Bases de données et langage SQL

Table des matières

4. Chapitre 4 - Langage SQL▲

4-1. Introduction▲

4-1-1. Présentation générale▲

4-1-1-a. Introduction▲

Le langage SQL (Structured Query Language) peut être considéré comme le langage d'accès normalisé aux bases de données. Il est aujourd'hui supporté par la plupart des produits commerciaux que ce soit par les systèmes de gestion de bases de données micro tel que Access ou par les produits plus professionnels tels que Oracle. Il a fait l'objet de plusieurs normes ANSI/ISO dont la plus répandue aujourd'hui est la norme SQL2 qui a été définie en 1992.

Le succès du langage SQL est dû essentiellement à sa simplicité et au fait qu'il s'appuie sur le schéma conceptuel pour énoncer des requêtes en laissant le SGBD responsable de la stratégie d'exécution. Le langage SQL propose un langage de requêtes ensembliste et assertionnel. Néanmoins, le langage SQL ne possède pas la puissance d'un langage de programmation : entrées/sorties, instructions conditionnelles, boucles et affectations. Pour certains traitements il est donc nécessaire de coupler le langage SQL avec un langage de programmation plus complet.

De manière synthétique, on peut dire que SQL est un langage relationnel, il manipule donc des tables (i.e. des relations, c'est-à-dire des ensembles) par l'intermédiaire de requêtes qui produisent également des tables.

4-1-1-b. Historique rapide▲

- En 1970, E.F. CODD, directeur de recherche du centre IBM de San José, invente le modèle relationnel qui repose sur une algèbre relationnelle. Ce modèle provoque une révolution dans l'approche des bases des données.
- En 1977, création du langage SEQUEL (Structured English Query Language) et mise en place du Système R, prototype de base de données reposant sur la théorie de CODD. SEQUEL continue de s'enrichir pour devenir SQL (Structured Query Language).
- En 1981, la société ORACLE CORP lance la première version de son système de gestion de base de données relationnelle (SGBDR), IBM sort SQL/DS et RTI lance INGRES.
- En 1982, IBM sort SQL/DS pour son environnement VM/CMS et l'ANSI (American National Standard Institute) lance un projet de normalisation d'un langage relationnel.
- En 1983, IBM lance DB2 pour l'environnement MVS.
- En 1986, la société SYBASE lance son SGBDR conçu selon le modèle Client-Serveur.
- La première norme SQL (SQL-1) de l'ISO (International Standard Organisation) apparaît. Il existe désormais plusieurs dizaines de produits proposant le langage SQL

et tournant sur des machines allant des micros aux gros systèmes.
- Depuis, les différents produits phares ont évolué, la norme SQL est passée à SQL-2, puis SQL-3. SQL est désormais un langage incontournable pour tout SGBD moderne. Par contre, bien qu'une norme existe, on assiste à une prolifération de dialectes propres à chaque produit : soit des sous-ensembles de la norme (certaines fonctionnalités n'étant pas implantées), soit des sur-ensembles (ajout de certaines fonctionnalités, propres à chaque produit).

Oracle et Informix dominent le marché actuel, SQL-Server (de Microsoft) tente de s'imposer dans le monde des PC sous NT. À côté des ces produits, très chers, existent heureusement des systèmes libres et gratuits : MySQL et PostgreSQL sont les plus connus.

Bien que ces SGBDR n'aient pas la puissance des produits commerciaux, certains s'en approchent de plus en plus. Les différences notables concernent principalement les environnements de développement qui sont de véritables ateliers logiciels sous Oracle et qui sont réduits à des interfaces de programmation C, Python, Perl sous PostgreSQL. Il en va de même pour les interfaces utilisateurs : il en existe pour PostgreSQL, mais ils n'ont certainement pas la puissance de leurs équivalents commerciaux.

4-1-1-c. Terminologie▲

Modèle relationnel		
Français	**Anglais**	**Standard SQL**
Relation	Relation	Table
Domaine	Domain	Domaine
Attribut	Attribute	Colonne
N-uplet	tuple	ligne
Clé primaire	Primary key	Primary key

4-1-2. Catégories d'instructions▲

Les instructions SQL sont regroupées en catégories en fonction de leur utilité et des entités manipulées. Nous pouvons distinguer cinq catégories, qui permettent :

1. la définition des éléments d'une base de données (tables, colonnes, clés, index, contraintes…),
2. la manipulation des données (insertion, suppression, modification, extraction…),
3. la gestion des droits d'accès aux données (acquisition et révocation des droits),
4. la gestion des transactions,
5. et enfin le SQL intégré.

4-1-2-a. Langage de définition de données▲

Le **langage de définition de données** (LDD, ou Data Definition Language, soit DDL en anglais) est un langage orienté au niveau de la structure de la base de données. Le LDD permet de créer, modifier, supprimer des objets. Il permet également de définir le domaine des données (nombre, chaîne de caractères, date, booléen…) et d'ajouter des contraintes de valeur sur les données. Il permet enfin d'autoriser ou d'interdire l'accès aux données et d'activer ou de désactiver l'audit pour un utilisateur donné.

Les instructions du LDD sont : CREATE, ALTER, DROP, AUDIT, NOAUDIT, ANALYZE, RENAME, TRUNCATE.

4-1-2-b. Langage de manipulation de données▲

Le **langage de manipulation de données** (LMD, ou Data Manipulation Language, soit DML en anglais) est l'ensemble des commandes concernant la manipulation des données dans une base de données. Le LMD permet l'ajout, la suppression et la modification de lignes, la visualisation du contenu des tables et leur verrouillage.

Les instructions du LMD sont : INSERT, UPDATE, DELETE, SELECT, EXPLAIN, PLAN, LOCK TABLE.

Ces éléments doivent être validés par une transaction pour qu'ils soient pris en compte.

4-1-2-c. Langage de protections d'accès▲

Le **langage de protections d'accès** (ou Data Control Language, soit DCL en anglais) s'occupe de gérer les droits d'accès aux tables.

Les instructions du DCL sont : GRANT, REVOKE.

4-1-2-d. Langage de contrôle de transaction▲

Le **langage de contrôle de transaction** (ou Transaction Control Language, soit TCL en anglais) gère les modifications faites par le LMD, c'est-à-dire les caractéristiques des transactions et la validation et l'annulation des modifications.

Les instructions du TCL sont : COMMIT, SAVEPOINT, ROLLBACK, SET TRANSACTION

4-1-2-e. SQL intégré▲

Le **SQL intégré** (Embedded SQL) permet d'utiliser SQL dans un langage de troisième génération (C, Java, Cobol, etc.) :

- déclaration d'objets ou d'instructions ;
- exécution d'instructions ;
- gestion des variables et des curseurs ;
- traitement des erreurs.

Les instructions du SQL intégré sont : DECLARE, TYPE, DESCRIBE, VAR, CONNECT, PREPARE, EXECUTE, OPEN, FETCH, CLOSE, WHENEVER.

4-1-3. PostgreSQL▲

Les systèmes traditionnels de gestion de bases de données relationnelles (SGBDR) offrent un modèle de données composé d'une collection de relations contenant des attributs relevant chacun d'un type spécifique. Les systèmes commerciaux gèrent par exemple les nombres décimaux, les entiers, les chaînes de caractères, les monnaies et les dates. Il est communément admis que ce modèle est inadéquat pour les applications de traitement de données de l'avenir, car, si le modèle relationnel a remplacé avec succès les modèles précédents en partie grâce à sa « simplicité spartiate », cette dernière complique cependant l'implémentation de certaines applications. PostgreSQL apporte une puissance additionnelle substantielle en incorporant les quatre concepts de base suivants afin que les utilisateurs puissent facilement étendre le système : classes, héritage, types, fonctions. D'autres fonctionnalités accroissent la puissance et la souplesse : contraintes, déclencheurs, règles, intégrité des transactions.

Ces fonctionnalités placent PostgreSQL dans la catégorie des bases de données relationnel-objet. Ne confondez pas cette catégorie avec celle des serveurs d'objets qui ne tolère pas aussi bien les langages traditionnels d'accès aux SGBDR. Ainsi, bien que PostgreSQL possède certaines fonctionnalités orientées objet, il appartient avant tout au monde des SGBDR. C'est essentiellement l'aspect SGBDR de PostgreSQL que nous aborderons dans ce cours.

L'une des principales qualités de PostgreSQL est d'être un logiciel libre, c'est-à-dire gratuit et dont les sources sont disponibles. Il est possible de l'installer sur les systèmes Unix/Linux et Win32.

PostgreSQL fonctionne selon une architecture client/serveur, il est ainsi constitué :

- d'une partie serveur, c'est-à-dire une application fonctionnant sur la machine hébergeant la base de données (le serveur de bases de données) capable de traiter les requêtes des clients ; il s'agit dans le cas de PostgreSQL d'un programme résident en mémoire appelé postmaster ;
- d'une partie client (psql) devant être installée sur toutes les machines nécessitant d'accéder au serveur de base de données (un client peut éventuellement fonctionner sur le serveur lui-même).

Les clients (les machines sur lesquelles le client PostgreSQL est installé) peuvent interroger le serveur de bases de données à l'aide de requêtes SQL.

4-2. Définir une base ? Langage de définition de données (LDD)▲

4-2-1. Introduction aux contraintes d'intégrité▲

Soit le schéma relationnel minimaliste suivant:

- Acteur(Num-Act, Nom, Prénom)
- Jouer(Num-Act, Num-Film)
- Film(Num-Film, Titre, Année)

4-2-1-a. Contrainte d'intégrité de domaine▲

Toute comparaison d'attributs n'est acceptée que si ces attributs sont définis sur le même domaine. Le SGBD doit donc constamment s'assurer de la validité des valeurs d'un attribut. C'est pourquoi la commande de création de tables doit préciser, en plus du nom, le type de chaque colonne.

Par exemple, pour la table Film, on précisera que le Titre est une chaîne de caractères et l'Année une date. Lors de l'insertion de n-uplets dans cette table, le système s'assurera que les différents champs du n-uplet satisfont les contraintes d'intégrité de domaine des attributs précisés lors de la création de la base. Si les contraintes ne sont pas satisfaites, le n-uplet n'est, tout simplement, pas inséré dans la table.

4-2-1-b. Contrainte d'intégrité de table (ou de relation ou d'entité)▲

Lors de l'insertion de n-uplets dans une table (i.e. une relation), il arrive qu'un attribut soit inconnu ou non défini. On introduit alors une valeur conventionnelle notée NULL et appelée valeur nulle.

Cependant, une clé primaire ne peut avoir une valeur nulle. De la même manière, une clé primaire doit toujours être unique dans une table. Cette contrainte forte qui porte sur la clé primaire est appelée contrainte d'intégrité de relation.

Tout SGBD relationnel doit vérifier l'unicité et le caractère défini (NOT NULL**) des valeurs de la clé primaire**.

4-2-1-c. Contrainte d'intégrité de référence▲

Dans tout schéma relationnel, il existe deux types de relation :

- les relations qui représentent des entités de l'univers modélisé ; elles sont qualifiées de statiques, ou d'indépendantes ; les relations Acteur et Film en sont des exemples ;
- les relations dont l'existence des n-uplets dépend des valeurs d'attributs situées dans d'autres relations ; il s'agit de relations dynamiques ou dépendantes ; la relation Jouer en est un exemple.

Lors de l'insertion d'un n-uplet dans la relation Jouer, le SGBD doit vérifier que les valeurs Num-Act et Num-Film correspondent bien, respectivement, à une valeur de Num-Act

existant dans la relation Acteur et une valeur Num-Film existant dans la relation Film.

Lors de la suppression d'un n-uplet dans la relation Acteur, le SGBD doit vérifier qu'aucun n-uplet de la relation Jouer ne fait référence, par l'intermédiaire de l'attribut Num-Act, au n-uplet que l'on cherche à supprimer. Le cas échéant, c'est-à-dire si une, ou plusieurs, valeur correspondante de Num-Act existe dans Jouer, quatre possibilités sont envisageables :

- interdire la suppression ;
- supprimer également les n-uplets concernés dans Jouer ;
- avertir l'utilisateur d'une incohérence ;
- mettre les valeurs des attributs concernés à une valeur nulle dans la table Jouer, si l'opération est possible (ce qui n'est pas le cas si ces valeurs interviennent dans une clé primaire) ;

4-2-2. Créer une table : CREATE TABLE▲

4-2-2-a. Introduction▲

Une table est un ensemble de lignes et de colonnes. La création consiste à définir (en fonction de l'analyse) le nom de ces colonnes, leur format (type), la valeur par défaut à la création de la ligne (DEFAULT) et les règles de gestion s'appliquant à la colonne (CONSTRAINT).

4-2-2-b. Création simple▲

La commande de création de tables la plus simple ne comportera que le nom et le type de chaque colonne de la table. À la création, la table sera vide, mais un certain espace lui sera alloué. La syntaxe est la suivante :

Sélectionnez

```
CREATE TABLE nom_table (nom_col1 TYPE1, nom_col2 TYPE2, ...)
```

Quand on crée une table, il faut définir les contraintes d'intégrité que devront respecter les données que l'on mettra dans la table (cf. section 4.2.3Contraintes d'intégrité).

4-2-2-c. Les types de données▲

Les types de données peuvent être :

INTEGER :

- Ce type permet de stocker des entiers signés codés sur 4 octets.

BIGINT :

- Ce type permet de stocker des entiers signés codés sur 8 octets.

REAL :

- Ce type permet de stocker des réels comportant 6 chiffres significatifs codés sur 4 octets.

DOUBLE PRÉCISION :

- Ce type permet de stocker des réels comportant 15 chiffres significatifs codés sur 8 octets.

NUMERIC[(précision, [longueur])] :

- Ce type de données permet de stocker des données numériques à la fois entières et réelles avec une précision de 1000 chiffres significatifs. longueur précise le nombre maximum de chiffres significatifs stockés et précision donne le nombre maximum de chiffres après la virgule.

CHAR(longueur) :

- Ce type de données permet de stocker des chaînes de caractères de longueur fixe. longueur doit être inférieur à 255, sa valeur par défaut est 1.

VARCHAR(longueur) :

- Ce type de données permet de stocker des chaînes de caractères de longueur variable. longueur doit être inférieur à 2000, il n'y a pas de valeur par défaut.

DATE :

- Ce type de données permet de stocker des données constituées d'une date.

