. Remplir les tables

8.3.1. DEPT

```
INSERT INTO dept VALUES (10 , 'Finance' , 1 ) ;
INSERT INTO dept VALUES (20 , 'Atelier' , 2 ) ;
INSERT INTO dept VALUES (30 , 'Atelier' , 3 ) ;
INSERT INTO dept VALUES (31 , 'Vente' , 1 ) ;
INSERT INTO dept VALUES (32 , 'Vente' , 2 ) ;
INSERT INTO dept VALUES (33 , 'Vente' , 3 ) ;
INSERT INTO dept VALUES (34 , 'Vente' , 4 ) ;
```

```
INSERT INTO dept VALUES (35 , 'Vente' , 5 ) ;
INSERT INTO dept VALUES (41 , 'Distribution' , 1 ) ;
INSERT INTO dept VALUES (42 , 'Distribution' , 2 ) ;
INSERT INTO dept VALUES (43 , 'Distribution' , 3 ) ;
INSERT INTO dept VALUES (44 , 'Distribution' , 4 ) ;
INSERT INTO dept VALUES (45 , 'Distribution' , 5 ) ;
INSERT INTO dept VALUES (50 , 'Administration' , 1 ) ;
```

8.3.2. EMP

```
INSERT INTO emp VALUES (1 , 'Patamob' , 'Adhémar' , '03/26/2000' ,
                        NULL , 'Président' , 50 , 50000 , NULL) ;
INSERT INTO emp VALUES (2 , 'Zeublouze' , 'Agathe' , '04/15/2000' ,
                        1 , 'Dir. Distrib' , 41 , 35000 , NULL) ;
INSERT INTO emp VALUES (3 , 'Kuzbidon' , 'Alex' , '05/05/2000' ,
                        1 , 'Dir. Vente' , 31 , 34000 , NULL) ;
INSERT INTO emp VALUES (4 , 'Locale' , 'Anasthasie' , '05/25/2000' ,
                        1 , 'Dir. Finance' , 10 , 36000 , NULL) ;
INSERT INTO emp VALUES (5 , 'Teutmaronne' , 'Armand' , '06/14/2000' ,
                        1 , 'Dir. Administr' , 50 , 36000 , NULL) ;
INSERT INTO emp VALUES (6 , 'Zoudanlkou' , 'Debbie' , '07/04/2000' ,
                        2 , 'Chef Entrepôt' , 41 , 25000 , NULL) ;
INSERT INTO emp VALUES (7 , 'Rivenbusse' , 'Elsa' , '07/24/2000' ,
                        2 , 'Chef Entrepôt' , 42 , 24000 , NULL) ;
INSERT INTO emp VALUES (8 , 'Ardelpic' , 'Helmut' , '08/13/2000' ,
                        2 , 'Chef Entrepôt' , 43 , 23000 , NULL) ;
INSERT INTO emp VALUES (9 , 'Peursconla' , 'Humphrey' , '09/02/2000' ,
                        2 , 'Chef Entrepôt' , 44 , 22000 , NULL) ;
INSERT INTO emp VALUES (10 , 'Vrante' , 'Héléna' , '09/22/2000' ,
                        2 , 'Chef Entrepôt' , 45 , 21000 , NULL) ;
INSERT INTO emp VALUES (11 , 'Enfaillite' , 'Mélusine' , '10/12/2000' ,
                        3 , 'Représentant' , 31 , 25000 , 10) ;
INSERT INTO emp VALUES (12 , 'Eurktumeme' , 'Odile' , '11/01/2000' ,
                        3 , 'Représentant' , 32 , 26000 , 12.5) ;
INSERT INTO emp VALUES (13 , 'Hotdeugou' , 'Olaf' , '11/21/2000' ,
                        3 , 'Représentant' , 33 , 27000 , 10) ;
INSERT INTO emp VALUES (14 , 'Odlavieille' , 'Pacôme' , '12/11/2000' ,
                        3 , 'Représentant' , 34 , 25500 , 15) ;
INSERT INTO emp VALUES (15 , 'Amartakaldire' , 'Quentin' , '12/31/2000' ,
                        3 , 'Représentant' , 35 , 23000 , 17.5) ;
INSERT INTO emp VALUES (16 , 'Traibien' , 'Samira' , '01/10/2001' ,
                        6 , 'Secrétaire' , 41 , 15000 , NULL) ;
INSERT INTO emp VALUES (17 , 'Fonfec' , 'Sophie' , '01/20/2001' ,
                        6 , 'Secrétaire' , 41 , 14000 , NULL) ;
INSERT INTO emp VALUES (18 , 'Fairant' , 'Teddy' , '02/09/2001' ,
                        7 , 'Secrétaire' , 42 , 13000 , NULL) ;
INSERT INTO emp VALUES (19 , 'Blaireur' , 'Terry' , '02/09/2001' ,
                        7 , 'Secrétaire' , 42 , 13000 , NULL) ;
INSERT INTO emp VALUES (20 , 'Ajerre' , 'Tex' , '02/19/2001' ,
                        8 , 'Secrétaire' , 43 , 13000 , NULL) ;
INSERT INTO emp VALUES (21 , 'Chmonfisse' , 'Thierry' , '02/19/2001' ,
                        8 , 'Secrétaire' , 43 , 12000 , NULL) ;
INSERT INTO emp VALUES (22 , 'Phototetedemort' , 'Thomas' , '03/01/2001' ,
                        9 , 'Secrétaire' , 44 , 22500 , NULL) ;
INSERT INTO emp VALUES (23 , 'Kaécouté' , 'Xavier' , '03/11/2001' ,
                        9 , 'Secrétaire' , 34 , 11500 , NULL) ;
INSERT INTO emp VALUES (24 , 'Adrouille-Toultan' , 'Yves' , '03/21/2001' ,
                        10 , 'Secrétaire' , 45 , 11000 , NULL) ;
INSERT INTO emp VALUES (25 , 'Anchier' , 'Yvon' , '12/31/2000' ,
                        10 , 'Secrétaire' , 45 , 10000 , NULL) ;
```

8.4. Téléchargement

Vous trouverez aussi les fichiers SQL à télécharger :

- `creer_dept.sql`[1] pour créer la table DEPT.
- `creer_emp.sql`[2] pour créer la table EMP.
- `remplir_dept.sql`[3] pour remplir la table DEPT.
- `remplir_emp.sql`[4] pour remplir la table EMP.

1. http://www.grappa.univ-lille3.fr/polys/sql/exemples/creer_dept.sql
2. http://www.grappa.univ-lille3.fr/polys/sql/exemples/creer_emp.sql
3. http://www.grappa.univ-lille3.fr/polys/sql/exemples/remplir_dept.sql
4. http://www.grappa.univ-lille3.fr/polys/sql/exemples/remplir_emp.sql

Chapitre 9. Modification de base, transactions, tables et vues

Vous trouverez les réponses des exercices au Chapitre 17.

L'utilisateur qui a créé les tables a tous les droits sur ces tables. Si rien n'est précisé aucun autre utilisateur ne peut faire la moindre modification sur les tables (contenu ou structure). Ce qui fait que vous ne pourrez tester les commandes de ce chapitre que sur une base de données que vous aurez créée personnellement (voir Chapitre 8).

On étudiera plus tard (Chapitre 10) la possibilité de donner (ou enlever) aux autres utilisateurs les droits de modifier la base de données.

9.1. Transactions

9.1.1. Pourquoi utiliser les transactions ?

Le principe de base des transactions est de grouper un ensemble de commandes de façon à être sûr que TOUTES les commandes seront exécutées, ou sinon qu'AUCUNE ne sera exécutée.

Dans quel but ? Pour pouvoir résoudre un certain de problèmes différents. Par exemple :

- Imaginez une transaction bancaire : la banque doit débiter un compte pour en créditer un autre. Facile à réaliser avec deux commandes `UPDATE`. Mais que faire si une fois la première commande exécutée, on se rend compte que la deuxième ne peut pas l'être ? Il faut annuler la première. Regrouper les deux commandes dans une transaction permet d'automatiser ce comportement : si tout se passe bien les deux se feront, si un quelconque problème survient aucune ne se fera.
- D'une manière plus générale, une suite de commandes quelconques qui entraîne des modifications de la base ne doit surtout pas être arrêtée en plein milieu, ce qui risquerait de laisser la base dans un état inconsistant. Une transaction transforme cette suite de commandes en un ensemble insécable (on dira aussi *atomique*) : ou TOUT se fait, ou RIEN ne sera fait.
- Les transactions permettent aussi de gérer les accès concurrents dans une base en réseau : si deux (ou plus) utilisateurs utilisent en même temps la même base pour exécuter chacun une série de commandes du style :
  - `commande1` : recherche d'une information dans une table (par exemple : calcul d'un stock disponible) ;
  - `commande2` : modification de la table en fonction de l'information reçue (par exemple retrait d'un certain nombre d'articles dans la limite du stock diponible).

  Dans ce cas rien ne permet de supposer que tout se passera bien. Il est tout à fait possible que la séquence de commandes de déroule de la façon suivante : `commande1` pour le premier client, puis `commande1` pour le deuxième client, puis `commande2` pour le premier client, puis `commande2` pour le deuxième client.

  Dans ce cas la `commande2` du deuxième client a toutes les chances d'entraîner des erreurs car elle est basée sur un résultat calculé avant l'exécution de la `commande2` du premier client.

  L'utilisation de transaction transforme chaque suite (`commande1`, `commande2`) en un bloc non interruptible (*atomique*) qui assure que l'information du deuxième ne sera fournie que quand le premier aura modifié la table.
- Enfin il peut être utile de se ménager une *porte de sortie* quand on exécute une commande modifiant la table : on peut se tromper, changer d'avis, vouloir seulement faire un essai, etc. Il est toujours utile de disposer de l'équivalent du *undo* de la plupart des logiciels.

  La seule solution est alors d'utiliser une transaction : une fois les commandes exécutées, on peut décider de tout abandonner (comme pour un *undo*) ou au contraire d'accepter tout.

Vous pourrez trouver d'autre explications sur l'utilité des transactions dans les pages suivantes :

- Comment protéger des ensembles d'opérations par des transactions ?[1]
- À quoi servent les transactions ?[2]

1. http://hcesbronlavau.developpez.com/Transactions/
2. http://sqlpro.developpez.com/cours/sqlaz/techniques/#L1

- JDBC : Gestion des transactions[3]

9.1.2. Comment utiliser les transactions ?

Une *transaction* commence par « `BEGIN ;` » :

- Si elle se termine par « `ROLLBACK ;` », aucune des modifications faites ne sera prise ne compte.
- Si elle se termine par « `COMMIT ;` », toutes les modifications sont prises en compte.
- Ne pas taper « `BEGIN ;` » avant de commencer vos modifications revient à taper chacune de vos commandes en mode *autocommit* : chaque instruction isolée est une transaction en elle-même, une fois tapée elle est exécutée comme si elle était précédée d'un « `BEGIN ;` » et suivie d'un « `COMMIT ;` », et elle ne peut plus être défaite.

Toutes les commandes qui vont suivre dans ce chapitre vont modifier votre base de données. Nul n'est à l'abri d'une erreur, il vous est donc fortement conseillé d'inclure vos commandes dans des transactions, pour vous permettre d'annuler éventuellement toute fausse manœuvre.

Dans notre cas il ne se produira rien de castrophique : cette base n'a pas d'importance vitale, et si vous y faites des erreurs, il vous reste toujours la possibilité de la recréer complétement (Chapitre 8).

Dans la *vraie* vie c'est rarement le cas. C'est donc une bonne habitude à prendre...

9.2. Modifier le contenu

9.2.1. Créer une table

La commande `CREATE TABLE` permet de créer une table. On la fait suivre d'une liste (entre parenthèses) des champs accompagnés de leur type.

Exemple 9-1. `CREATE TABLE`

Créons la table `AFFAIRE` qui gardera les informations sur les affaires en cours :

- `NOAFF`, numérique sur 3 chiffres ; il s'agit de la clef primaire (bien que ce ne soit pas précisé dans sa déclaration) ;
- `NOM`, texte sur 10 caractères ; texte décrivant l'affaire ;
- `BUDGET`, numérique sur 8 chiffres dont 2 après la virgule ; budget consacrée à l'affaire.

```
CREATE TABLE AFFAIRE (
             NOAFF NUMERIC(3) NOT NULL,
             NOM VARCHAR (10) ,
             BUDGET NUMERIC(8,2))
;
```

On peut également, à la place de la liste des champs, utiliser une requête (précédée de `AS`) pour remplir en même temps la table.

1. Créer une table `TRUC` copie d'une autre table (`EMP`, par exemple)

3. http://www.infini-fr.com/Sciences/Informatique/Langages/Imperatifs/Java/Jdbc/transactions.html

9.2.2. Insertion de lignes

La commande INSERT permet de créer une nouvelle ligne dans une table.

Par exemple, pour insérer une nouvelle ligne dans la table EMP, essayez la suite de commandes que voici :

```
BEGIN /* début de transaction */
;
INSERT INTO EMP /* insertion d'une ligne dans la table EMP */
       VALUES (1001,'Bodinoz','Arnaud','1998-05-19',null,1, 50,15000,null)
;
SELECT * FROM EMP   /* liste, pour vérifier l'insertion */
;
ROLLBACK /* on annule la transaction */
;
SELECT * FROM EMP   /* on vérifie à nouveau : la nouvelle ligne n'est plus là */
;
BEGIN /* on recommence... */
;
INSERT INTO EMP
       VALUES (1001,'Bodinoz','Arnaud','1998-05-19',null,1, 50,15000,null)
;
SELECT * FROM EMP   /* on vérifie à nouveau */
;
COMMIT /* on confirme la transaction */
;
SELECT * FROM EMP   /* la nouvelle ligne est toujours là */
;
```

La mise à jour n'est prise en compte que si l'intégrité des données est satisfaite.