TIMESTAMP :

- Ce type de données permet de stocker des données constituées d'une date et d'une heure.

BOOLEAN :

- Ce type de données permet de stocker des valeurs Booléenne.

MONEY :

- Ce type de données permet de stocker des valeurs monétaires.

TEXT :

- Ce type de données permet de stocker des chaînes de caractères de longueur variable.

4-2-2-d. Création avec Insertion de données▲

On peut insérer des données dans une table lors de sa création par la commande suivante :

Sélectionnez

```
CREATE TABLE nom_table [(nom_col1, nom_col2, ...)] AS SELECT ...
```

On peut ainsi, en un seul ordre SQL créer une table et la remplir avec des données provenant du résultat d'un SELECT (cf. section 4.4Interroger une base (LMD) : SELECT (1re partie) et 4.5Interroger une base (LMD) : SELECT (2e partie)). Si les types des colonnes ne sont pas spécifiés, ils correspondront à ceux du SELECT. Il en va de même pour les noms des colonnes. Le SELECT peut contenir des fonctions de groupes, mais pas d'ORDER BY (cf. section 4.5.2Les clauses GROUP BY et HAVING et les fonctions d'agrégation et 4.4.6La clause ORDER BY), car les lignes d'une table ne peuvent pas être classées.

4-2-3. Contraintes d'intégrité▲

4-2-3-a. Syntaxe▲

À la création d'une table, les contraintes d'intégrité se déclarent de la façon suivante :

Sélectionnez

```
CREATE TABLE nom_table (
nom_col_1 type_1 [CONSTRAINT nom_1_1] contrainte_de_colonne_1_1
                 [CONSTRAINT nom_1_2] contrainte_de_colonne_1_2
                 ...                  ...
                 [CONSTRAINT nom_1_m] contrainte_de_colonne_2_m,
nom_col_2 type_2 [CONSTRAINT nom_2_1] contrainte_de_colonne_2_1
                 [CONSTRAINT nom_2_2] contrainte_de_colonne_2_2
                 ...                  ...
                 [CONSTRAINT nom_2_m] contrainte_de_colonne_2_m,
...
nom_col_n type_n [CONSTRAINT nom_n_1] contrainte_de_colonne_n_1
                 [CONSTRAINT nom_n_2] contrainte_de_colonne_n_2
                 ...                  ...
                 [CONSTRAINT nom_n_m] contrainte_de_colonne_n_m,
[CONSTRAINT nom_1] contrainte_de_table_1,
[CONSTRAINT nom_2] contrainte_de_table_2,
...                 ...
[CONSTRAINT nom_p] contrainte_de_table_p
)
```

4-2-3-b. Contraintes de colonne▲

Les différentes contraintes de colonne que l'on peut déclarer sont les suivantes :

NOT NULL **ou** NULL :

- Interdit (NOT NULL) ou autorise (NULL) l'insertion de valeur NULL pour cet attribut.

UNIQUE :

- Désigne l'attribut comme clé secondaire de la table. Deux n-uplets ne peuvent recevoir des valeurs identiques pour cet attribut, mais l'insertion de valeur NULL est toutefois autorisée. Cette contrainte peut apparaître plusieurs fois dans l'instruction.

PRIMARY KEY :

- Désigne l'attribut comme clé primaire de la table. La clé primaire étant unique, cette contrainte ne peut apparaître qu'une seule fois dans l'instruction. La définition d'une clé primaire composée se fait par l'intermédiaire d'une contrainte de table. En fait, la contrainte PRIMARY KEY est totalement équivalente à la contrainte UNIQUE NOT NULL.

REFERENCES table [(colonne)] [ON DELETE CASCADE] :

- Contrainte d'intégrité référentielle pour l'attribut de la table en cours de définition. Les valeurs prises par cet attribut doivent exister dans l'attribut colonne qui possède une contrainte PRIMARY KEY ou UNIQUE dans la table table. En l'absence de précision d'attribut colonne, l'attribut retenu est celui correspondant à la clé primaire de la table table spécifiée.

CHECK (condition) :

- Vérifie lors de l'insertion de n-uplets que l'attribut réalise la condition condition.

DEFAULT valeur :

- Permet de spécifier la valeur par défaut de l'attribut.

4-2-3-c. Contraintes de table▲

Les différentes contraintes de table que l'on peut déclarer sont les suivantes :

PRIMARY KEY (colonne…) :

- Désigne la concaténation des attributs cités comme clé primaire de la table. Cette contrainte ne peut apparaître qu'une seule fois dans l'instruction.

UNIQUE (colonne…) :

- Désigne la concaténation des attributs cités comme clé secondaire de la table. Dans ce cas, au moins une des colonnes participant à cette clé secondaire doit permettre de distinguer le n-uplet. Cette contrainte peut apparaître plusieurs fois dans l'instruction.

FOREIGN KEY (colonne…) REFERENCES table [(colonne…)]

[ON DELETE CASCADE | SET NULL] :

- Contrainte d'intégrité référentielle pour un ensemble d'attributs de la table en cours de définition. Les valeurs prises par ces attributs doivent exister dans l'ensemble d'attributs spécifié et posséder une contrainte PRIMARY KEY ou UNIQUE dans la table table.

CHECK (condition) :

- Cette contrainte permet d'exprimer une condition qui doit exister entre plusieurs attributs de la ligne.

Les contraintes de tables portent sur plusieurs attributs de la table sur laquelle elles sont définies. Il n'est pas possible de définir une contrainte d'intégrité utilisant des attributs provenant de deux ou plusieurs tables. Ce type de contrainte sera mis en œuvre par l'intermédiaire de déclencheurs de base de données (trigger).

4-2-3-d. Complément sur les contraintes▲

ON DELETE CASCADE :

- Demande la suppression des n-uplets dépendants, dans la table en cours de définition, quand le n-uplet contenant la clé primaire référencée est supprimé dans la table maître.

ON DELETE SET NULL :

- Demande la mise à NULL des attributs constituant la clé étrangère qui font référence au n-uplet supprimé dans la table maître.

La suppression d'un n-uplet dans la table maître pourra être impossible s'il existe des n-uplets dans d'autres tables référençant cette valeur de clé primaire et ne spécifiant pas l'une de ces deux options.

4-2-4. Supprimer une table : DROP TABLE▲

Supprimer une table revient à éliminer sa structure et toutes les données qu'elle contient. Les index associés sont également supprimés.

La syntaxe est la suivante :

Sélectionnez

```
DROP TABLE nom_table
```

4-2-5. Modifier une table : ALTER TABLE▲

4-2-5-a. Ajout ou modification de colonne▲

Sélectionnez

```
ALTER TABLE nom_table {ADD/MODIFY} ([nom_colonne type [contrainte], ...])
```

4-2-5-b. Ajout d'une contrainte de table▲

Sélectionnez

```
ALTER TABLE nom_table ADD [CONSTRAINT nom_contrainte] contrainte
```

La syntaxe de déclaration de contrainte est identique à celle vue lors de la création de tables.

Si des données sont déjà présentes dans la table au moment où la contrainte d'intégrité est ajoutée, toutes les lignes doivent vérifier la contrainte. Dans le cas contraire, la contrainte n'est pas posée sur la table.

4-2-5-c. Renommer une colonne▲

Sélectionnez

```
ALTER TABLE nom_table RENAME COLUMN ancien_nom TO nouveau_nom
```

4-2-5-d. Renommer une table▲

Sélectionnez

```
ALTER TABLE nom_table RENAME TO nouveau_nom
```

4-3. Modifier une base - Langage de manipulation de données (LMD)▲

4-3-1. Insertion de n-uplets : INSERT INTO▲

La commande INSERT permet d'insérer une ligne dans une table en spécifiant les valeurs à insérer. La syntaxe est la suivante :

Sélectionnez

```
INSERT INTO nom_table(nom_col_1, nom_col_2, ...)
VALUES (val_1, val_2, ...)
```

La liste des noms de colonne est optionnelle. Si elle est omise, la liste des colonnes sera par défaut la liste de l'ensemble des colonnes de la table dans l'ordre de la création de la table. Si une liste de colonnes est spécifiée, les colonnes ne figurant pas dans la liste auront la valeur NULL.

Il est possible d'insérer dans une table des lignes provenant d'une autre table. La syntaxe est la suivante :

Sélectionnez

```
INSERT INTO nom_table(nom_col1, nom_col2, ...)
SELECT ...
```

Le SELECT (cf. section 4.4Interroger une base (LMD) : SELECT (1re partie) et 4.5Interroger une base (LMD) : SELECT (2e partie)) peut contenir n'importe quelle clause sauf un ORDER BY (cf. section 4.4.6La clause ORDER BY).

4-3-2. Modification de n-uplets : UPDATE▲

La commande UPDATE permet de modifier les valeurs d'une ou plusieurs colonnes, dans une ou plusieurs lignes existantes d'une table. La syntaxe est la suivante :

Sélectionnez

```
UPDATE nom_table
SET nom_col_1 = {expression_1 | ( SELECT ...) },
    nom_col_2 = {expression_2 | ( SELECT ...) },
    ...
    nom_col_n = {expression_n | ( SELECT ...) }
WHERE predicat
```

Les valeurs des colonnes nom_col_1, nom_col_2,… nom_col_n sont modifiées dans toutes les lignes qui satisfont le prédicat predicat. En l'absence d'une clause WHERE, toutes les lignes sont mises à jour. Les expressions expression_1, expression_2,… expression_n peuvent faire référence aux anciennes valeurs de la ligne.

4-3-3. Suppression de n-uplets : DELETE▲

La commande DELETE permet de supprimer des lignes d'une table.

La syntaxe est la suivante :

Sélectionnez

```
DELETE FROM nom_table
WHERE predicat
```

Toutes les lignes pour lesquelles predicat est évalué à vrai sont supprimées. En l'absence de clause WHERE, toutes les lignes de la table sont supprimées.

4-4. Interroger une base (LMD) : SELECT (1re partie)▲

4-4-1. Introduction à la commande SELECT▲

4-4-1-a. Introduction▲

La commande SELECT constitue, à elle seule, le langage permettant d'interroger une base de données. Elle permet de :

- sélectionner certaines colonnes d'une table (projection) ;
- sélectionner certaines lignes d'une table en fonction de leur contenu (sélection) ;
- combiner des informations venant de plusieurs tables (jointure, union, intersection, différence et division) ;
- combiner entre elles ces différentes opérations.

Une requête (i.e. une interrogation) est une combinaison d'opérations portant sur des tables (relations) et dont le résultat est lui-même une table dont l'existence est éphémère (le temps de la requête).

4-4-1-b. Syntaxe simplifiée de la commande SELECT▲

Une requête se présente généralement sous la forme :

Sélectionnez

```
SELECT  [ ALL | DISTINCT ] { * | attribut [, ...] }
    FROM nom_table [, ...]
    [ WHERE condition ]
```

- la clause SELECT permet de spécifier les attributs que l'on désire voir apparaître dans le résultat de la requête ; le caractère étoile (*) récupère tous les attributs de la table générée par la clause FROM de la requête ;
- la clause FROM spécifie les tables sur lesquelles porte la requête ;
- la clause WHERE, qui est facultative, énonce une condition que doivent respecter les n-uplets sélectionnés.

Par exemple, pour afficher l'ensemble des n-uplets de la table film, vous pouvez utiliser la requête :

Sélectionnez

```
SELECT * FROM film
```

De manière synthétique, on peut dire que la clause SELECT permet de réaliser la projection, la clause FROM le produit cartésien et la clause WHERE la sélection (cf. section 4.4.2Traduction des opérateurs de l'algèbre relationnelle (1 ère partie)).

4-4-1-c. Délimiteurs : apostrophes simples et doubles▲

Pour spécifier littéralement une chaîne de caractères, il faut l'entourer d'apostrophes (i.e. guillemets simples). Par exemple, pour sélectionner les films policiers, on utilise la requête :

Sélectionnez

```
SELECT * FROM film WHERE genre='Policier'
```

Les dates doivent également être entourées d'apostrophes (ex. : '01/01/2005').

Comme l'apostrophe est utilisée pour délimiter les chaînes de caractères, pour la représenter dans une chaîne, il faut la dédoubler (exemple : 'l''arbre'), ou la faire précéder d'un antislash (exemple : 'l\'arbre').

Lorsque le nom d'un élément d'une base de données (un nom de table ou de colonne par exemple) est identique à un mot-clé du SQL, il convient de l'entourer d'apostrophes doubles. Par exemple, si la table achat possède un attribut date, on pourra écrire :

Sélectionnez

```
SELECT ''date'' FROM achat
```

Bien entendu, les mots réservés du SQL sont déconseillés pour nommer de tels objets. Les apostrophes doubles sont également nécessaires lorsque le nom (d'une colonne ou d'une table) est composé de caractères particuliers tels que les blancs ou autres, ce qui est évidemment déconseillé.

4-4-2. Traduction des opérateurs de l'algèbre relationnelle (1ère partie)▲

4-4-2-a. Traduction de l'opérateur de projection▲

L'opérateur de projection $\Pi_{(A1,\dots An)}$(relation) se traduit tout simplement en SQL par la requête :

Sélectionnez

```
SELECT DISTINCT A_1, ..., A_n FROM relation
```

DISTINCT permet de ne retenir qu'une occurrence de n-uplet dans le cas où une requête produit plusieurs n-uplets identiques (cf. section 4.4.4La clause SELECT).

4-4-2-b. Traduction de l'opérateur de sélection▲

L'opérateur de sélection $\sigma_{(prédicat)}$(relation) se traduit tout simplement en SQL par la requête :

Sélectionnez

```
SELECT * FROM relation WHERE prédicat
```

De manière simplifiée, un prédicat est une expression logique sur des comparaisons. Reportez-vous à la section 4.4.7La clause WHERE pour une description plus complète.

4-4-2-c. Traduction de l'opérateur de produit cartésien▲

L'opérateur de produit cartésien $relation_1 \times relation_2$ se traduit en SQL par la requête :

Sélectionnez

```
SELECT * FROM relation_1, relation_2
```

Nous reviendrons sur le produit cartésien dans les sections 4.4.5La clause FROM (1re partie) et 4.5.1La clause FROM (2e partie) : les jointures.