Par exemple, si on veut insérer une nouvelle ligne dans la table EMP avec le numéro d'employé non renseigné :

```
BEGIN
;
INSERT INTO EMP /* cela produit une erreur */
       VALUES (null,'Nogues', 'Annie', '11-SEP-97',null,1, 50,12000,null)
;
SELECT * FROM EMP /* l'insertion n'a pas eu lieu */
;
COMMIT
;
```

Même quand on ne souhaite pas remplir toutes les colonnes on peut insérer une nouvelle ligne dans la table EMP sans utiliser la clause NULL pour les valeurs non renseignées.

```
BEGIN
;
INSERT INTO EMP (NOEMP, NOM, EMBAUCHE, NODEPT) /
       VALUES (1002, 'Charon','22-APR-98',50)
; /* on a créé une nouvelle ligne en ne remplissant que 4 champs */
SELECT *
  FROM EMP
 WHERE NOM = 'Charon'
;
COMMIT
;
```

Créer la table BONUS vide avec la commande suivante :

```
CREATE TABLE bonus (
             nom VARCHAR(25) ,           /* nom d'un employé */
             titre  VARCHAR(25) ,        /* son titre */
             salaire NUMERIC(11, 2) ,    /* son  salaire */
             tx_commission NUMERIC(4, 2) /* son taux de commission*/
                           CONSTRAINT emp_tx_commission_ck
                                      CHECK (tx_commission BETWEEN 10 AND 20)
             /* avec une contrainte sur le taux de commission qui devra être entre 10 et 20 */
            )
;
```

L'insertion dans une table peut se faire par une sélection dans une autre table. Par exemple pour ajouter dans la table `BONUS` les lignes de `EMP` dont le titre est `Chef Entrepôt` ou dont le taux de commission est supérieur à `14` :

```
BEGIN
;
SELECT *
  FROM BONUS /* vérifier que la table est vide */
;
INSERT INTO BONUS /* insertion dans la table... */
       SELECT NOM, TITRE, SALAIRE, TX_COMMISSION
         FROM EMP                    /* ...à partir d'éléments de EMP */
        WHERE TITRE = 'Chef Entrepôt'
           OR TX_COMMISSION > 14
;
SELECT * /* vérifier que l'insertion est faite */
  FROM BONUS
;
COMMIT
;
```

2. Insérer des données dans la table `AFFAIRES` :

   - 101, 'ALPHA' ,250000 ;
   - 102, 'BETA' , 175000 ;
   - 103, 'GAMMA' ,95000.

9.2.3. Modification de lignes

La commande `UPDATE` permet de mettre à jour dans une table une ou plusieurs colonnes pour une ou plusieurs lignes.

Pour mettre à jour le salaire de l'employé `Amartakaldire`, en l'augmentant de `10%`, il faudra taper :

```
BEGIN /* ça DOIT être un réflexe avant un UPDATE */
;
SELECT *
  FROM EMP
 WHERE NOM ='Amartakaldire'
;
UPDATE EMP   /* modification */
   SET SALAIRE = SALAIRE * 1.1 ;
 WHERE NOM = 'Amartakaldire'
;
SELECT * /* vérification */
  FROM EMP
 WHERE NOM ='Amartakaldire'
;
COMMIT   /* confirmation */
;
```

3. Mettre à jour le titre (Représentant), le salaire (20000) et le taux de commission (15%) de `Kaécouté`.
4. Augmenter de `5%` les employés de la table `EMP` qui se trouvent dans la table `BONUS`.

9.2.4. Suppression de lignes

La commande `DELETE` permet de supprimer une ou plusieurs lignes d'une table. Si elle n'est pas suivi d'une restriction (`WHERE`, avec la même syntaxe que pour `SELECT`), c'est toute la table qui est effacée.

5. Supprimer de la table `BONUS` l'employé `Zoudanlkou`.
6. Supprimer l'employé `Amartakaldire` de la table `BONUS`.
7. Supprimer l'employé `Amartakaldire` de la table `EMP` et faire `ROLLBACK`.
8. Supprimer toutes les lignes de `BONUS` et faire `ROLLBACK`, recommencer mais faire `COMMIT` (et essayer `ROLLBACK` après).

9.2.5. Suppression d'une table

La commande `DROP TABLE` permet de détruire une table (lignes et structure).

Pour supprimer totalement la table `EMP` :

```
DROP TABLE EMP ;
```

Pour la recréer voir Chapitre 8.

9.2.6. Modification de la structure d'une table

La commande `ALTER TABLE` permet de modifier la structure de la table, même si elle contient des données.

Les principales façons de l'utiliser sont :

`ALTER TABLE` *`nom`* `RENAME TO` *`nouveau_nom`*
: renommer une table

`ALTER TABLE` *`nom`* `RENAME` *`colonne`* `TO` *`nouvelle_colonne`*
: renommer une colonne d'une table

`ALTER TABLE` *`nom`* `ADD` *`colonne type`*
: ajouter une colonne à une table

`ALTER TABLE` *`nom`* `DROP` *`colonn`*
: supprimer une colonne

`ALTER TABLE` *`nom`* `ALTER` *`colonne`* `TYPE` *`type`*
: changer le type d'une colonne

`ALTER TABLE` *`nom`* `OWNER TO` *`nouveau_propriétaire`*
: changer le propriétaire d'une table

9. Modifier la structure de `EMP` en ajoutant une colonne `NOAFF`.
10. Mettre à jour la table `EMP` :
    - `noaff` prend la valeur 101 quand `nodept` vaut 10 ou 50 ;
    - `noaff` prend la valeur 102 quand `nodept` vaut 31, 32, 33, 34 ou 35 ;
    - `noaff` prend la valeur 103 quand `nodept` vaut 41, 42, 43, 44 ou 45.
11. Détruire la table `AFFAIRE`.

    Détruire et re-créer la table `EMP` à l'original.

9.3. Vues

Une vue est une table virtuelle, c'est-à-dire dont les données ne sont pas stockées dans une table de la base de données, et dans laquelle il est possible de rassembler des informations provenant de plusieurs tables. On parle de « vue » car il s'agit simplement d'une représentation des données dans le but d'une exploitation visuelle. Les données présentes dans une vue sont définies grâce à une clause `SELECT`.

Une vue est une définition dynamique dans le sens suivant : si on modifie, ou si on ajoute ou si on supprime des enregistrements, chaque exécution de la vue comprendra ces modifications.

La norme `SQL` propose un ensemble important de restrictions pour la modification ou l'insertion ou la modification des données dans les vues. Les systèmes de gestion de base de données ont aussi chacun leur implantation de ce concept et chacun leurs contraintes et restrictions. En particulier, peu d'opérations sont autorisées dès qu'une vue porte sur plusieurs tables ; aucune n'est possible si la vue comporte des opérateurs d'agrégation.

En `PostgreSQL` les vues ne sont que consultables par des instructions `SELECT`. Aucune autre opération n'est possible. Par contre, c'est la notion propre à `PostgreSQL` de règles qui assure cette fonctionnalité. Cette notion s'avère plus souple et puissante que les restrictions communément appliquées aux SGBD classiques.

La commande `CREATE VIEW` permet de créer une vue relative à une ou plusieurs tables de la base. Elle est suivie de `AS` puis d'une requête qui la définit.

12. Créer une vue relative au numéro, nom et titre des employés des départements supérieurs à `40` et utiliser la vue pour interroger. Mettre à jour `EMP` et utiliser la vue pour interroger.

Dans la version actuelle de `PostgreSQL` une vue ne peut pas servir à mettre à jour une table.

13. Essayez de mettre à jour la table `EMP` en utilisant la vue `DEPT4`.
14. Créer une vue relative au nom, numéro de service des employés et au nom du service. Utiliser la vue.
15. Créer une vue relative au nom, salaire et salaire annuel des employés.

Chapitre 10. Les droits

Vous trouverez les réponses des exercices au Chapitre 18.

Une table ayant été créée, son propriétaire peut accorder (`GRANT`) ou retirer (`REVOKE`) à un autre utilisateur (ou à tous : `PUBLIC`) les droits suivants :

- `SELECT` : Accès en lecture à toutes les colonnes d'une table ou d'une vue.
- `INSERT` : Insérer des données dans toutes les colonnes d'une table.
- `UPDATE` : Mettre à jour dans toutes les colonnes d'une table.
- `DELETE` : Supprimer des enregistrements d'une table.
- `ALL` : Donner tous les privilèges.

La syntaxe en est :

```
GRANT droit ON table TO user ;
REVOKE droit ON table FROM user ;
```

À tout moment vous pouvez afficher les droits actuellement donnés par la commande « `\dp` » ou « `\z` ».

1. Accorder le droit de `SELECT` sur la table `DEPT` à un autre *user*.
2. Accorder les droits de `INSERT` et `UPDATE` sur `DEPT` à un autre *user*.
3. Accorder tous les droits sur la table `DEPT` à un autre *user*.
4. Créer une vue relative au numéro, nom, titre, numéro de département des employés.
   Puis accorder les droits de `SELECT` sur la vue à un autre *user*
5. Créer une vue relative à toutes les informations des employés du service `50`.
   Puis accorder tous les droits sur la vue à un autre *user*.
6. Retirer le droit `INSERT` à l'autre *user*, sur la table `DEPT`.

Chapitre 11. Les groupes

Vous trouverez les réponses des exercices au Chapitre 19.

11.1. Utilisation de fonctions de groupe

Par exemple : `AVG` (moyenne), `MIN` (minimum), `MAX` (maximum), `COUNT` (dénombrement)...

Ces fonctions *travaillent* au niveau de groupe de lignes et non plus au niveau des lignes.

Exemple 11-1. Moyenne

Par exemple pour rechercher la moyenne des salaires des secrétaires :

```
          SELECT AVG(SALAIRE)
            FROM EMP
           WHERE TITRE = 'Secrétaire'
;
```

Avec `SELECT` on ne peut pas *travailler* à la fois au niveau des lignes et des groupes.

Si vous recherchez le nom et la moyenne des salaires des employés (cette phrase a-t-elle d'ailleurs un sens ?), vous allez essayer :

```
          SELECT NOM, AVG(SALAIRE)
            FROM EMP
          ;
```

Ce qui ne produira qu'un message d'erreur.

Par contre avec deux `SELECT` imbriqués on peut rechercher le nom et le salaire de l'employé dont le salaire est le plus grand.

```
          SELECT NOM, SALAIRE
            FROM EMP
           WHERE SALAIRE = (SELECT MAX (SALAIRE)
                              FROM EMP
                           )
          ;
```

11.2. Les groupes

Pour exprimer le groupe sur lequel doit porter la fonction de groupe on utilise la clause `GROUP BY`.

Ces fonctions et clauses peuvent s'utiliser avec une jointure.

Toute colonne qui intervient dans l'affichage sans être utilisée dans une fonction de groupe doit être aussi incluse dans la clause `GROUP BY`.

Pour rechercher la moyenne des salaires de chaque département on écrira :

```
          SELECT NODEPT, AVG(SALAIRE)
            FROM EMP
        GROUP BY NODEPT
        ;
```

11.3. La clause `HAVING`

La clause `WHERE` permet d'écrire une restriction au niveau ligne, la clause `HAVING` permet d'écrire une restriction au niveau groupe.

Pour rechercher les titres et le nombre d'employés pour les titres représentés plus de 2 fois, on écrira :

```
  SELECT TITRE, COUNT(*)
    FROM EMP
GROUP BY TITRE
  HAVING COUNT(*) > 2
;
```

11.4. Exercices

1. Rechercher le salaire maximum et le salaire minimum parmi tous les salariés et l'écart entre les deux.
2. Rechercher le nombre de titres différents.
3. Rechercher le numéro de département, le titre et le nombre d'employés par département, titre.
4. Rechercher le nom du département, le titre et le nombre d'employés par département, titre.
5. Rechercher les titres et la moyenne des salaires par titre dont la moyenne est supérieure à la moyenne des salaires des `Représentants`.
6. Rechercher le nombre de salaires renseignés et le nombre de taux de commission renseignés.
7. Rechercher la moyenne des taux de commission renseignés et la moyenne des taux de commission en tenant compte des taux de commission non renseignés.

Chapitre 12. Les dates

Vous trouverez les réponses des exercices au Chapitre 20.

12.1. Généralités

Pour avoir la description de la table `EMP`, vous pouvez utiliser la commande « `\d emp` ». Cela vous permettra de vérifier que la colonne `embauche` est bien de type `date`.

1. Rechercher le nom et la date d'embauche des employés.

   Puis recommencez plusieurs fois en faisant précéder votre requête de l'instruction « `SET DATESTYLE TO` *`xxx`* `;` » avec les valeurs suivantes de *`xxx`* :

   ```
   'European' 'ISO' 'Postgres' DEFAULT 'German' 'NonEuropean' 'SQL'
   ```

12.2. Affichage d'une date

L'affichage d'une date peut être modifié en utilisant la fonction `TO_CHAR` :

```
TO_CHAR(arg1,arg2)
```

où `arg1` est une colonne (ou valeur) de type `DATE`, et `arg2` est un masque de sortie (chaîne de caractères).

Tableau 12-1. Liste des masques au format numérique

Masque	Signification
CC	Siècle
YYYY	Année
YYY	3 derniers chiffres de l'année
YY	2 derniers chiffres de l'année
Y	Le dernier chiffre de l'année
Q	Trimestre
WW	Semaine dans l'année
W	Semaine dans le mois
MM	Mois
DDD	Jour dans l'année
DD	Jour dans le mois
D	Jour dans la semaine
HH ou HH12	Heure de la journée (1-12)
HH24	Heure de la journée (0-23)
MI	Minutes
SS	Secondes
SSSSS	Secondes aprés minuit (0-86399)
J	Jour julien

Exemple 12-1. Utilisation de `TO_CHAR` (numérique)

```
SELECT TO_CHAR(embauche,'CC'), TO_CHAR(embauche,'W') FROM emp ;
SELECT TO_CHAR(embauche,'DD/MM/YYYY') FROM emp ;
SELECT TO_CHAR(embauche, WWème semaine, MMème mois') FROM emp ;
```

Tableau 12-2. Liste des masques au format caractère

Masque	Signification
MONTH	Nom du mois en toutes lettres
MON	3 premieres lettres du nom du mois
DAY	Nom du jour en toutes lettres
DY	3 premieres lettres du nom du jour
AM ou PM	Avant midi ou aprés midi
BC ou AD	Avant ou aprés J.C.

Exemple 12-2. Utilisation de TO_CHAR (caractères)

```
SELECT TO_CHAR(embauche,'DAY MONTH YYYY') FROM emp ;
SELECT TO_CHAR(embauche,'DY MON YYYY BC') FROM emp ;
```

Tableau 12-3. Masque de suffixe

Masque	Signification
TH	ST, ND, RD, TH aprés le nombre

Exemple 12-3. Utilisation de TO_CHAR (suffixe)

```
SELECT TO_CHAR(embauche, WWTH semaine, MMTH mois, CCTH siècle') FROM emp ;
```

Le masque d'affichage par défaut est `YYYY-MM-DD`.