4-4-2-d. Traduction de l'opérateur d'équijointure▲

L'opérateur d'équijointure relation1 $\triangleright\triangleleft_{A_1,A_2}$ relation2 se traduit en SQL par la requête :

Sélectionnez

```
SELECT * FROM relation_1, relation_2 WHERE relation_1.A_1 = relation_2.A_2
```

Nous reviendrons sur les différents types de jointure dans la section 4.5.1La clause FROM (2e partie) : les jointures.

4-4-3. Syntaxe générale de la commande SELECT▲

Voici la syntaxe générale d'une commande SELECT :

Sélectionnez

```
SELECT [ ALL | DISTINCT ] { * | expression [ AS nom_affiché ] } [, ...]
    FROM nom_table [ [ AS ] alias ] [, ...]
    [ WHERE prédicat ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ {UNION | INTERSECT | EXCEPT [ALL]} requête ]
    [ ORDER BY expression [ ASC | DESC ] [, ...] ]
```

En fait l'ordre SQL SELECT est composé de 7 clauses dont 5 sont optionnelles :

SELECT :

- Cette clause permet de spécifier les attributs que l'on désire voir apparaître dans le résultat de la requête (cf. section 4.4.4La clause SELECT).

FROM :

- Cette clause spécifie les tables sur lesquelles porte la requête (cf. section 4.4.5La clause FROM (1re partie) et 4.5.1La clause FROM (2e partie) : les jointures).

WHERE :

- Cette clause permet de filtrer les n-uplets en imposant une condition à remplir pour qu'ils soient présents dans le résultat de la requête (cf. section 4.4.7La clause WHERE).

GROUP BY :

- Cette clause permet de définir des groupes (i.e. sous-ensemble ; cf. section 4.5.2Les clauses GROUP BY et HAVING et les fonctions d'agrégation).

HAVING :

- Cette clause permet de spécifier un filtre (condition de regroupement des n-uplets) portant sur les résultats (cf. section 4.5.2Les clauses GROUP BY et HAVING et les fonctions d'agrégation).

UNION**,** INTERSECT **et** EXCEPT :

- Cette clause permet d'effectuer des opérations ensemblistes entre plusieurs résultats de requête (i.e. entre plusieurs SELECT) (cf. section 4.5.3Opérateurs ensemblistes : UNION, INTERSECT et EXCEPT).

ORDER BY :

- Cette clause permet de trier les n-uplets du résultat (cf. section 4.4.6La clause ORDER BY).

4-4-4. La clause SELECT▲

4-4-4-a. Introduction▲

Comme nous l'avons déjà dit, la clause SELECT permet de spécifier les attributs que l'on désire voir apparaître dans le résultat de la requête. Pour préciser explicitement les attributs que l'on désire conserver, il faut les lister en les séparant par une virgule. Cela revient en fait à opérer une projection de la table intermédiaire générée par le reste de la requête. Nous verrons dans cette section que la clause SELECT permet d'aller plus loin que la simple opération de projection. En effet, cette clause permet également de renommer des colonnes, voire d'en créer de nouvelles à partir des colonnes existantes.

Pour illustrer par des exemples les sections qui suivent, nous utiliserons une table dont le schéma est le suivant :

Sélectionnez

```
employee(id_employee, surname, name, salary)
```

Cette table contient respectivement l'identifiant, le nom, le prénom et le salaire mensuel des employés d'une compagnie.

4-4-4-b. L'opérateur étoile (*)▲

Le caractère étoile (*) permet de récupérer automatiquement tous les attributs de la table générée par la clause FROM de la requête.

Pour afficher la table employee on peut utiliser la requête :

Sélectionnez

```
SELECT * FROM employee
```

4-4-4-c. Les opérateurs DISTINCT et ALL▲

Lorsque le SGBD construit la réponse d'une requête, il rapatrie toutes les lignes qui satisfont la requête, généralement dans l'ordre ou il les trouve, même si ces dernières sont en double (comportement ALL par défaut). C'est pourquoi il est souvent nécessaire d'utiliser le mot-clé DISTINCT qui permet d'éliminer les doublons dans la réponse.

Par exemple, pour afficher la liste des prénoms, sans doublon, des employés de la compagnie, il faut utiliser la requête :

Sélectionnez

```
SELECT DISTINCT name FROM employee
```

4-4-4-d. Les opérations mathématiques de base▲

Il est possible d'utiliser les opérateurs mathématiques de base (i.e. +, -, * et /) pour générer de nouvelles colonnes à partir, en général, d'une ou plusieurs colonnes existantes.

Pour afficher le nom, le prénom et le salaire annuel des employés, on peut utiliser la requête :

Sélectionnez

```
SELECT surname, name, salary*12 FROM employee
```

4-4-4-e. L'opérateur AS▲

Le mot-clé AS permet de renommer une colonne, ou de nommer une colonne créée dans la requête.

Pour afficher le nom, le prénom et le salaire annuel des employés, on peut utiliser la requête :

Sélectionnez

```
SELECT surname AS nom, name AS prénom, salary*12 AS salaire FROM employee
```

4-4-4-f. L'opérateur de concaténation▲

L'opérateur || (double barre verticale) permet de concaténer des champs de type caractères.

Pour afficher le nom et le prénom sur une colonne, puis le salaire annuel des employés, on peut utiliser la requête :

Sélectionnez

```
SELECT surname || ' ' || name AS nom, salary*12 AS salaire FROM employee
```

4-4-5. La clause FROM (1re partie)▲

4-4-5-a. Comportement▲

Comme nous l'avons déjà dit, la clause FROM spécifie les tables sur lesquelles porte la requête. Plus exactement, cette clause construit la table intermédiaire (i.e. virtuelle), à partir d'une ou de plusieurs tables, sur laquelle des modifications seront apportées par les clauses SELECT, WHERE, GROUP BY et HAVING pour générer la table finale résultat de la requête. Quand plusieurs tables, séparées par des virgules, sont énumérées dans la clause FROM, la table intermédiaire est le résultat du produit cartésien de toutes les tables énumérées.

4-4-5-b. L'opérateur AS▲

Le mot-clé AS permet de renommer une table, ou de nommer une table créée dans la requête (c'est-à-dire une sous-requête) afin de pouvoir ensuite y faire référence. Le renommage du nom d'une table se fait de l'une des deux manières suivantes :

Sélectionnez

```
FROM nom_de_table AS nouveau_nom
FROM nom_de_table nouveau_nom
```

Une application typique du renommage de table est de simplifier les noms trop long :

Sélectionnez

```
SELECT * FROM nom_de_table_1 AS t1, nom_de_table_1 AS t2 WHERE t1.A_1 = t2.A_2
```

Attention, le nouveau nom remplace complètement l'ancien nom de la table dans la requête. Ainsi, quand une table a été renommée, il n'est plus possible d'y faire référence en utilisant son ancien nom. La requête suivante n'est donc pas valide :

Sélectionnez

```
SELECT * FROM nom_table AS t WHERE nom_table.a > 5
```

4-4-5-c. Sous-requête▲

Les tables mentionnées dans la clause FROM peuvent très bien correspondre à des tables résultant d'une requête, spécifiée entre parenthèses, plutôt qu'à des tables existantes dans la base de données. Il faut toujours nommer les tables correspondant à des sous-requêtes en utilisant l'opérateur AS.

Par exemple, les deux requêtes suivantes sont équivalentes :

Sélectionnez

```
SELECT * FROM table_1, table_2
SELECT * FROM (SELECT * FROM table_1) AS t1, table_2
```

4-4-5-d. Les jointures▲

Nous traiterons cet aspect de la clause FROM dans la section 4.5.1La clause FROM (2e partie) : les jointures.

4-4-6. La clause ORDER BY▲

Comme nous l'avons déjà dit, la clause ORDER BY permet de trier les n-uplets du résultat et sa syntaxe est la suivante :

Sélectionnez

```
ORDER BY expression [ ASC | DESC ] [, ...]
```

expression désigne soit une colonne, soit une opération mathématique de base (nous avons abordé ce type d'opérations dans la section 4.4.4La clause SELECT sur « La clause SELECT ») sur les colonnes.

ASC spécifie l'ordre ascendant et DESC l'ordre descendant du tri. En l'absence de précision ASC ou DESC, c'est l'ordre ascendant qui est utilisé par défaut.

Quand plusieurs expressions, ou colonnes sont mentionnées, le tri se fait d'abord selon les premières, puis suivant les suivantes pour les n-uplet qui sont égaux selon les premières.

Le tri est un tri interne sur le résultat final de la requête, il ne faut donc placer dans cette clause que les noms des colonnes mentionnés dans la clause SELECT.

La clause ORDER BY permet de trier le résultat final de la requête, elle est donc la dernière clause de tout ordre SQL et ne doit figurer qu'une seule fois dans le SELECT, même s'il existe des requêtes imbriquées ou un jeu de requêtes ensemblistes (cf. section 4.5.3Opérateurs ensemblistes : UNION, INTERSECT et EXCEPT).

En l'absence de clause ORDER BY, l'ordre des n-uplet est aléatoire et non garanti. Souvent, le fait de placer le mot-clé DISTINCT suffit à établir un tri puisque le SGBD doit se livrer à une comparaison des lignes, mais ce mécanisme n'est pas garanti, car ce tri s'effectue dans un ordre non contrôlable qui peut varier d'un serveur à l'autre.

4-4-7. La clause WHERE▲

4-4-7-a. Comportement▲

Comme nous l'avons déjà dit, la clause WHERE permet de filtrer les n-uplets en imposant une condition à remplir pour qu'ils soient présents dans le résultat de la requête ; sa syntaxe est la suivante :

Sélectionnez

```
WHERE prédicat
```

Concrètement, après que la table intermédiaire (i.e. virtuelle) de la clause FROM a été construite, chaque ligne de la table est confrontée au prédicat prédicat afin de vérifier si la ligne satisfait (i.e. le prédicat est vrai pour cette ligne) ou ne satisfait pas (i.e. le prédicat est faux ou NULL pour cette ligne) le prédicat. Les lignes qui ne satisfont pas le prédicat sont supprimées de la table intermédiaire.

Le prédicat n'est rien d'autre qu'une expression logique. En principe, celle-ci fait intervenir une ou plusieurs lignes de la table générée par la clause FROM, cela n'est pas impératif, mais, dans le cas contraire, l'utilité de la clause WHERE serait nulle.

4-4-7-b. Expression simple▲

Une expression simple peut être une variable désignée par un nom de colonne ou une constante. Si la variable désigne un nom de colonne, la valeur de la variable sera la valeur située dans la table à l'intersection de la colonne et de la ligne dont le SGBD cherche à vérifier si elle satisfait le prédicat de la clause WHERE.

Les expressions simples peuvent être de trois types : numérique, chaîne de caractères ou date.

Une expression simple peut également être le résultat d'une sous-requête, spécifiée entre parenthèses, qui retourne une table ne contenant qu'une seule ligne et qu'une seule colonne (i.e. une sous-requête retournant une valeur unique).

4-4-7-c. Prédicat simple▲

Un prédicat simple peut être le résultat de la comparaison de deux expressions simples au moyen de l'un des opérateurs suivants :

=	égal

!=	différent
<	strictement inférieur
<=	inférieur ou égal
>	strictement supérieur
>=	supérieur ou égal

Dans ce cas, les trois types d'expressions (numérique, chaîne de caractères et date) peuvent être comparés. Pour les types date, la relation d'ordre est l'ordre chronologique. Pour les caractères, la relation d'ordre est l'ordre lexicographique.

Un prédicat simple peut également correspondre à un test de description d'une chaîne de caractères par une expression régulière :

~	décrit par l'expression régulière
~*	comme LIKE, mais sans tenir compte de la casse
!~	non décrit par l'expression régulière
!~*	comme NOT LIKE, mais sans tenir compte de la casse

Dans ce cas, la chaîne de caractères faisant l'objet du test est à gauche et correspond à une expression simple du type chaîne de caractères, il s'agit généralement d'un nom de colonne. L'expression régulière, qui s'écrit entre apostrophes simples, comme une chaîne de caractères, est située à droite de l'opérateur. La section 4.4.8Les expressions régulières donne une description détaillée du formalisme des expressions régulières.

Un prédicat simple peut enfin correspondre à l'un des tests suivants :

expr IS NULL	test sur l'indétermination de expr
expr IN (expr_1 [...])	comparaison de expr à une liste de valeurs
expr NOT IN (expr_1 [...])	test d'absence d'une liste de valeurs
expr IN (requête) expr NOT IN (requête)	même chose, mais la liste de valeurs est le résultat d'une sous-requête qui doit impérativement retourner une table ne contenant qu'une colonne
EXIST (requête)	vraie si la sous-requête retourne au moins un n-uplet
expr operateur ANY (requête)	vraie si au moins un n-uplet de la sous-requête vérifie la comparaison « expr opérateur n-uplet » ; la sous-requête doit impérativement retourner une table ne contenant qu'une colonne ; IN est équivalent à = ANY
expr operateur ALL (requête)	vraie si tous les n-uplets de la sous-requête vérifient la comparaison « expr opérateur n-uplet » ; la sous-requête doit impérativement retourner une table ne contenant qu'une colonne

Dans ce tableau, expr désigne une expression simple et requête une sous-requête.

4-4-7-d. Prédicat composé▲

Les prédicats simples peuvent être combinés au sein d'expressions logiques en utilisant les opérateurs logiques AND (et logique), OR (ou logique) et NOT (négation logique).

4-4-8. Les expressions régulières▲

4-4-8-a. Introduction▲

Le terme expression régulière est issu de la théorie informatique et fait référence à un ensemble de règles permettant de définir un ensemble de chaînes de caractères.

Une expression régulière constitue donc une manière compacte de définir un ensemble de chaînes de caractères. Nous dirons qu'une chaîne de caractères est décrite par une expression régulière si cette chaîne est un élément de l'ensemble de chaînes de caractères défini par l'expression régulière.

PostgreSQL dispose de trois opérateurs de description par une expression régulière :

1. LIKE ou ~~
2. ~
3. SIMILAR TO

La syntaxe et le pouvoir expressif des expressions régulières diffèrent pour ces trois opérateurs. Nous ne décrirons ici que la syntaxe du formalisme le plus standard et le plus puissant, celui que l'on retrouve sous Unix avec les commandes egrep, sed et awk. Ce formalisme est celui associé à l'opérateur ~.