2. Rechercher le nom et la date d'embauche (masque `dd/mm/yy`) des employés du département `41`.
3. Rechercher le nom et la date d'embauche (masque `Day DD Month YYYY`) des employés du département `41`.
4. Rechercher le nom et la date d'embauche (masque `DD MON yyyy HH24:MI:SS`) des employés.

12.3. Calculs sur les dates

Une colonne de type `DATE` peut être additionnée à un nombre qui sera considéré comme un nombre de jours même s'il est fractionnaire. Le résultat affiché sera une date.

5. Rechercher le nom, la date d'embauche (masque par défaut) et la date d'embauche `+ 90` jours renommée « `Fin période essai` » des employés du département `41`.

La différence entre deux dates sera un nombre de jours qui peut être fractionnaire si les dates stockées contiennent des heures, des minutes et des secondes.

6. Rechercher le nom et l'intervalle en jours entre aujourd'hui et la date d'embauche (renommé `DELTA`) des employés.

La date actuelle s'obtient par « `CURRENT_DATE` » ;

L'instant actuel (avec heures, minutes, secondes et fractions de secondes) s'obtient par « `NOW()` », qu'on peut transformer en date simple par « `CAST(NOW() AS DATE)` ».

Les calculs sur les dates peuvent servir de critères de restriction.

7. Rechercher le nom, la date d'embauche des employés embauchés moins d'une année après la première embauche d'un employé.

Chapitre 13. Exercices récapitulatifs

Vous trouverez les réponses des exercices au Chapitre 21.

1. Afficher le nombre d'employés de chaque département (en affichant les noms des départements).
2. Afficher dans l'ordre alphabétique la liste des employés qui possédent plus de deux subordonnés (donc au moins 3).
3. Afficher les employés qui ont le salaire minimum et le salaire maximum.
4. Afficher par ordre alphabétique les employés qui ont au moins deux niveaux de subordonnés (ils sont le supérieur de quelqu'un, qui est lui-même le supérieur de quelqu'un).
5. Afficher le nom du département qui a le plus d'employés.
6. Afficher l'employé dont le salaire est le plus proche du salaire moyen des employés.

Chapitre 14. Réponses aux premiers exercices sur la base *jouet*

Vous trouverez les énoncés correspondant au Chapitre 4.

Solution de l'exercice 1

a	b	c
x	m	2
x	n	1
y	m	4
z	p	1

Solution de l'exercice 2

a
x
x
y
z

Solution de l'exercice 3

a
x
z

Solution de l'exercice 4

a
x
x
z

Solution de l'exercice 5

a
x
z

Solution de l'exercice 6

a

a
x
y
x
z

Solution de l'exercice 7

a	**e**
x	8
x	4
x	1
x	8
x	4
x	1
y	8
y	4
y	1
z	8
z	4
z	1

On remarquera que cela donne 12 lignes c'est-à-dire 4×3, soit le produit du nombre de lignes de `UN` et du nombre de lignes de `DEUX`.

Solution de l'exercice 8

a	**e**
x	1
y	8
z	1

Chapitre 15. Réponses aux premiers exercices sur la commande `SELECT`

Vous trouverez les énoncés correspondant au Chapitre 6.

Solution de l'exercice 1

```
SELECT *
  FROM EMP ;
```

Solution de l'exercice 2

```
SELECT *
  FROM DEPT
;
```

Solution de l'exercice 3

```
SELECT NOM, EMBAUCHE, NOSUPR, NODEPT, SALAIRE
  FROM EMP
;
```

Solution de l'exercice 4

```
SELECT TITRE
  FROM EMP
;
```

Solution de l'exercice 5

```
SELECT DISTINCT TITRE
  FROM EMP
;
```

Solution de l'exercice 6

```
SELECT NOM, NOEMP, NODEPT
  FROM EMP
 WHERE TITRE = 'Secrétaire'
;
```

La valeur `Secrétaire` est entre deux « ' » car la colonne `TITRE` est alpha-numérique, ce qui n'est pas le cas pour `NODEPT` qui est numérique.

Solution de l'exercice 7

```
SELECT NOM, NODEPT
  FROM DEPT
 WHERE NODEPT > 40
;
```

Solution de l'exercice 8

```
SELECT NOM, PRENOM
  FROM EMP
 WHERE NOM < PRENOM
;
```

Solution de l'exercice 9

```
SELECT NOM, SALAIRE, NODEPT
  FROM EMP
 WHERE TITRE = 'Représentant'
   AND NODEPT = 35
   AND SALAIRE > 20000
;
```

Solution de l'exercice 10

```
SELECT NOM, TITRE, SALAIRE
  FROM EMP
 WHERE TITRE = 'Représentant'
    OR TITRE = 'Président'
;
```

Solution de l'exercice 11

```
SELECT NOM, TITRE, NODEPT, SALAIRE
  FROM EMP
 WHERE (TITRE = 'Représentant' OR TITRE = 'Secrétaire')
   AND NODEPT = 34
;
```

Si les clauses de restriction sont reliées par « `AND` » ou « `OR` », ATTENTION aux parenthèses.

Solution de l'exercice 12

```
SELECT NOM, TITRE, NODEPT, SALAIRE
  FROM EMP
 WHERE TITRE = 'Représentant'
    OR (TITRE = 'Secrétaire' AND NODEPT = 34)
;
```

Solution de l'exercice 13

```
SELECT NOM, SALAIRE
  FROM EMP
 WHERE SALAIRE >= 20000 AND SALAIRE <= 30000
;
```

La clause « `WHERE SALAIRE >= 20000 AND SALAIRE <= 30000` » peut s'écrire avec l'opérateur « `BETWEEN` » :

```
SELECT NOM, SALAIRE
  FROM EMP
 WHERE SALAIRE BETWEEN 20000 AND 30000
;
```

Solution de l'exercice 14

```
SELECT NOM, TITRE, NODEPT
  FROM EMP
 WHERE TITRE='Représentant' OR TITRE='Secrétaire'
;
```

La clause « `WHERE colonne IN ('val1', 'val2')` » permet d'éviter « `WHERE colonne = 'val1' OR colonne = 'val2'` » :

```
SELECT NOM, TITRE, NODEPT
  FROM EMP
 WHERE TITRE IN ('Représentant' , 'Secrétaire')
;
```

Solution de l'exercice 15

```
SELECT NOM
  FROM EMP
 WHERE NOM LIKE 'H%'
;
```

Solution de l'exercice 16

```
SELECT NOM
  FROM EMP
 WHERE NOM LIKE '%n'
;
```

Solution de l'exercice 17

```
SELECT NOM
  FROM EMP
 WHERE NOM LIKE '__u%'
;
```

Solution de l'exercice 18

```
  SELECT SALAIRE, NOM
    FROM EMP
   WHERE NODEPT = 41
ORDER BY SALAIRE
;
```

Solution de l'exercice 19

```
  SELECT SALAIRE, NOM
    FROM EMP
   WHERE NODEPT = 41
ORDER BY SALAIRE DESC
;
```

Solution de l'exercice 20

```
  SELECT TITRE, SALAIRE, NOM
    FROM EMP
ORDER BY TITRE, SALAIRE DESC
;
```

Solution de l'exercice 21

```
  SELECT TX_COMMISSION, SALAIRE, NOM
    FROM EMP
ORDER BY TX_COMMISSION
;
```

Solution de l'exercice 22

```
SELECT NOM, SALAIRE, TX_COMMISSION, TITRE
  FROM EMP
 WHERE TX_COMMISSION IS NULL
;
```

Solution de l'exercice 23

```
SELECT NOM, SALAIRE, TX_COMMISSION, TITRE
  FROM EMP
 WHERE TX_COMMISSION IS NOT NULL
;
```

Solution de l'exercice 24

```
SELECT NOM, SALAIRE, TX_COMMISSION, TITRE
  FROM EMP
 WHERE TX_COMMISSION < 15
;
```

Par défaut, les colonnes alphanumériques non renseignées ne sont pas considérées comme une suite d'espaces, les colonnes numériques ne sont pas considérées comme égale à 0.

Solution de l'exercice 25

```
SELECT NOM, SALAIRE, TX_COMMISSION, SALAIRE * TX_COMMISSION/100
  FROM EMP
 WHERE TX_COMMISSION IS NOT NULL
;
```

Solution de l'exercice 26

```
  SELECT NOM, SALAIRE, TX_COMMISSION, SALAIRE * TX_COMMISSION/100
    FROM EMP
   WHERE TX_COMMISSION IS NOT NULL
ORDER BY 3
;
```

Solution de l'exercice 27

```
SELECT NOM, SALAIRE * (1 + TX_COMMISSION/100) AS "Rémunération  totale"
  FROM EMP ;
```

Certains salaires semblent vides. C'est parce qu'une expression algébrique[1] qui contient une valeur `NULL` donne toujours un résultat `NULL`. Une valeur `NULL` n'est pas considérée comme zéro.

Solution de l'exercice 28

```
SELECT NOM, SALAIRE * (1 + COALESCE(TX_COMMISSION,0)/100) AS "Remuneration totale"
  FROM EMP
;
```

Solution de l'exercice 29

```
SELECT NOM, SALAIRE AS "PAR MOIS", ROUND (SALAIRE/150,2) AS "PAR HEURE"
  FROM EMP
 WHERE NODEPT = 35
;
```

Solution de l'exercice 30

```
SELECT NOM ||' '|| PRENOM AS "Nom, Prenom"
  FROM EMP
;
```

Solution de l'exercice 31

```
SELECT NOM, NODEPT,
       CASE WHEN NODEPT=10 THEN 'Vu'
            WHEN NODEPT=50 THEN 'Pas vu'
            ELSE 'En cours'
        END AS "Situation"
  FROM DEPT
;
```

Solution de l'exercice 32

```
  SELECT CASE WHEN NODEPT=41 THEN PRENOM
              ELSE NOM
         END AS "NOM OU PRENOM", NODEPT, SALAIRE
    FROM EMP
ORDER BY 2
;
```

Solution de l'exercice 33

```
SELECT SUBSTR (NOM, 1,5)
  FROM EMP
;
```

1. Ici : `SALAIRE*(1+TX_COMMISSION/100)`.

Solution de l'exercice 34

Pour la position à partir du début :

```
SELECT NOM, STRPOS (NOM,'r')
  FROM EMP
;
```

Pour avoir le résultat à partir de la 3ème lettre, il suffit de « supprimer » les 2 premières...

```
SELECT NOM, STRPOS (SUBSTR(NOM,3),'r')
  FROM EMP
;
```

Solution de l'exercice 35

```
SELECT NOM, UPPER(NOM) , LOWER(NOM)
  FROM EMP
 WHERE UPPER(NOM) = UPPER('Vrante')
;
```

Solution de l'exercice 36

```
SELECT NOM, LENGTH(NOM)
  FROM EMP
;
```

Solution de l'exercice 37

```
 SELECT NOM, 'Salaire' AS "TYPE", SALAIRE AS "MONTANT"
   FROM EMP
UNION
 SELECT NOM, 'Commission' AS "TYPE", SALAIRE * TX_COMMISSION/100  AS "MONTANT"
   FROM EMP
  WHERE TX_COMMISSION IS NOT NULL
;
```

Solution de l'exercice 38

```
 SELECT NODEPT  AS "Dep."
   FROM EMP
INTERSECT
 SELECT NODEPT AS "Dep."
   FROM DEPT
;
```

Solution de l'exercice 39

```
 SELECT NODEPT AS "Dep."
   FROM DEPT
EXCEPT
 SELECT NODEPT AS "Dep."
   FROM EMP
;
```

Chapitre 16. Réponses aux exercices sur les jointures et les sous-requêtes

Vous trouverez les énoncés correspondant au Chapitre 7.

Solution de l'exercice 1

```
SELECT PRENOM, NOREGION
  FROM EMP,DEPT
 WHERE EMP.NODEPT = DEPT.NODEPT
;
```

Solution de l'exercice 2

```
  SELECT D.NODEPT, D.NOM, E.NOM
    FROM EMP E, DEPT D
   WHERE E.NODEPT = D.NODEPT
ORDER BY D.NODEPT
;
```

Solution de l'exercice 3

```
SELECT EMP.NOM, DEPT.NOM
  FROM EMP,DEPT
 WHERE EMP.NOM = 'Amartakaldire'
;
```

Solution de l'exercice 4

```
SELECT E.NOM, E.SALAIRE, P.NOM, P.SALAIRE
  FROM EMP E,EMP P
 WHERE E.NOSUPR = P.NOEMP
   AND E.SALAIRE > P.SALAIRE
;
```

Solution de l'exercice 5

```
SELECT X.NOM, X.SALAIRE, Y.NOM, Y.SALAIRE
  FROM EMP X, EMP Y
 WHERE X.SALAIRE > Y.SALAIRE
   AND Y.NOM = 'Amartakaldire'
;
```

Solution de l'exercice 6

```
SELECT TITRE
  FROM EMP
 WHERE NOM ='Amartakaldire'
;
SELECT NOM
  FROM EMP
 WHERE TITRE = 'Représentant'
;
```

Mais aussi... (et c'est mieux !) :

```
SELECT NOM, TITRE
  FROM EMP
 WHERE TITRE = (SELECT TITRE
                  FROM EMP
                 WHERE NOM = 'Amartakaldire')
;
```

Solution de l'exercice 7

```
  SELECT NOM, SALAIRE, NODEPT
    FROM EMP
   WHERE SALAIRE > ANY (SELECT SALAIRE
                          FROM EMP
                         WHERE NODEPT = 31)
ORDER BY 3, 2
;
```

Solution de l'exercice 8

```
  SELECT NOM, SALAIRE, NODEPT
    FROM EMP
   WHERE SALAIRE > ALL (SELECT SALAIRE
                          FROM EMP
                         WHERE NODEPT = 31)
ORDER BY 3, 2
;
```

Solution de l'exercice 9

```
SELECT NOM, TITRE
  FROM EMP
 WHERE NODEPT = 31
   AND TITRE IN (SELECT TITRE
                   FROM EMP
                  WHERE NODEPT = 32)
;
```

Solution de l'exercice 10

```
SELECT NOM, TITRE
  FROM EMP
 WHERE NODEPT = 31
   AND TITRE NOT IN (SELECT TITRE
                       FROM EMP
                      WHERE NODEPT = 32)
;
```