Avec PostgreSQL, le test d'égalité avec une chaîne de caractères s'écrit :

Sélectionnez

```
expression='chaine'
```

De manière équivalente, le test de description par une expression régulière s'écrit :

Sélectionnez

```
expression~'expression_régulière'
```

L'opérateur de description ~ est sensible à la casse, l'opérateur de description insensible à la casse est ~*. L'opérateur de non-description sensible à la casse est !~, son équivalent insensible à la casse se note !~*.

4-4-8-b. Formalisme▲

Comme nous allons le voir, dans une expression régulière, certains symboles ont une signification spéciale. Dans ce qui suit, expreg, expreg_1, expreg_2 désignent des expressions régulières, caractère un caractère quelconque et liste_de_caractères une liste de caractères quelconque.

caractère :

- un caractère est une expression régulière qui désigne le caractère lui-même, excepté pour les caractères ., ?, +, *, {, |, (,), ^, $, \, [,]. Ces derniers sont des métacaractères et ont une signification spéciale. Pour désigner ces métacaractères, il faut les faire précéder d'un antislash (\., \?, \+, *, \{, \|, \(, \), \^, \$, \\, \[, \]).

[liste_de_caractères] :

- est une expression régulière qui décrit l'un des caractères de la liste de caractères, par exemple [abcdf] décrit le caractère a, le b, le c, le d ou le f ; le caractère - permet de décrire des ensembles de caractères consécutifs, par exemple [a-df] est équivalent à [abcdf] ; la plupart des métacaractères perdent leur signification spéciale dans une liste, pour insérer un] dans une liste, il faut le mettre en tête de liste, pour inclure un ^, il faut le mettre n'importe où sauf en tête de liste, enfin un - se place à la fin de la liste.

[^liste_de_caractères] :

- est une expression régulière qui décrit les caractères qui ne sont pas dans la liste de caractères.

[:alnum:] :

- à l'intérieur d'une liste, décrit un caractère alphanumérique ([[:alnum:]] est équivalent à [0-9A-Za-z]) ; sur le même principe, on a également [:alpha:], [:cntrl:], [:digit:], [:graph:], [:lower:], [:print:], [:punct:], [:space:], [:upper:] et [:xdigit:].

. :

- est une expression régulière et un métacaractère qui désignent n'importe quel caractère.

^ :

- est une expression régulière et un métacaractère qui désignent le début d'une chaîne de caractères.

$:

- est une expression régulière et un métacaractère qui désignent la fin d'une chaîne de caractères.

expreg? :

- est une expression régulière qui décrit zéro ou une fois expreg.

expreg* :

- est une expression régulière qui décrit expreg un nombre quelconque de fois, zéro compris.

expreg+ :

- est une expression régulière qui décrit expreg au moins une fois.

expreg{n} :

- est une expression régulière qui décrit expreg n fois.

expreg{n,} :

- est une expression régulière qui décrit expreg au moins n fois.

expreg{n,m} :

- décrit expreg au moins n fois et au plus m fois.

expreg_1expreg_2 :

- est une expression régulière qui décrit une chaîne constituée de la concaténation de deux sous-chaînes respectivement décrites par expreg_1 et expreg_2.

expreg_1|expreg_2 :

- est une expression régulière qui décrit toute chaîne décrite par expreg_1 ou par expreg_2.

(expreg) :

- est une expression régulière qui décrit ce que décrit expreg.

n :

- où n est un chiffre, est une expression régulière qui décrit la sous-chaîne décrite par la n ème sous-expression parenthèsée de l'expression régulière.

la concaténation de deux expressions régulières (expreg_1expreg_2) est une opération prioritaire sur l'union (expreg_1|expreg_2).

4-4-8-c. Exemples▲

Un caractère, qui n'est pas un métacaractère, se décrit lui-même. Ce qui signifie que si vous cherchez une chaîne qui contient « voiture », vous devez utiliser l'expression régulière 'voiture'.

Si vous ne cherchez que les motifs situés en début de ligne, utilisez le symbole ^. Pour chercher toutes les chaînes qui commencent par « voiture », utilisez '^voiture'.

Le signe $ (dollar) indique que vous souhaitez trouver les motifs en fin de ligne. Ainsi : 'voiture$' permet de trouver toutes les chaînes finissant par « voiture ».

Le symbole . (point) remplace n'importe quel caractère. Pour trouver toutes les occurrences du motif composé des lettres vo, de trois lettres quelconques, et de la lettre e, utilisez : 'vo…e'. Cette commande permet de trouver des chaînes comme : voyagent, voyage, voyager, voyageur, vous_e.

Vous pouvez aussi définir un ensemble de lettres en les insérant entre crochets [ ]. Pour chercher toutes les chaînes qui contiennent les lettres P ou p suivies de rince, utilisez :'[Pp]rince'.

Si vous voulez spécifier un intervalle de caractères, servez-vous d'un trait d'union pour délimiter le début et la fin de l'intervalle. Vous pouvez aussi définir plusieurs intervalles simultanément. Par exemple [A-Za-z] désigne toutes les lettres de l'alphabet, hormis les caractères accentués, quelle que soit la casse. Notez bien qu'un intervalle ne correspond qu'à un caractère dans le texte.

Le symbole * est utilisé pour définir zéro ou plusieurs occurrences du motif précédent. Par exemple, l'expression régulière '^Pa(pa)*$' décrit les chaînes : Pa, Papa, Papapa, Papapapapapapa…

Si vous souhaitez qu'un symbole soit interprété littéralement, il faut le préfixer par un \. Pour trouver toutes les lignes qui contiennent le symbole $, utilisez : \$

4-5. Interroger une base (LMD) : SELECT (2e partie)▲

4-5-1. La clause FROM (2e partie) : les jointures▲

4-5-1-a. Recommandation▲

Dans la mesure du possible, et contrairement à ce que nous avons fait jusqu'à présent, il est préférable d'utiliser un opérateur de jointure normalisé SQL2 (mot-clé JOIN) pour effectuer une jointure. En effet, les jointures faites dans la clause WHERE (ancienne syntaxe datant de 1986) ne permettent pas de faire la distinction, de prime abord, entre ce qui relève de la sélection et ce qui relève de la jointure puisque tout est regroupé dans une seule clause (la clause WHERE). La lisibilité des requêtes est plus grande en utilisant la syntaxe de l'opérateur JOIN qui permet d'isoler les conditions de sélections (clause WHERE) de celles de jointures (clauses JOIN), et qui permet également de cloisonner les conditions de jointures entre chaque couple de tables. De plus, l'optimisation d'exécution de la requête est souvent plus pointue lorsque l'on utilise l'opérateur JOIN. Enfin, lorsque l'on utilise l'ancienne syntaxe, la suppression de la clause WHERE à des fins de tests pose évidemment des problèmes.

4-5-1-b. Le produit cartésien▲

Prenons une opération de jointure entre deux tables R_1 et R_2 selon une expression logique E. En algèbre relationnelle, cette opération se note : $R_1 \triangleright\triangleleft E\ R_2$

Dans la section 3.3.8Jointure, thêta-jointure, équijointure, jointure naturelle, nous avons vu que la jointure n'est rien d'autre qu'un produit cartésien suivi d'une sélection : $R_1 \triangleright\triangleleft E\ R_2 = \sigma E\ (R_1 \times R_2)$

On peut également dire que le produit cartésien n'est rien d'autre qu'une jointure dans laquelle l'expression logique E est toujours vraie : $R_1 \times R_2 = R_1 \triangleright\triangleleft \text{true}\ R_2$

Nous avons vu section 4.4.5La clause FROM (1re partie) que le produit cartésien entre deux tables table_1 et table_2 peut s'écrire en SQL :

Sélectionnez

```
SELECT * FROM table_1, table_2
```

Il peut également s'écrire en utilisant le mot-clé JOIN dédié aux jointures de la manière suivante :

Sélectionnez

```
SELECT * FROM table_1 CROSS JOIN table_2
```

En fait, sous PostgreSQL, les quatre écritures suivantes sont équivalentes :

Sélectionnez

```
SELECT * FROM table_1, table_2
SELECT * FROM table_1 CROSS JOIN table_2
```

```
SELECT * FROM table_1 JOIN table_2 ON TRUE
SELECT * FROM table_1 INNER JOIN table_2 ON TRUE
```

Les deux dernières écritures prendront un sens dans les sections qui suivent.

4-5-1-c. Syntaxe générale des jointures▲

Sans compter l'opérateur CROSS JOIN, voici les trois syntaxes possibles de l'expression d'une jointure dans la clause FROM en SQL :

Sélectionnez

```
table_1         [ INNER | { { LEFT | RIGHT | FULL } [OUTER] } ] JOIN table_2 ON predicat [..
table_1         [ INNER | { { LEFT | RIGHT | FULL } [OUTER] } ] JOIN table_2 USING (colonnes
table_1 NATURAL [ INNER | { { LEFT | RIGHT | FULL } [OUTER] } ] JOIN table_2 [...]
```

Ces trois syntaxes diffèrent par la condition de jointure spécifiée par les clause ON ou USING, ou implicite dans le cas d'une jointure naturelle introduite par le mot-clé NATURAL.

ON :

- La clause ON correspond à la condition de jointure la plus générale. Le prédicat predicat est une expression logique de la même nature que celle de la clause WHERE décrite dans la section 4.4.7La clause WHERE.

USING :

- La clause USING est une notation correspondant à un cas particulier de la clause ON qui, de plus supprime les colonnes superflues. Les deux tables, sur lesquelles porte la jointure, doivent posséder toutes les colonnes qui sont mentionnées, en les séparant par des virgules, dans la liste spécifiée entre parenthèses juste après le mot-clé USING. La condition de jointure sera l'égalité des colonnes au sein de chacune des paires de colonnes. De plus, les paires de colonnes seront fusionnées en une colonne unique dans la table résultat de la jointure. Par rapport à une jointure classique, la table résultat comportera autant de colonnes de moins que de colonnes spécifiées dans la liste de la clause USING.

NATURAL :

- Il s'agit d'une notation abrégée de la clause USING dans laquelle la liste de colonnes est implicite et correspond à la liste des colonnes communes aux deux tables participant à la jointure. Tout comme dans le cas de la clause USING, les colonnes communes n'apparaissent qu'une fois dans la table résultat.

INNER **et** OUTER :

- Les mots-clés INNER et OUTER permettent de préciser s'il s'agit d'une jointure interne ou externe. INNER et OUTER sont toujours optionnels. En effet, le comportement par défaut est celui de la jointure interne (INNER) et les mots-clés LEFT, RIGHT et FULL impliquent forcément une jointure externe (OUTER).

INNER JOIN :

- La table résultat est constituée de toutes les juxtapositions possibles d'une ligne de la table table_1 avec une ligne de la table table_2 qui satisfont la condition de jointure.

LEFT OUTER JOIN :

- Dans un premier temps, une jointure interne (i.e. de type INNER JOIN) est effectuée. Ensuite, chacune des lignes de la table table_1 qui ne satisfait la condition de jointure avec aucune des lignes de la table table_2 (i.e. les lignes de table_1 qui n'apparaissent pas dans la table résultat de la jointure interne) est ajoutée à la table résultats. Les attributs correspondant à la table table_2, pour cette ligne, sont affectés de la valeur NULL. Ainsi, la table résultat contient au moins autant de lignes que la table table_1.

RIGHT OUTER JOIN :

- Même scénario que pour l'opération de jointure de type LEFT OUTER JOIN, mais en inversant les rôles des tables table_1 et table_2.

FULL OUTER JOIN :

La jointure externe bilatérale est la combinaison des deux opérations précédentes (LEFT OUTER JOIN et RIGHT OUTER JOIN) afin que la table résultat contienne au moins une occurrence de chacune des lignes des deux tables impliquées dans l'opération de jointure.

La jointure externe droite peut être obtenue par une jointure externe gauche dans laquelle on inverse l'ordre des tables (et vice-versa). La jointure externe bilatérale peut être obtenue par la combinaison de deux jointures externes unilatérales avec l'opérateur ensembliste UNION que nous verrons dans la section 4.5.3Opérateurs ensemblistes : UNION, INTERSECT et EXCEPT.

Des jointures de n'importe quel type peuvent être chaînées les unes derrière les autres. Les jointures peuvent également être imbriquées étant donné que les tables table_1 et table_2 peuvent très bien être elles-mêmes le résultat de jointures de n'importe quel type. Les opérations de jointures peuvent être parenthésées afin de préciser l'ordre dans lequel elles sont effectuées. En l'absence de parenthèses, les jointures s'effectuent de gauche à droite.

4-5-1-d. Définition de deux tables pour les exemples qui suivent▲

Afin d'illustrer les opérations de jointure, considérons les tables realisateur et film définies de la manière suivante :

Sélectionnez

```
create table realisateur (
    id_real integer primary key,
    nom varchar(16),
    prenom varchar(16)
);
create table film (
    num_film integer primary key,
    id_real integer,
    titre varchar(32)
);
```

On notera que dans la table film, l'attribut id_real correspond à une clé étrangère et aurait dû être défini de la manière suivante : id_real integer references realisateur. Nous ne l'avons pas fait dans le but d'introduire des films dont le réalisateur n'existe pas dans la table realisateur afin d'illustrer les différentes facettes des opérations de jointure.

La table realisateur contient les lignes suivantes :

Sélectionnez

```
 id_real |    nom    | prenom
---------+-----------+---------
       1 | von Trier | Lars
       4 | Tarantino | Quentin
       3 | Eastwood  | Clint
       2 | Parker    | Alan
```

La table film contient les lignes suivantes :

Sélectionnez

```
 id_film | id_real |           titre
---------+---------+----------------------------
       1 |       1 | Dogville
       2 |       1 | Breaking the waves
       3 |       5 | Faux-Semblants
       4 |       5 | Crash
       5 |       3 | Chasseur blanc, cœur noir
Exemples de jointures internes
```

La **jointure naturelle** entre les tables film et réalisateur peut s'écrire indifféremment de l'une des manières suivantes :

Sélectionnez

```
SELECT * FROM film NATURAL JOIN realisateur
SELECT * FROM film NATURAL INNER JOIN realisateur;
SELECT * FROM film JOIN realisateur USING (id_real);
SELECT * FROM film INNER JOIN realisateur USING (id_real);
```

pour produire le résultat suivant :

Sélectionnez

```
 id_real | id_film |           titre            |    nom    | prenom
---------+---------+----------------------------+-----------+--------
       1 |       1 | Dogville                   | von Trier | Lars
       1 |       2 | Breaking the waves         | von Trier | Lars
       3 |       5 | Chasseur blanc, cœur noir   | Eastwood  | Clint
```

Nous aurions également pu effectuer une **équijointure** en écrivant :

Sélectionnez

```
SELECT * FROM film, realisateur WHERE film.id_real = realisateur.id_real;
SELECT * FROM film JOIN realisateur ON film.id_real = realisateur.id_real;
SELECT * FROM film INNER JOIN realisateur ON film.id_real = realisateur.id_real;
```

Mais la colonne id_real aurait été dupliquée :

Sélectionnez

```
 id_film | id_real |           titre            | id_real |    nom    | prenom
---------+---------+----------------------------+---------+-----------+--------
       1 |       1 | Dogville                   |       1 | von Trier | Lars
       2 |       1 | Breaking the waves         |       1 | von Trier | Lars
       5 |       3 | Chasseur blanc, cœur noir   |       3 | Eastwood  | Clint
```

4-5-1-e. Exemples de jointures externes gauches▲

La jointure externe gauche entre les tables film et réalisateur permet de conserver, dans la table résultat, une trace des films dont le réalisateur n'apparaît pas dans la table realisateur. Une telle jointure peut s'écrire indifféremment comme suit :

Sélectionnez

```
SELECT * FROM film NATURAL LEFT JOIN realisateur;
SELECT * FROM film NATURAL LEFT OUTER JOIN realisateur;
SELECT * FROM film LEFT JOIN realisateur USING (id_real);
SELECT * FROM film LEFT OUTER JOIN realisateur USING (id_real);
```

Elle produit le résultat suivant :

Sélectionnez

```
 id_real | id_film |           titre            |    nom    | prenom
---------+---------+----------------------------+-----------+--------
       1 |       1 | Dogville                   | von Trier | Lars
       1 |       2 | Breaking the waves         | von Trier | Lars
       5 |       3 | Faux-Semblants             |           |
       5 |       4 | Crash                      |           |
       3 |       5 | Chasseur blanc, cœur noir | Eastwood  | Clint
```

Naturellement, en écrivant :

Sélectionnez

```
SELECT * FROM film LEFT JOIN realisateur ON film.id_real = realisateur.id_real;
SELECT * FROM film LEFT OUTER JOIN realisateur ON film.id_real = realisateur.id_real;
```

la colonne id_real serait dupliquée :

Sélectionnez

```
 id_film | id_real |           titre            | id_real |    nom    | prenom
---------+---------+----------------------------+---------+-----------+--------
       1 |       1 | Dogville                   |       1 | von Trier | Lars
       2 |       1 | Breaking the waves         |       1 | von Trier | Lars
       3 |       5 | Faux-Semblants             |         |           |
       4 |       5 | Crash                      |         |           |
       5 |       3 | Chasseur blanc, cœur noir  |       3 | Eastwood  | Clint
```

4-5-1-f. Exemples de jointures externes droites▲

La jointure externe droite entre les tables film et réalisateur permet de conserver, dans la table résultat, une trace des réalisateurs dont aucun film n'apparaît dans la table film. Une telle jointure peut s'écrire indifféremment comme suit :

Sélectionnez

```
SELECT * FROM film NATURAL RIGHT JOIN realisateur;
SELECT * FROM film NATURAL RIGHT OUTER JOIN realisateur;
SELECT * FROM film RIGHT JOIN realisateur USING (id_real);
SELECT * FROM film RIGHT OUTER JOIN realisateur USING (id_real);
```

Elle produit le résultat suivant :

Sélectionnez

```
 id_real | id_film |           titre            |    nom    | prenom
---------+---------+----------------------------+-----------+---------
       1 |       1 | Dogville                   | von Trier | Lars
       1 |       2 | Breaking the waves         | von Trier | Lars
       2 |         |                            | Parker    | Alan
       3 |       5 | Chasseur blanc, cœur noir  | Eastwood  | Clint
       4 |         |                            | Tarantino | Quentin
```

4-5-1-g. Exemples de jointures externes bilatérales▲

La jointure externe bilatérale entre les tables film et réalisateur permet de conserver, dans la table résultat, une trace de tous les réalisateurs et de tous les films. Une telle jointure peut indifféremment s'écrire :

Sélectionnez

```
SELECT * FROM film NATURAL FULL JOIN realisateur;
SELECT * FROM film NATURAL FULL OUTER JOIN realisateur;
SELECT * FROM film FULL JOIN realisateur USING (id_real);
SELECT * FROM film FULL OUTER JOIN realisateur USING (id_real);
```

Elle produit le résultat suivant :

Sélectionnez

```
 id_real | id_film |           titre            |    nom    | prenom
---------+---------+----------------------------+-----------+---------
       1 |       1 | Dogville                   | von Trier | Lars
       1 |       2 | Breaking the waves         | von Trier | Lars
       2 |         |                            | Parker    | Alan
       3 |       5 | Chasseur blanc, cœur noir  | Eastwood  | Clint
       4 |         |                            | Tarantino | Quentin
       5 |       3 | Faux-Semblants             |           |
       5 |       4 | Crash                      |           |
```

4-5-2. Les clauses GROUP BY et HAVING et les fonctions d'agrégation▲

4-5-2-a. Notion de groupe▲

Un groupe est un sous-ensemble des lignes d'une table ayant la même valeur pour un attribut. Par exemple, on peut grouper les films en fonction de leur réalisateur. Un groupe est déterminé par la clause GROUP BY suivie du nom du ou des attributs sur lesquels s'effectue le regroupement.

4-5-2-b. Syntaxe▲

La syntaxe d'une requête faisant éventuellement intervenir des fonctions d'agrégation, une clause GROUP BY et une clause HAVING est la suivante :

Sélectionnez

```
SELECT expression_1, [...,] expression_N [, fonction_agrégation [, ...] ]
  FROM nom_table [ [ AS ] alias ] [, ...]
  [ WHERE prédicat ]
  [ GROUP BY expression_1, [...,] expression_N ]
  [ HAVING condition_regroupement ]
```

4-5-2-c. La clause GROUP BY▲

La commande GROUP BY permet de définir des regroupements (i.e. des agrégats) qui sont projetés dans la table résultat (un regroupement correspond à une ligne) et d'effectuer des calculs statistiques, définis par les expressions fonction_agrégation […], pour chacun des regroupements. La liste d'expressions expression_1, […,] expression_N correspond généralement à une liste de colonnes colonne_1, […,] colonne_N. La liste de colonnes spécifiée derrière la commande SELECT (expression_1, […,] expression_N) doit être identique à la liste de colonnes de regroupement spécifiée derrière la commande GROUP BY (expression_1, […,] expression_N). À la place des noms de colonne, il est possible de spécifier des opérations mathématiques de base sur les colonnes (comme définies dans la section 4.4.4La clause SELECT). Dans ce cas, les regroupements doivent porter sur les mêmes expressions.

Si les regroupements sont effectués selon une expression unique, les groupes sont définis par les ensembles de lignes pour lesquelles cette expression prend la même valeur. Si plusieurs expressions sont spécifiées (expression_1, expression_2…) les groupes sont définis de la façon suivante : parmi toutes les lignes pour lesquelles expression_1 prend la même valeur, on regroupe celles ayant expression_2 identique, etc.

Un SELECT avec une clause GROUP BY produit une table résultat comportant une ligne pour chaque groupe.

4-5-2-d. Les fonctions d'agrégation▲

AVG([DISTINCT | ALL] expression) :

- Calcule la moyenne des valeurs de l'expression expression.

COUNT(* | [DISTINCT | ALL] expression) :

- Dénombre le nombre de lignes du groupe. Si expression est présent, on ne compte que les lignes pour lesquelles cette expression n'est pas NULL.

MAX([DISTINCT | ALL] expression) :

- Retourne la plus petite des valeurs de l'expression expression.

MIN([DISTINCT | ALL] expression) :

- Retourne la plus grande des valeurs de l'expression expression.

STDDEV([DISTINCT | ALL] expression) :

- Calcule l'écart-type des valeurs de l'expression expression.

SUM([DISTINCT | ALL] expression) :

- Calcule la somme des valeurs de l'expression expression.

VARIANCE([DISTINCT | ALL] expression) :

- Calcule la variance des valeurs de l'expression expression.

DISTINCT indique à la fonction de groupe de ne prendre en compte que des valeurs distinctes. ALL indique à la fonction de groupe de prendre en compte toutes les valeurs, c'est la valeur par défaut.

Aucune des fonctions de groupe ne tient compte des valeurs NULL à l'exception de COUNT(*). Ainsi, SUM(col) est la somme des valeurs non NULL de la colonne col. De même AVG est la somme des valeurs non NULL divisée par le nombre de valeurs non NULL.

Il est tout à fait possible d'utiliser des fonctions d'agrégation sans clause GROUP BY. Dans ce cas, la clause SELECT ne doit comporter que des fonctions d'agrégation et aucun nom de colonne. Le résultat d'une telle requête ne contient qu'une ligne.

4-5-2-e. Exemples▲

Soit la base de données sur les films et les cinémas suivante :

- film (num_film, num_realisateur, titre, genre, annee)
- cinema (num_cinema, nom, adresse)
- individu (num_individu, nom prenom)
- jouer (num_acteur, num_film, role)
- projection (num_cinema, num_film, jour)

Pour connaître le nombre de fois que chacun des films a été projeté, on utilise la requête :

Sélectionnez

```
SELECT num_film, titre, COUNT(*)
  FROM film NATURAL JOIN projection
  GROUP BY num_film, titre;
```

Si l'on veut également connaître la date de la première et de la dernière projection, on utilise :

Sélectionnez

```
SELECT num_film, titre, COUNT(*), MIN(jour), MAX(jour)
  FROM film NATURAL JOIN projection
  GROUP BY num_film, titre;
```

Pour connaître le nombre total de films projetés au cinéma Le Fontenelle, ainsi que la date de la première et de la dernière projection dans ce cinéma, la requête ne contient pas de clause GROUP BY, mais elle contient des fonctions d'agrégation :

Sélectionnez

```
SELECT COUNT(*), MIN(jour), MAX(jour)
  FROM film NATURAL JOIN projection NATURAL JOIN cinema
  WHERE cinema.nom = 'Le Fontenelle';
```

4-5-2-f. La clause HAVING▲

De la même façon qu'il est possible de sélectionner certaines lignes au moyen de la clause WHERE, il est possible, dans un SELECT comportant une fonction de groupe, de sélectionner certains groupes par la clause HAVING. Celle-ci se place après la clause GROUP BY.

Le prédicat dans la clause HAVING suit les mêmes règles de syntaxe qu'un prédicat figurant dans une clause WHERE. Cependant, il ne peut porter que sur des caractéristiques du groupe : fonction d'agrégation ou expression figurant dans la clause GROUP BY.

Une requête de groupe (i.e. comportant une clause GROUP BY) peut contenir à la fois une clause WHERE et une clause HAVING. La clause WHERE sera d'abord appliquée pour sélectionner les lignes, puis les groupes seront constitués à partir des lignes sélectionnées, les fonctions de groupe seront ensuite évaluées et la clause HAVING sera enfin appliquée pour sélectionner les groupes.

4-5-2-g. Exemples▲

Pour connaître le nombre de fois que chacun des films a été projeté en ne s'intéressant qu'aux films projetés plus de 2 fois, on utilise la requête :

Sélectionnez

```
SELECT num_film, titre, COUNT(*)
  FROM film NATURAL JOIN projection
  GROUP BY num_film, titre
  HAVING COUNT(*)>2;
```

Si en plus, on ne s'intéresse qu'aux films projetés au cinéma Le Fontenelle, il faut ajouter une clause WHERE :

Sélectionnez

```
SELECT num_film, titre, COUNT(*)
  FROM film NATURAL JOIN projection NATURAL JOIN cinema
  WHERE cinema.nom = 'Le Fontenelle'
  GROUP BY num_film, titre
  HAVING COUNT(*)>2;
```

4-5-3. Opérateurs ensemblistes : UNION, INTERSECT et EXCEPT▲

Les résultats de deux requêtes peuvent être combinés en utilisant les opérateurs ensemblistes d'union (UNION), d'intersection (INTERSECT) et de différence (EXCEPT). La syntaxe d'une telle requête est la suivante :

Sélectionnez

```
requête_1 { UNION | INTERSECT | EXCEPT } [ALL] requête_2 [...]
```

Pour que l'opération ensembliste soit possible, il faut que requête_1 et requête_2 aient le même schéma, c'est-à-dire le même nombre de colonnes respectivement du même type. Les noms de colonnes (titres) sont ceux de la première requête (requête_1).

Il est tout à fait possible de chaîner plusieurs opérations ensemblistes. Dans ce cas, l'expression est évaluée de gauche à droite, mais on peut modifier l'ordre d'évaluation en utilisant des parenthèses.

Dans une requête on ne peut trouver qu'une seule instruction ORDER BY. Si elle est présente, elle doit être placée dans la dernière requête (cf. section 4.4.6La clause ORDER BY). La clause ORDER BY ne peut faire référence qu'aux numéros des colonnes (la première portant le numéro 1), et non pas à leurs noms, car les noms peuvent être différents dans chacune des requêtes sur lesquelles portent le ou les opérateurs ensemblistes.

Les opérateurs UNION et INTERSECT sont commutatifs.

Contrairement à la commande SELECT, le comportement par défaut des opérateurs ensemblistes élimine les doublons. Pour les conserver, il faut utiliser le mot-clé ALL.

Attention, il s'agit bien d'opérateurs portant sur des tables générées par des requêtes. On ne peut pas faire directement l'union de deux tables de la base de données.

4-5-4. Traduction des opérateurs de l'algèbre relationnelle (2e partie)▲

4-5-4-a. Traduction de l'opérateur d'union▲

L'opérateur d'union $relation_1 \cup relation_2$ se traduit tout simplement en SQL par la requête :

Sélectionnez

```
SELECT * FROM relation_1 UNION SELECT * FROM relation_2
```

4-5-4-b. Traduction de l'opérateur d'intersection▲

L'opérateur d'intersection $R_1 \cap R_2$ se traduit tout simplement en SQL par la requête :

Sélectionnez

```
SELECT * FROM relation_1 INTERSECT SELECT * FROM relation_2
```

4-5-4-c. Traduction de l'opérateur de différence▲

L'opérateur de différence $R_1 - R_2$ se traduit tout simplement en SQL par la requête :

Sélectionnez

```
SELECT * FROM relation_1 EXCEPT SELECT * FROM relation_2
```

4-5-4-d. Traduction de l'opérateur de division▲

Il n'existe pas de commande SQL permettant de réaliser directement une division. Prenons la requête :

> Quels sont les acteurs qui ont joué dans tous les films de Lars von Trier ?