Solution de l'exercice 11

```
SELECT NOM, TITRE, SALAIRE
  FROM EMP
 WHERE (TITRE, SALAIRE) = (SELECT TITRE, SALAIRE
                             FROM EMP
                            WHERE NOM = 'Fairant')
;
```

Solution de l'exercice 12

```
  SELECT NODEPT, NOM, SALAIRE
    FROM EMP E
   WHERE SALAIRE > (SELECT AVG(SALAIRE)
                      FROM EMP
                     WHERE NODEPT = E.NODEPT)
ORDER BY NODEPT
;
```

Solution de l'exercice 13

```
SELECT NOEMP, NOM, PRENOM
  FROM EMP E
 WHERE EXISTS (SELECT NULL
                 FROM EMP
                WHERE TITRE='Représentant'
                  AND NODEPT = E.NODEPT)
;
```

Solution de l'exercice 14

```
SELECT NODEPT, NOM AS "Dept"
  FROM DEPT
 WHERE NODEPT NOT IN (SELECT DEPT.NODEPT
                        FROM EMP, DEPT
                       WHERE EMP.NODEPT = DEPT.NODEPT)
;
```

Solution de l'exercice 15

Avec `LEFT JOIN` cela donne :

```
  SELECT D.NODEPT,D.NOM,E.NOM
    FROM DEPT D LEFT JOIN EMP E
      ON E.NODEPT=D.NODEPT
ORDER BY D.NOM
;
```

On peut aussi se passer de `LEFT JOIN` (certains systèmes ne l'autorisent pas), mais c'est moins simple. Avec une union cela donne :

```
(
  SELECT DEPT.NODEPT, DEPT.NOM AS "Dept", EMP.NOM AS "Emp"
    FROM EMP, DEPT
   WHERE EMP.NODEPT = DEPT.NODEPT
UNION
  SELECT NODEPT, NOM AS "Dept", NULL AS "Emp"
    FROM DEPT
)
ORDER BY DEPT.NODEPT
;
```

Mais on a une ligne vide pour chaque département, même quand ce n'est pas nécessaire. On peut améliorer ainsi :

```
(
   SELECT DEPT.NODEPT, DEPT.NOM AS "Dept", EMP.NOM AS "Emp"
     FROM EMP, DEPT
    WHERE EMP.NODEPT = DEPT.NODEPT
 UNION
   SELECT NODEPT, NOM AS "Dept", NULL AS "Emp"
     FROM DEPT
    WHERE NODEPT NOT IN (SELECT DEPT.NODEPT
```

```
                                FROM EMP, DEPT
                               WHERE EMP.NODEPT = DEPT.NODEPT)
    )
    ORDER BY "Dept"
    ;
```

Chapitre 17. Réponses aux exercices sur modification de base, etc.

Vous trouverez les énoncés correspondant au Chapitre 9.

Solution de l'exercice 1

```
CREATE TABLE TRUC
    AS (SELECT *
        FROM EMP)
;
SELECT * FROM TRUC;
```

Solution de l'exercice 2

```
INSERT INTO AFFAIRE VALUES (101, 'ALPHA' ,250000)
;
INSERT INTO AFFAIRE VALUES (102, 'BETA' , 175000)
;
INSERT INTO AFFAIRE VALUES (103, 'GAMMA' ,95000)
;
SELECT * FROM AFFAIRE
;
SELECT * FROM AFFAIRE ;
```

Solution de l'exercice 3

```
BEGIN
;
SELECT * FROM EMP WHERE NOM ='Kaécouté'
;
UPDATE EMP
   SET TITRE = 'Représentant' ,
       SALAIRE = 20000 ,
       TX_COMMISSION = 15
 WHERE NOM = 'Kaécouté'
;
SELECT * FROM EMP WHERE NOM ='Kaécouté'
;
COMMIT ;
```

Solution de l'exercice 4

```
BEGIN
;
SELECT *
  FROM EMP
 WHERE NOM IN (SELECT NOM
                 FROM BONUS)
;
UPDATE EMP
   SET SALAIRE = SALAIRE * CAST (1.05 AS NUMERIC)
 WHERE NOM IN (SELECT NOM
                 FROM BONUS)
;
SELECT *
  FROM EMP
 WHERE NOM IN (SELECT NOM
                 FROM BONUS)
;
COMMIT ;
```

Solution de l'exercice 5

```
BEGIN
;
SELECT * FROM BONUS
;
DELETE FROM BONUS WHERE NOM = 'Zoudanlkou'
;
SELECT *  FROM BONUS
;
COMMIT ;
```

Solution de l'exercice 6

```
BEGIN
;
SELECT NOM FROM BONUS
;
DELETE FROM BONUS WHERE NOM = 'Amartakaldire'
;
SELECT NOM FROM BONUS
;
COMMIT;
```

Solution de l'exercice 7

```
BEGIN
;
SELECT * FROM EMP WHERE NOM ='Amartakaldire'
;
DELETE FROM EMP WHERE NOM = 'Amartakaldire'
;
SELECT *  FROM EMP WHERE NOM ='Amartakaldire'
;
ROLLBACK
;
SELECT * FROM EMP WHERE NOM ='Amartakaldire' ;
```

Solution de l'exercice 8

```
BEGIN
;
SELECT * FROM BONUS
;
DELETE  FROM BONUS
;
SELECT * FROM BONUS
;
ROLLBACK
;
SELECT * FROM BONUS
;
BEGIN
;
DELETE FROM BONUS
;
SELECT * FROM BONUS
;
COMMIT
;
ROLLBACK
;
SELECT * FROM BONUS;
```

Solution de l'exercice 9

```
ALTER TABLE EMP
      ADD NOAFF NUMERIC(3) ;
```

Solution de l'exercice 10

```
BEGIN
;
SELECT NOM, NOAFF FROM EMP
;
UPDATE EMP
   SET NOAFF = 101
 WHERE NODEPT IN (10,50)
;
SELECT NOM, NOAFF FROM EMP WHERE NOAFF = 101
;
UPDATE EMP
   SET NOAFF = 102
 WHERE NODEPT IN (31,32,33,34,35)
;
SELECT NOM, NOAFF FROM EMP WHERE NOAFF = 102
;
UPDATE EMP
   SET NOAFF = 103
 WHERE NODEPT IN (41,42,43,44,45)
;
SELECT NOM, NOAFF FROM EMP
;
COMMIT ;
```

Solution de l'exercice 11

```
DROP TABLE AFFAIRE ;
```

Pour la table `EMP` voir plus haut.

Solution de l'exercice 12

```
BEGIN
;
CREATE VIEW DEPT4
    AS SELECT NOEMP, NOM, TITRE
         FROM EMP
        WHERE NODEPT > 40
;
SELECT * FROM DEPT4
;
SELECT NOM, TITRE FROM DEPT4 WHERE NOEMP > 20
;
UPDATE EMP
   SET TITRE = 'Chef Entrepôt'
 WHERE NOM = 'Phototetedemort'
;
SELECT * FROM DEPT4
;
COMMIT ;
```

Solution de l'exercice 13

```
UPDATE DEPT4
   SET TITRE = 'Chef Entrepôt'
 WHERE NOM = 'Anchier'
;
SELECT * FROM DEPT4 ;
```

Solution de l'exercice 14

```
CREATE VIEW EMPDEPT(NOM, NOSERV, NOMSERV)
    AS SELECT EMP.NOM, EMP.NODEPT, DEPT.NOM
         FROM EMP,DEPT
        WHERE EMP.NODEPT = DEPT.NODEPT
;
SELECT NOM, NOMSERV FROM EMPDEPT WHERE NOSERV = 50 ;
```

Solution de l'exercice 15

```
CREATE VIEW PAYE (NOM, SALAIRE, SALANNUEL)
    AS SELECT NOM, SALAIRE, SALAIRE * 13
         FROM EMP
;
SELECT * FROM PAYE ;
```

Chapitre 18. Réponses aux exercices sur les droits

Vous trouverez les énoncés correspondant au Chapitre 10.