Cela peut se reformuler par :

> Quels sont les acteurs qui vérifient : quel que soit un film de Lars von Trier, l'acteur a joué dans ce film.

Malheureusement, le quantificateur universel ($\forall$) n'existe pas en SQL. Par contre, le quantificateur existentiel ($\exists$) existe : EXISTS. Or, la logique des prédicats nous donne l'équivalence suivante : $\forall x P(x) = \neg \exists x \neg P(x)$

On peut donc reformuler le problème de la manière suivante :

> Quels sont les acteurs qui vérifient : il est faux qu'il existe un film de Lars von Trier dans lequel l'acteur n'a pas joué.

Ce qui correspond à la requête SQL :

Sélectionnez

```
SELECT DISTINCT nom, prenom FROM individu AS acteur_tous_lars
WHERE NOT EXISTS (
  SELECT * FROM ( film JOIN individu ON num_realisateur = num_individu
                  AND nom = 'von Trier' AND prenom = 'Lars' ) AS film_lars
  WHERE NOT EXISTS (
    SELECT * FROM individu JOIN jouer ON num_individu = num_acteur
          AND num_individu = acteur_tous_lars.num_individu
          AND num_film = film_lars.num_film
  )
);
```

En prenant le problème d'un autre point de vue, on peut le reformuler de la manière suivante :

> Quels sont les acteurs qui vérifient : le nombre de films réalisés par Lars von Trier dans lequel l'acteur a joué est égal au nombre de films réalisés par Lars von Trier.

Ce qui peut se traduire en SQL indifféremment par l'une des deux requêtes suivantes :

Sélectionnez

```
SELECT acteur.nom, acteur.prenom
FROM individu AS acteur JOIN jouer ON acteur.num_individu = jouer.num_acteur
    JOIN film ON jouer.num_film = film.num_film
    JOIN individu AS realisateur ON film.num_realisateur = realisateur.num_individu
WHERE realisateur.nom = 'von Trier' AND realisateur.prenom = 'Lars'
GROUP BY acteur.nom, acteur.prenom
HAVING COUNT (DISTINCT film.num_film) = (
  SELECT DISTINCT COUNT(*)
  FROM film JOIN individu ON num_realisateur = num_individu
  WHERE nom = 'von Trier' AND prenom = 'Lars'
);

SELECT DISTINCT acteur_tous_lars.nom, acteur_tous_lars.prenom
FROM individu AS acteur_tous_lars
WHERE (
  SELECT DISTINCT COUNT(*)
  FROM jouer JOIN film ON jouer.num_film = film.num_film
    JOIN individu ON num_realisateur = num_individu
  WHERE nom = 'von Trier' AND prenom = 'Lars'
    AND jouer.num_acteur = acteur_tous_lars.num_individu
```

```
) = (
  SELECT DISTINCT COUNT(*)
  FROM film JOIN individu ON num_realisateur = num_individu
  WHERE nom = 'von Trier' AND prenom = 'Lars'
);
```

4-6. Nouveaux objets - Langage de définition de données (LDD)▲

4-6-1. Séquences (CREATE SEQUENCE) et type SERIAL▲

4-6-1-a. Création d'une séquence▲

Une séquence est en fait une table spéciale contenant une seule ligne. Cet objet est utilisé pour créer une suite de nombres entiers dont l'évolution, généralement croissante, est régie par un certain nombre de paramètres.

Voici la syntaxe de création d'une séquence :

Sélectionnez

```
CREATE SEQUENCE nom [ INCREMENT [ BY ] incrément ]
    [ MINVALUE valeurmin ]
    [ MAXVALUE valeurmax ]
    [ START [ WITH ] début ]
    [ [ NO ] CYCLE ]
```

La commande CREATE SEQUENCE crée un nouveau générateur de nombre. Ceci implique la création et l'initialisation d'une nouvelle table portant le nom nom.

INCREMENT BY :

- La clause optionnelle INCREMENT BY incrément spécifie la valeur ajoutée à la valeur de la séquence courante pour créer une nouvelle valeur. Une valeur positive créera une séquence ascendante, une négative en créera une descendante. La valeur par défaut est 1.

MINVALUE :

- La clause optionnelle MINVALUE valeurmin précise la valeur minimale qu'une séquence peut générer. Si cette clause n'est pas fournie, alors les valeurs par défaut seront utilisées. Les valeurs par défaut sont 1 et −263−1 pour les séquences respectivement ascendantes et descendantes.

MAXVALUE :

- La clause optionnelle MAXVALUE valeurmax précise la valeur maximale pour la séquence. Si cette clause n'est pas fournie, alors les valeurs par défaut seront utilisées. Les valeurs par défaut sont 263−1 et −1 pour les séquences respectivement ascendantes et descendantes.

START WITH :

- La clause optionnelle START WITH début précise la valeur d'initialisation de la séquence. La valeur de début par défaut est valeurmin pour les séquences ascendantes et valeurmax pour les séquences descendantes.

[NO] CYCLE :

- L'option CYCLE autorise la séquence à recommencer au début lorsque valeurmax ou valeurmin a été atteinte par une séquence respectivement ascendante ou descendante. Si la limite est atteinte, le prochain nombre généré sera respectivement valeurmin ou valeurmax. Si NO CYCLE est spécifié, tout appel à nextval après que la séquence a atteint la valeur minimale renverra une erreur. NO CYCLE est le comportement par défaut.

4-6-1-b. Utilisation d'une séquence▲

Bien que vous ne pouvez pas mettre à jour directement une séquence, vous pouvez toujours utiliser une requête comme :

Sélectionnez

```
SELECT * FROM nom_sequence;
```

Après la création d'une séquence, il faut utiliser les fonctions nextval(), currval() et setval() pour la manipuler.

nextval('nom_sequence') :

- incrémente la valeur courante de la séquence nom_sequence (excepté la première fois) et retourne cette valeur.

currval('nom_sequence') :

- retourne la valeur courante de la séquence nom_sequence cette fonction ne peut être appelée que si nextval() l'a été au moins une fois.

setval('nom_sequence', nombre) :

- initialise la valeur courante de la séquence nom_sequence à nombre.

Vous pouvez appeler ces différentes fonctions de la manière suivante :

Sélectionnez

```
SELECT nextval('nom_sequence');
SELECT currval('nom_sequence');
SELECT setval('nom_sequence', nombre);
```

Utilisez DROP SEQUENCE pour supprimer une séquence.

4-6-1-c. Type SERIAL▲

Le type de donnée SERIAL n'est pas un vrai type, mais plutôt un raccourci de notation pour décrire une colonne d'identifiants uniques. Ainsi, la commande

Sélectionnez

```
CREATE TABLE nom_de_table (
    nom_de_colonne SERIAL
);
```

est équivalente à la commande :

Sélectionnez

```
CREATE SEQUENCE nom_de_sequence;
CREATE TABLE nom_de_table (
    nom_de_colonne integer DEFAULT nextval('nom_de_sequence') NOT NULL
);
```

Ainsi, nous avons créé une colonne d'entiers et fait en sorte que ses valeurs par défaut soient assignées par un générateur de séquence. Une contrainte NOT NULL est ajoutée pour s'assurer qu'une valeur nulle ne puisse pas être explicitement insérée. Dans la plupart des cas, on ajoute également une contrainte UNIQUE ou PRIMARY KEY pour interdire que des doublons soient créés par accident.

Pour insérer la valeur suivante de la séquence dans la colonne de type SERIAL, il faut faire en sorte d'utiliser la valeur par défaut de la colonne. Cela peut se faire de deux façons : soit en excluant cette colonne de la liste des colonnes de la commande INSERT, ou en utilisant le mot-clé DEFAULT.

4-6-2. Règles (CREATE RULE)▲

4-6-2-a. Description▲

Le système de règles autorise la définition d'actions alternatives à réaliser sur les insertions, mises à jour ou suppressions dans les tables de la base de données. Concrètement, une règle permet d'exécuter des commandes supplémentaires lorsqu'une commande donnée est exécutée sur une table donnée. Autrement dit, une règle peut remplacer une commande donnée par une autre ou faire qu'une commande ne soit pas exécutée. Les règles sont aussi utilisées pour implémenter les vues de tables (cf. section 4.6.3Vues (CREATE VIEW)).

4-6-2-b. Syntaxe de définition▲

Voici la syntaxe de création d'une règle :

Sélectionnez

```
CREATE [ OR REPLACE ] RULE nom AS
ON événement
TO table [ WHERE condition ]
DO [ INSTEAD ] { NOTHING | commande | ( commande ; commande ... ) }
```

CREATE RULE :

- définit une nouvelle règle s'appliquant à une table ou à une vue.

CREATE OR REPLACE RULE :

- définit une nouvelle règle, ou, le cas échéant, remplace une règle existante du même nom pour la même table.

nom :

- désigne le nom d'une règle à créer. Elle doit être distincte du nom de toute autre règle sur la même table. Lorsque plusieurs règles portent sur la même table et le même type d'événement, elles sont appliquées dans l'ordre alphabétique de leur nom.

événement :

- SELECT, INSERT, UPDATE ou DELETE. Les règles qui sont définies sur INSERT, UPDATE ou DELETE sont appelées des règles de mise à jour. Les règles définies sur SELECT permettent la création de vues (cf. section 4.6.3Vues (CREATE VIEW)).

table :

- Le nom (pouvant être qualifié par le nom du schéma) de la table ou de la vue où s'applique la règle.

condition :

- Toute expression SQL conditionnelle (i.e. de type boolean). L'expression de condition ne peut pas référer à une table autre que NEW et OLD et ne peut pas contenir de fonction d'agrégat.

commande :

- Zone de spécification des commandes réalisant l'action de la règle. Les commandes valides sont SELECT, INSERT, UPDATE, DELETE ou NOTIFY. Le mot-clé NOTHING permet de spécifier que l'on ne veut rien faire.

INSTEAD :

- Si ce mot-clé est utilisé, la ou les commandes sont exécutées à la place de la requête déclenchante. En l'absence de INSTEAD, la ou les commandes sont exécutées après la requête déclenchante dans le cas ON INSERT (pour permettre aux commandes de voir les lignes insérées) et avant dans le cas ON UPDATE ou ON DELETE (pour permettre aux commandes de voir les lignes à mettre à jour ou à supprimer).

À l'intérieur d'une condition et d'une commande, deux tables spéciales, NEW et OLD, peuvent être utilisées pour se référer à la table sur laquelle porte la règle. NEW est valide dans les règles ON INSERT et ON UPDATE pour désigner la nouvelle ligne en cours d'insertion ou de mise à jour. OLD est valide dans les règles ON UPDATE et ON DELETE pour désigner la ligne existante en cours de modification ou de suppression.

4-6-2-c. Syntaxe de suppression▲

Sélectionnez

```
DROP RULE nom ON relation [ CASCADE | RESTRICT ]
```

DROP RULE :

- Supprime une règle de réécriture.

nom :

- Le nom de la règle à supprimer.

relation :

- Le nom (qualifié ou non du nom du schéma) de la table ou vue où s'applique la règle.

CASCADE :

- Supprime automatiquement les objets dépendant de la règle.

RESTRICT :

- Refuse de supprimer la règle si un objet en dépend. Ceci est la valeur par défaut.

4-6-3. Vues (CREATE VIEW)▲

4-6-3-a. Description▲

Les vues sont des tables virtuelles qui « contiennent » le résultat d'une requête SELECT. L'un des intérêts de l'utilisation des vues vient du fait que la vue ne stocke pas les données, mais fait référence à une ou plusieurs tables d'origine à travers une requête SELECT, requête qui est exécutée chaque fois que la vue est référencée. De ce fait, toute modification de données dans les tables d'origine est immédiatement visible dans la vue dès que celle-ci est à nouveau référencée dans une requête.

Les utilisations possibles d'une vue sont multiples :

- Cacher aux utilisateurs certaines colonnes ou certaines lignes en mettant à leur disposition des vues de projection ou de sélection. Ceci permet de fournir un niveau de confidentialité et de sécurité supplémentaire.
- Simplifier l'utilisation de tables comportant de nombreuses colonnes, de nombreuses lignes ou des noms complexes, en créant des vues avec des structures plus simples et des noms plus intelligibles.
- Nommer des requêtes fréquemment utilisées pour simplifier et accélérer l'écriture de requête y faisant référence.

4-6-3-b. Syntaxe de définition▲

Voici la syntaxe de création d'une vue :

Sélectionnez

```
CREATE [ OR REPLACE ] VIEW nom [ ( nom_colonne [, ...] ) ] AS requête
CREATE VIEW :
```

définit une nouvelle vue.

CREATE OR REPLACE VIEW :

- définit une nouvelle vue, ou la remplace si une vue du même nom existe déjà. Vous pouvez seulement remplacer une vue avec une nouvelle requête qui génère un ensemble de colonnes identiques.

nom :

- Le nom de la vue à créer (qualifié ou non du nom du schéma). Si un nom de schéma (cf. section 4.6.4Schémas (CREATE SCHEMA)) est donné (par exemple CREATE VIEW monschema.mavue …), alors la vue est créée dans le schéma donné. Dans les autres cas, elle est créée dans le schéma courant. Le nom de la vue doit être différent du nom des autres vues, tables, séquences ou index du même schéma.

nom_colonne :

- Une liste optionnelle de noms à utiliser pour les colonnes de la vue. Si elle n'est pas donnée, le nom des colonnes sera déduit de la requête.

requête :

- Une requête (c'est-à-dire une instruction SELECT) qui définit les colonnes et les lignes de la vue.

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.

Avec PostgreSQL les vues ne sont que consultables par des instructions SELECT (i.e. lecture seule). Aucune autre opération n'est possible (insertion, mise à jour ou suppression de lignes). Par contre, la notion de règles permet, avec PostgreSQL, d'implémenter ces fonctionnalités. Cette notion s'avère plus souple et puissante que les restrictions communément appliquées aux SGBD classiques.

4-6-3-c. Implémentation interne▲

Avec PostgreSQL, les vues sont implémentées en utilisant le système de règles. En fait, il n'y aucune différence entre

Sélectionnez

```
CREATE VIEW mavue AS SELECT * FROM matable;
```

et ces deux commandes

Sélectionnez

```
CREATE TABLE mavue (liste de colonnes identique à celle de matable);

CREATE RULE "_RETURN" AS ON SELECT TO mavue DO INSTEAD
    SELECT * FROM matable;
```

parce que c'est exactement ce que fait la commande CREATE VIEW en interne. Ainsi, pour l'analyseur, il n'y a aucune différence entre une table et une vue : il s'agit de relations.

4-6-3-d. Syntaxe de suppression▲

Sélectionnez

```
DROP VIEW nom [, ...] [ CASCADE | RESTRICT ]
```

DROP VIEW :

- Supprime une vue existante.

nom :

- Le nom de la vue à supprimer (qualifié ou non du nom du schéma).

CASCADE :

- Supprime automatiquement les objets qui dépendent de la vue (par exemple d'autres vues).

RESTRICT :

- Refuse de supprimer la vue si un objet en dépend. Ceci est la valeur par défaut.

4-6-4. Schémas (CREATE SCHEMA)▲

4-6-4-a. Description▲

Les schémas sont des espaces dans lesquels sont référencés des éléments (tables, vues, index…). La notion de schéma est très liée à la notion d'utilisateur ou de groupe d'utilisateurs.

4-6-4-b. Syntaxe de définition▲

Sélectionnez

```
CREATE SCHEMA nom_schéma
```

CREATE SCHEMA crée un nouveau schéma dans la base de données en cours. Le nom du schéma doit être distinct du nom des différents schémas existants dans la base de données en cours.

Le paramètre nom_schéma est le nom du schéma à créer.

4-6-4-c. Accès aux tables▲

Lorsqu'une table nom_table est dans un schéma nom_schema, pour la désigner, il faut faire précéder son nom par le nom du schéma qui la contient de la manière suivante : nom_schema.nom_table. C'est ce que l'on appelle un nom qualifié.

4-6-4-d. Le Chemin de Recherche de Schéma▲

Les noms qualifiés sont pénibles à écrire et il est, de toute façon, préférable de ne pas coder un nom de schéma dans une application. Donc, les tables sont souvent appelées par des noms non qualifiés (i.e. nom de la table lui-même). Le système détermine quelle table est appelée en suivant un chemin de recherche qui est une liste de schémas à regarder. La première table correspondante est considérée comme la table voulue. S'il n'y a pas de correspondance, une erreur est levée, même si des noms de table correspondants existent dans d'autres schémas dans la base.

Le premier schéma dans le chemin de recherche est appelé le schéma courant. En plus d'être le premier schéma parcouru, il est aussi le schéma dans lequel de nouvelles tables seront créées si la commande CREATE TABLE ne précise pas de nom de schéma.

Pour voir le chemin de recherche courant, utilisez la commande suivante :

Sélectionnez

```
SHOW search_path;
```

Pour ajouter un nouveau schéma mon_schema dans le chemin tout en conservant dans ce chemin le schéma par défaut (public), nous utilisons la commande :

Sélectionnez

```
SET search_path TO mon_schema,public;
```

4-6-4-e. Syntaxe de suppression▲

La commande DROP SCHEMA permet de supprimer des schémas de la base de données. La syntaxe de la commande est la suivante :

Sélectionnez

```
DROP SCHEMA nom [, ...] [ CASCADE | RESTRICT ]
```

Un schéma peut seulement être supprimé par son propriétaire ou par un super utilisateur.

nom :

- Le nom du schéma

CASCADE :

- Supprime automatiquement les objets (tables, fonctions, etc.) contenus dans le schéma.

RESTRICT :

- Refuse de supprimer le schéma s'il contient un objet. Ceci est la valeur par défaut.

4-7. SQL intégré▲

4-7-1. Introduction▲

Ce chapitre décrit le pacquage SQL intégré (ou embarqué) pour PostgreSQL ECPG. Il est compatible avec les langages C et C++ et a été développé par Linus Tolke et Michael Meskes.

Un programme SQL intégré est en fait un programme ordinaire, dans notre cas un programme en langage C, dans lequel nous insérons des commandes SQL incluses dans des sections spécialement marquées. Ainsi les instructions Embedded SQL commencent par les mots EXEC SQL et se terminent par un point-virgule (« ; »). Pour générer l'exécutable, le code source est d'abord traduit par le préprocesseur SQL qui convertit les sections SQL en code source C ou C++, après quoi il peut être compilé de manière classique.

Le SQL intégré présente des avantages par rapport à d'autres méthodes pour prendre en compte des commandes SQL dans du code C. Par exemple, le passage des informations de et vers les variables du programme C est entièrement pris en charge. Ensuite, le code SQL du programme est vérifié syntaxiquement au moment de la précompilation. Enfin, le SQL intégré en C est spécifié dans le standard SQL et supporté par de nombreux systèmes de bases de données SQL. L'implémentation PostgreSQL est conçue pour correspondre à ce standard autant que possible, afin de rendre le code facilement portable vers des SGBD autre que PostgreSQL.

Comme alternative au SQL intégré, on peut citer l'utilisation d'une API (Application Programming Interface) permettant au programme de communiquer directement avec le SGBD via des fonctions fournies par l'API. Dans ce cas de figure, il n'y a pas de précompilation à effectuer. Se référer à la documentation PostgreSQL [28] pour plus d'information à ce sujet : Chapitre 27. libpq - Bibliothèque C

4-7-2. Connexion au serveur de bases de données▲

4-7-2-a. Introduction▲

Quel que soit le langage utilisé (C, Java, PHP, etc.), pour pouvoir effectuer un traitement sur une base de données, il faut respecter les étapes suivantes :

1. établir une connexion avec la base de données ;
2. récupérer les informations relatives à la connexion ;
3. effectuer les traitements désirés (requêtes ou autres commandes SQL) ;
4. fermer la connexion avec la base de données.

Nous allons voir dans cette section comment ouvrir et fermer une connexion, et nous verrons dans les sections suivantes comment effectuer des traitements.

4-7-2-b. Ouverture de connexion▲

La connexion à une base de données se fait en utilisant l'instruction suivante :

Sélectionnez

```
EXEC SQL CONNECT TO cible [AS nom_connexion] [USER utilisateur];
```

La cible cible peut être spécifiée de l'une des façons suivantes :

- nom_base[@nom_hôte][:port] ;
- tcp:postgresql://nom_hôte [:port] [/nom_base][? options] ;
- unix:postgresql://nom_hôte[: port][/nom_base][? options] ;
- une chaîne SQL littérale contenant une des formes ci-dessus ;
- une référence à une variable contenant une des formes ci-dessus ;
- DEFAULT.

En pratique, utiliser une chaîne littérale (entre guillemets simples) ou une variable de référence génère moins d'erreurs. La cible de connexion DEFAULT initie une connexion sur la base de données par défaut avec l'utilisateur par défaut. Aucun nom d'utilisateur ou nom de connexion ne pourrait être spécifié isolément dans ce cas.

Il existe également différentes façons de préciser l'utilisateur utilisateur :

- nom_utilisateur
- nom_utilisateur/ mot_de_passe
- nom_utilisateur IDENTIFIED BY mot_de_passe
- nom_utilisateur USING mot_de_passe

nom_utilisateur et mot_de_passe peuvent être un identificateur SQL, une chaîne SQL littérale ou une référence à une variable de type caractère.

nom_connexion est utilisé pour gérer plusieurs connexions dans un même programme. Il peut être omis si un programme n'utilise qu'une seule connexion. La dernière connexion ouverte devient la connexion courante, utilisée par défaut lorsqu'une instruction SQL est à exécuter.

4-7-2-c. Fermeture de connexion▲

Pour fermer une connexion, utilisez l'instruction suivante :

Sélectionnez

```
EXEC SQL DISCONNECT [connexion];
```

Le paramètre connexion peut prendre l'une des valeurs suivantes :

- nom_connexion
- DEFAULT
- CURRENT
- ALL

Si aucun nom de connexion n'est spécifié, c'est la connexion courante qui est fermée. Il est préférable de toujours fermer explicitement chaque connexion ouverte.

4-7-3. Exécuter des commandes SQL▲

Toute commande SQL, incluse dans des sections spécialement marquées, peut être exécutée à l'intérieur d'une application SQL intégrée. Ces sections se présentent toujours de la manière suivante :

Sélectionnez

```
EXEC SQL instructions_SQL ;
```

Dans le mode par défaut, les instructions ne sont validées que lorsque EXEC SQL COMMIT est exécuté. L'interface SQL intégrée supporte aussi la validation automatique des transactions via l'instruction EXEC SQL SET AUTOCOMMIT TO ON. Dans ce cas, chaque commande est automatiquement validée. Ce mode peut être explicitement désactivé en utilisant EXEC SQL SET AUTOCOMMIT TO OFF.

Voici un exemple permettant de créer une table :

Sélectionnez

```
EXEC SQL create table individu ( num_individu integer primary key,
                                 nom varchar(64), prenom varchar(64) );
EXEC SQL COMMIT;
```

4-7-4. Les variables hôtes▲

4-7-4-a. Introduction aux variables hôtes▲

La transmission de données entre le programme C et le serveur de base de données est particulièrement simple en SQL intégré. En effet, il est possible d'utiliser une variable C, dans une instruction SQL, simplement en la préfixant par le caractère deux-points (« : »). Par exemple, pour insérer une ligne dans la table individu on peut écrire :

Sélectionnez

```
EXEC SQL INSERT INTO individu VALUES (:var_num, 'Poustopol', :var_prenom);
```

Cette instruction fait référence à deux variables C nommées var_num et var_prenom et utilise aussi une chaîne littérale SQL ('Poustopol') pour illustrer que vous n'êtes pas restreint à utiliser un type de données plutôt qu'un autre.

Dans l'environnement SQL, nous appelons les références à des variables C des **variables hôtes**.

4-7-4-b. Déclaration des variables hôtes▲

Les **variables hôtes** sont des variables de langage C identifiées auprès du préprocesseur SQL. Ainsi, pour être définies, les variables hôtes doivent être placées dans une section de déclaration, comme suit :

Sélectionnez

```
EXEC SQL BEGIN DECLARE SECTION;
  declarations_des_variables_C
EXEC SQL END DECLARE SECTION;
```

Vous pouvez avoir autant de sections de déclaration dans un programme que vous le souhaitez.

Les variables hôtes peuvent remplacer les constantes dans n'importe quelle instruction SQL. Lorsque le serveur de base de données exécute la commande, il utilise la valeur de la variable hôte. Notez toutefois qu'une variable hôte ne peut pas remplacer un nom de table ou de colonne. Comme nous l'avons déjà dit, dans une instruction SQL, le nom de la variable est précédé du signe deux-points (« : ») pour le distinguer d'autres identificateurs admis dans l'instruction.

Les initialisations sur les variables sont admises dans une section de déclaration. Les sections de déclarations sont traitées comme des variables C normales dans le fichier de sortie du précompilateur. Il ne faut donc pas les redéfinir en dehors des sections de déclaration. Les variables qui n'ont pas pour but d'être utilisées dans des commandes SQL peuvent être normalement déclarées en dehors des sections de déclaration.

Les variables en langage C ont leur portée normale au sein du bloc dans lequel elles sont définies. Toutefois, le préprocesseur SQL n'analyse pas le code en langage C. Par conséquent, il ne respecte pas les blocs C. Aussi, pour le préprocesseur SQL, les variables hôtes sont globales : il n'est pas possible que deux de ces variables portent le même nom.

4-7-4-c. Types des variables hôtes▲

Seul un nombre limité de types de données du langage C est supporté pour les variables hôtes. En outre, certains types de variable hôte n'ont pas de type correspondant en langage C. Dans ce cas, des macros prédéfinies peuvent être utilisées pour déclarer les variables hôtes. Par exemple, le type prédéfini VARCHAR est la structure adéquate pour interfacer des données SQL de type varchar. Une déclaration comme

Sélectionnez

```
VARCHAR var[180];
```

est en fait convertie par le préprocesseur en une structure :

Sélectionnez

```
struct varchar_var { int len; char arr[180]; } var;
```

4-7-4-d. Utilisation d'une variable hôte : clause INTO▲

Dans le cas d'une requête de ligne unique, c'est-à-dire qui n'extrait pas plus d'une ligne de la base de données, les valeurs renvoyées peuvent être stockées directement dans des variables hôtes. Cependant, contrairement au langage C ou C++, le SQL est un langage ensembliste : une requête peut très bien retourner plus d'une ligne. Dans ce cas, il faut faire appel à la notion de curseur que nous abordons dans la section 4.7.7Curseurs pour résultats à lignes multiples.

Dans le cas d'une requête de ligne unique, une nouvelle clause INTO est intercalée entre la clause SELECT et la clause FROM. La clause INTO contient une liste de variables hôtes destinée à recevoir la valeur de chacune des colonnes mentionnées dans la clause SELECT. Le nombre de variables hôtes doit être identique au nombre de colonnes de la clause SELECT. Les variables hôtes peuvent être accompagnées de variables indicateur afin de prendre en compte les résultats NULL (cf. section 4.7.5Variables indicateur).

Lors de l'exécution de l'instruction SELECT, le serveur de base de données récupère les résultats et les place dans les variables hôtes. Si le résultat de la requête contient plusieurs lignes, le serveur renvoie une erreur. Si la requête n'aboutit pas à la sélection d'une ligne, un avertissement est renvoyé. Les erreurs et les avertissements sont renvoyés dans la structure SQLCA, comme décrit dans la section 4.7.6Gestion des erreurs.

Par exemple, en reprenons la base de données les films et les cinémas, et une requête que nous avons déjà rencontrée section 4.5.2Les clauses GROUP BY et HAVING et les fonctions d'agrégation : « nombre de fois que chacun des films a été projeté ». Nous pouvons récupérer les résultats de cette requête de ligne unique dans des variables hôtes de la manière suivante :

Sélectionnez

```
EXEC SQL BEGIN DECLARE SECTION;
  VARCHAR titre[128];
  int id_film;
  int nb_proj;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT num_film, titre, COUNT(*)
  INTO :id_film, :titre, :nb_proj
  FROM film NATURAL JOIN projection
  GROUP BY num_film, titre;
```

4-7-5. Variables indicateur▲

4-7-5-a. Présentation▲

Les variables indicateur sont des variables en langage C qui fournissent des informations complémentaires pour les opérations de lecture ou d'insertion de données. Il existe plusieurs types d'utilisation pour ces variables.