Solution de l'exercice 1

```
GRANT SELECT
   ON DEPT
   TO nom de l'autre user ;
```

Solution de l'exercice 2

```
GRANT INSERT,UPDATE
   ON DEPT
   TO nom de l'autre user ;
```

Solution de l'exercice 3

```
GRANT ALL ON DEPT TO nom de l'autre user ;
```

Solution de l'exercice 4

```
CREATE VIEW EMPVUE AS
            SELECT NOEMP, NOM, TITRE, NODEPT
              FROM EMP
;
GRANT SELECT
   ON EMPVUE
   TO nom de l'autre user
;
```

Solution de l'exercice 5

```
CREATE VIEW EMP50 AS
            SELECT *
              FROM EMP
             WHERE NODEPT = 50
;
GRANT ALL
   ON EMP50
   TO nom de l'autre user
;
```

Solution de l'exercice 6

```
REVOKE INSERT ON DEPT FROM nom de l'autre user ;
```

Chapitre 19. Réponses aux exercices sur les groupes

Vous trouverez les énoncés correspondant au Chapitre 11.

Solution de l'exercice 1

```
SELECT MAX(SALAIRE) ,
       MIN(SALAIRE) ,
       MAX(SALAIRE) - MIN(SALAIRE) AS "Différence"
  FROM EMP
;
```

Solution de l'exercice 2

```
SELECT COUNT(DISTINCT TITRE)
  FROM EMP
;
```

Essayez aussi, pour voir la différence\,:

```
SELECT COUNT(TITRE)
  FROM EMP
;
```

Solution de l'exercice 3

```
  SELECT NODEPT, TITRE, COUNT(*)
    FROM EMP
GROUP BY NODEPT, TITRE
;
```

Solution de l'exercice 4

```
  SELECT DEPT.NOM, TITRE, COUNT(*)
    FROM EMP,DEPT
   WHERE DEPT.NODEPT = EMP.NODEPT
GROUP BY DEPT.NOM, TITRE
;
```

Solution de l'exercice 5

```
  SELECT TITRE, AVG(SALAIRE)
    FROM EMP
GROUP BY TITRE
  HAVING AVG(SALAIRE) > (SELECT AVG(SALAIRE)
                           FROM EMP
                          WHERE TITRE = 'Représentant'
                        )
;
```

Solution de l'exercice 6

```
SELECT COUNT(SALAIRE),COUNT(TX_COMMISSION)
  FROM EMP
;
```

Solution de l'exercice 7

```
SELECT AVG(TX_COMMISSION) ,AVG(COALESCE(TX_COMMISSION,0))
  FROM EMP
;
```

Chapitre 20. Réponses aux exercices sur les dates

Vous trouverez les énoncés correspondant au Chapitre 12.

Solution de l'exercice 1

```
SELECT nom,embauche FROM emp ;

SET DATESTYLE TO 'European';
SELECT nom,embauche from emp;

SET DATESTYLE TO 'ISO';
SELECT nom,embauche from emp;

SET DATESTYLE TO 'Postgres';
SELECT nom,embauche from emp;

SET DATESTYLE TO DEFAULT;
SELECT nom,embauche from emp;

SET DATESTYLE TO 'German';
SELECT nom,embauche from emp;

SET DATESTYLE TO 'NonEuropean';
SELECT nom,embauche from emp;

SET DATESTYLE TO 'SQL';
SELECT nom,embauche from emp;
```

Solution de l'exercice 2

```
SELECT nom,TO_CHAR(embauche,'dd/mm/yy') FROM emp WHERE nodept=41;
```

Solution de l'exercice 3

```
SELECT nom,TO_CHAR(embauche,'Day DD Month YYYY') FROM emp WHERE nodept=41;
```

Solution de l'exercice 4

```
SELECT nom,TO_CHAR(embauche,'DD MON yyyy HH24:MI:SS') FROM emp WHERE nodept=41;
```

Solution de l'exercice 5

```
SELECT nom,embauche,embauche+90 AS "Fin période essai" FROM emp WHERE nodept=41;
```

Solution de l'exercice 7

```
SELECT nom,embauche FROM emp WHERE nodept=41;
```

Solution de l'exercice 6

```
SELECT nom,embauche,CAST(NOW() AS DATE)-embauche AS "Delta" FROM emp;
```

Solution de l'exercice 7

```
SELECT nom,embauche FROM emp
 WHERE embauche-(SELECT MIN(embauche) FROM emp) < 365;
```

Chapitre 21. Réponses aux exercices récapitulatifs

Vous trouverez les énoncés correspondant au Chapitre 13.

Solution de l'exercice 1

```
 SELECT d.nodept,d.nom,COUNT(*)
   FROM dept d,emp e
  WHERE d.nodept=e.nodept
  GROUP BY d.nodept,d.nom
  ;
```

Solution de l'exercice 2

```
  SELECT e.nom,e.prenom,COUNT(*)
    FROM emp e,emp s
   WHERE e.noemp=s.nosupr
GROUP BY e.noemp,e.nom,e.prenom
  HAVING count(*)>2
ORDER BY e.nom,e.prenom
;
```

Solution de l'exercice 3

```
 (SELECT prenom,nom,salaire,'Minimum' AS "min ou max?"
    FROM emp
   WHERE salaire=(SELECT MIN(salaire)
                    FROM emp)
 )
 UNION
 (SELECT prenom,nom,salaire,'Maximum' AS "min ou max?"
    FROM emp
   WHERE salaire=(SELECT MAX(salaire)
                   FROM emp)
 )
 ;
```

Solution de l'exercice 4

```
  SELECT DISTINCT e.prenom,e.nom
    FROM emp e,emp a,emp b
   WHERE e.noemp=a.nosupr
     AND a.noemp=b.nosupr
ORDER BY e.nom,e.prenom
;
```

Solution de l'exercice 5

```
SELECT nodept,nom
  FROM (
       SELECT d.nodept,d.nom,count(*) AS xxx
         FROM dept d,emp e
        WHERE d.nodept=e.nodept
     GROUP BY d.nodept,d.nom) yyy
 WHERE xxx=(SELECT MAX(xxx) FROM (
                                  SELECT COUNT(*) AS xxx
                                    FROM dept d,emp e
                                   WHERE d.nodept=e.nodept
                                GROUP BY d.nodept) yyy
            )
```

```
;
```

Solution de l'exercice 6

```
SELECT nom,prenom,salaire
  FROM emp
 WHERE (SELECT abs(salaire
                  -(SELECT avg(salaire)
                      FROM emp)))
      =(SELECT min(diff)
          FROM (SELECT abs(salaire
                           -(SELECT avg(salaire)
                               FROM emp)) as diff
                  FROM emp)  a)
;
```

Index