Valeurs NULL :

- Pour permettre aux applications de gérer les valeurs NULL.

Troncature de chaînes :

- Pour permettre aux applications de gérer les cas où les valeurs lues doivent être tronquées pour tenir dans les variables hôtes.

Erreurs de conversion :

- Pour stocker les informations relatives aux erreurs.

Une variable indicateur est une variable hôte de type int suivant immédiatement une variable hôte normale dans une instruction SQL.

4-7-5-b. Utilisation de variables indicateur▲

Dans les données SQL, la valeur NULL représente un attribut inconnu ou une information non applicable. Il ne faut pas confondre la valeur NULL de SQL avec la constante du langage C qui porte le même nom (NULL). Cette dernière représente un pointeur non initialisé, incorrect ou ne pointant pas vers un contenu valide de zone mémoire.

La valeur NULL n'équivaut à aucune autre valeur du type défini pour les colonnes. Ainsi, si une valeur NULL est lue dans la base de données et qu'aucune variable indicateur n'est fournie, une erreur est générée (SQLE_NO_INDICATOR). Pour transmettre des valeurs NULL à la base de données ou en recevoir des résultats NULL, des variables hôtes d'un type particulier sont requises : les variables indicateur.

Par exemple, dans l'exemple précédent, une erreur est générée si, pour une raison quelconque, le titre du film n'existe pas et que sa valeur est NULL. Pour s'affranchir de ce problème, on utilise une variable indicateur de la manière suivante :

Sélectionnez

```
EXEC SQL BEGIN DECLARE SECTION;
  VARCHAR titre[128];
  int id_film;
  int nb_proj;
  int val_ind;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT num_film, titre, COUNT(*)
  INTO :id_film, :titre :val_ind, :nb_proj
  FROM film NATURAL JOIN projection
  GROUP BY num_film, titre;
```

Dans cet exemple, la variable indicateur val_ind vaudra zéro si la valeur retournée n'est pas NULL et elle sera négative si la valeur est NULL. Si la valeur de l'indicateur est positive, cela signifie que la valeur retournée n'est pas NULL, mais que la chaîne a été tronquée pour tenir dans la variable hôte.

4-7-6. Gestion des erreurs▲

4-7-6-a. Configurer des rappels : instruction WHENEVER▲

L'instruction WHENEVER est une méthode simple pour intercepter les erreurs, les avertissements et les conditions exceptionnelles rencontrés par la base de données lors du traitement d'instructions SQL. Elle consiste à configurer une action spécifique à exécuter à chaque fois qu'une condition particulière survient. Cette opération s'effectue de la manière suivante :

Sélectionnez

```
EXEC SQL WHENEVER condition action;
```

Le paramètre condition peut prendre une des valeurs suivantes :

SQLERROR :

- L'action spécifiée est appelée lorsqu'une erreur survient pendant l'exécution d'une instruction SQL.

SQLWARNING :

- L'action spécifiée est appelée lorsqu'un avertissement survient pendant l'exécution d'une instruction SQL.

NOT FOUND :

- L'action spécifiée est appelée lorsqu'une instruction ne récupère ou n'affecte aucune ligne.

Le paramètre action peut avoir une des valeurs suivantes :

CONTINUE :

- Signifie effectivement que la condition est ignorée. C'est l'action par défaut.

SQLPRINT :

- Affiche un message sur la sortie standard. Ceci est utile pour des programmes simples ou lors d'un prototypage. Les détails du message ne peuvent pas être configurés.

STOP :

- Appel de exit(1), ce qui terminera le programme.

BREAK :

- Exécute l'instruction C break. Cette action est utile dans des boucles ou dans des instructions switch.

GOTO label **et** GO TO label :

- Saute au label spécifié (en utilisant une instruction C goto).

CALL nom (args) **et** DO nom (args) :

- Appelle les fonctions C spécifiées avec les arguments spécifiés.

Le standard SQL ne définit que les actions CONTINUE et GOTO ou GO TO.

L'instruction WHENEVER peut être insérée en un endroit quelconque d'un programme SQL intégré. Cette instruction indique au préprocesseur de générer du code après chaque instruction SQL. L'effet de cette instruction reste actif pour toutes les instructions en SQL intégré situées entre la ligne de l'instruction WHENEVER et l'instruction WHENEVER suivante contenant la même condition condition d'erreur, ou jusqu'à la fin du fichier source.

Les conditions d'erreur sont fonction du positionnement dans le fichier source de langage C et non du moment où l'instruction est exécutée.

Cette instruction est fournie pour vous faciliter le développement de programmes simples. Il est plus rigoureux de contrôler les conditions d'erreur en vérifiant directement le champ sqlcode de la zone SQLCA (cf. section suivante). Dans ce cas, l'instruction WHENEVER est inutile. En fait, l'instruction WHENEVER se contente de demander au préprocesseur de générer un test if (SQLCODE) après chaque instruction SQL.

4-7-6-b. Zone de communication SQL (SQLCA)▲

La zone de communication SQL (SQLCA) est une zone de mémoire qui permet, pour chaque demande adressée à la base de données, de communiquer des statistiques et de signaler des erreurs. En consultant la zone SQLCA, vous pouvez tester un code d'erreur spécifique. Un code d'erreur s'affiche dans les champs sqlcode et sqlstate lorsqu'une requête adressée à la base de données provoque une erreur. Une variable SQLCA globale (sqlca) est définie dans la bibliothèque d'interface, elle a la structure suivante :

Sélectionnez

```
struct {
  char sqlcaid[8];
  long sqlabc;
  long sqlcode;
  struct {
    int sqlerrml;
    char sqlerrmc[70];
    } sqlerrm;
  char sqlerrp[8];
  long sqlerrd[6];
  char sqlwarn[8];
  char sqlstate[5];
} sqlca;
```

SQLCA couvre à la fois les avertissements et les erreurs. Si plusieurs avertissements ou erreurs surviennent lors de l'exécution d'une instruction, alors sqlca ne contient que les informations relatives à la dernière. Si aucune erreur ne survient dans la dernière instruction SQL, sqlca.sqlcode vaut 0 et sqlca.sqlstate vaut « 00000 ». Si un avertissement ou une erreur a eu lieu, alors sqlca.sqlcode sera négatif et sqlca.sqlstate sera différent de « 00000 ».

Les champs sqlca.sqlstate et sqlca.sqlcode sont deux schémas différents fournissant des codes d'erreur. Les deux sont spécifiés dans le standard SQL, mais sqlcode est indiqué comme obsolète dans l'édition de 1992 du standard et a été supprimé dans celle de 1999. Du coup, les nouvelles applications sont fortement encouragées à utiliser sqlstate.

4-7-7. Curseurs pour résultats à lignes multiples▲

Lorsque vous exécutez une requête dans une application, le jeu de résultats est constitué d'un certain nombre de lignes. En général, vous ne connaissez pas le nombre de lignes que l'application recevra avant d'exécuter la requête. Les curseurs constituent un moyen de gérer les jeux de résultats d'une requête à lignes multiples.

Les curseurs vous permettent de naviguer dans les résultats d'une requête et d'effectuer des insertions, des mises à jour et des suppressions de données sous-jacentes en tout

point d'un jeu de résultats.

Pour gérer un curseur, vous devez respecter les étapes suivantes :

1. Déclarer un curseur pour une instruction SELECT donnée à l'aide de l'instruction DECLARE :

   Sélectionnez

   ```
   EXEC SQL DECLARE nom_curseur CURSOR FOR requête_select ;
   ```

2. Ouvrir le curseur à l'aide de l'instruction OPEN :

   Sélectionnez

   ```
   EXEC SQL OPEN nom_curseur ;
   ```

3. Récupérer une par une les lignes du curseur à l'aide de l'instruction FETCH :

   Sélectionnez

   ```
   FETCH [ [ NEXT | PRIOR | FIRST | LAST | { ABSOLUTE | RELATIVE } nombre ]
           { FROM | IN } ] nom_curseur
   INTO liste_variables
   NEXT :
   ```

4. Récupère la ligne suivante. Ceci est la valeur par défaut.

   PRIOR :

   - Récupère la ligne précédente.

   FIRST :

   - Récupère la première ligne de la requête (identique à ABSOLUTE 1).

   LAST :

   - Récupère la dernière ligne de la requête (identique à ABSOLUTE -1).

   ABSOLUTE nombre :

   - Récupère la nombre ème ligne de la requête ou la abs(nombre) ème ligne à partir de la fin si nombre est négatif. La position avant la première ligne ou après la dernière si nombre est en-dehors de l'échelle ; en particulier, ABSOLUTE 0 se positionne avant la première ligne.

   RELATIVE nombre :

   - Récupère la nombre ème ligne ou la abs(nombre) ème ligne avant si nombre est négatif. RELATIVE 0 récupère de nouveau la ligne actuelle si elle existe.

   nom_curseur :

   - Le nom d'un curseur ouvert.

   liste_variables :

   - La liste des variables hôtes destinées à recevoir la valeur de chacun des attributs de la ligne courante. Le nombre de variables hôtes doit être identique au nombre de colonnes de la table résultat.
5. Continuez l'extraction des lignes tant qu'il y en a.
6. Fermer le curseur à l'aide de l'instruction CLOSE :

   Sélectionnez

   ```
   EXEC SQL CLOSE nom_curseur
   ```

Lors de son ouverture, un curseur est placé avant la première ligne. Par défaut, les curseurs sont automatiquement refermés à la fin d'une transaction.

Voici un exemple utilisant la commande FETCH :

Sélectionnez

```
EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test;
EXEC SQL OPEN foo;
while (...) {
  EXEC SQL FETCH NEXT FROM foo INTO :v1, :v2;
  ...
}
```

4-7-8. Précompilation et compilation▲

4-7-8-a. Inclusion de fichiers▲

Pour inclure un fichier externe SQL intégré dans votre programme, utilisez la commande :

Sélectionnez

```
EXEC SQL INCLUDE nom_fichier;
```

Cette commande indique au préprocesseur du SQL intégré de chercher un fichier nommé nom_fichier.h, de traiter et de l'inclure dans le fichier C généré. Du coup, les instructions SQL intégrées du fichier inclus sont gérées correctement.

En utilisant la directive classique

Sélectionnez

```
#include <nom_fichier.h>
```

le fichier nom_fichier.h ne serait pas sujet au prétraitement des commandes SQL. Naturellement, vous pouvez continuer à utiliser la directive #include pour inclure d'autres fichiers d'en-tête.

4-7-8-b. Précompilation et compilation▲

La première étape consiste à traduire les sections SQL intégré en code source C, c'est-à-dire en appels de fonctions de la librairie libecpg. Cette étape est assurée par le préprocesseur appelé ecpg qui est inclus dans une installation standard de PostgreSQL. Les programmes SQL intégré sont nommés typiquement avec une extension .pgc. Si vous avez un fichier programme nommé prog.pgc, vous pouvez le passer au préprocesseur par la simple commande :

Sélectionnez

```
ecpg prog1.pgc
```

Cette étape permet de créer le fichier prog.c. Si vos fichiers en entrée ne suivent pas le modèle de nommage suggéré, vous pouvez spécifier le fichier de sortie explicitement en utilisant l'option -o.

Le fichier traité par le préprocesseur peut alors être compilé de façon classique, par exemple :

Sélectionnez

```
cc -c prog.c
```

Cette étape permet de créer le fichier prog.o. Les fichiers sources en C générés incluent les fichiers d'en-tête provenant de l'installation de PostgreSQL. Si vous avez installé PostgreSQL à un emplacement qui n'est pas parcouru par défaut, vous devez ajouter une option comme -I/usr/local/pgsql/include sur la ligne de commande de la compilation.

Vous devez enfin lier le programme avec la bibliothèque libecpg qui contient les fonctions nécessaires. Ces fonctions récupèrent l'information provenant des arguments, exécutent la commande SQL en utilisant l'interface libpq et placent le résultat dans les arguments spécifiés pour la sortie. Pour lier un programme SQL intégré, vous devez donc inclure la bibliothèque libecpg :

Sélectionnez

```
cc -o monprog prog.o -lecpg
```

De nouveau, vous pourriez avoir besoin d'ajouter une option comme -L/usr/local/pgsql/lib sur la ligne de commande.

4-7-9. Exemple complet▲

Voici un exemple complet qui effectue les opérations suivantes :

- connexion à la base ;
- vérification de la réussite de la connexion ;
- affichage du contenu de la table individu en utilisant un curseur ;
- fermeture de la connexion.

Sélectionnez

```
#include <stdio.h>
// ____ pour gérer les erreurs
EXEC SQL INCLUDE sqlca;
// ____ Définition des variables hôtes
EXEC SQL BEGIN DECLARE SECTION;
  char var_nom[256];
  char var_prenom[256];
  int  var_num;
EXEC SQL END DECLARE SECTION;

int main(void){
  // ____ Ouverture de la connexion à la base de données
  EXEC SQL CONNECT TO nom_base@aquanux;
  if(sqlca.sqlcode) {
    printf("erreur %s\n",sqlca.sqlerrm.sqlerrmc);
    exit(0);
  }
  printf(" connexion réussie \n");
  // ____ Utilisation d'un curseur pour afficher le contenu de la table individu
  EXEC SQL DECLARE curseur_individu CURSOR FOR
    SELECT num_individu, nom, prenom FROM individu;
  EXEC SQL OPEN curseur_individu;
  // Boucle d'affichage
  while(SQLCODE==0) {
```

```
    EXEC SQL FETCH FROM curseur_individu INTO :var_num, :var_nom, :var_prenom;
    printf("L'individu %d est %s %s\n, var_num, var_prenom, var_nom);
  }
  EXEC SQL CLOSE curseur_individu;
  // ____ Fermeture de connexion
  printf(" Déconnexion \n");
  EXEC SQL DISCONNECT;
  return 0;
}
```

En supposant que ce programme est enregistré dans un fichier nommé prog.pgc, l'exécutable est obtenu de la manière suivante :

Sélectionnez

```
ecpg prog.pgc
cc -c prog.c
cc -o prog prog.o -lecpg
```
