SSTEM : introduction à SQL

Présentation

SQL (Structured Query Language) est un langage complet de **gestion de bases de données relationnelles**. Il a été conçu par IBM dans les années 1970. Il comprend :

- un **langage de définition de données** (LDD, ordres CREATE, ALTER, DROP) qui permet de modifier la structure de la base ;
- un **langage de manipulation de données** (LMD, ordres UPDATE, INSERT, DELETE, SELECT) qui permet de consulter ou modifier le contenu de la base ;
- un **langage de contrôle de l'accès aux données** (LCD, ordres GRANT, REVOKE) qui permet de gérer les droits d'accès donnés aux utilisateurs envers les différentes données de la base ;
- un **langage de contrôle des transactions** (LCT), qui permet de gérer les transactions, c'est-à-dire de rendre atomiques divers ordres enchaînés en séquence.
- et d'autres modules destinés notamment à écrire des routines (procédures, fonctions ou déclencheurs) et interagir avec des langages externes.

Les principaux systèmes de gestion de bases de données (SGBD) implémentant SQL sont, dans le domaine des logiciels propriétaires, Oracle et SQL Server et, dans le secteur des logiciels libres, MySQL et PostgreSQL.

Quelques mots d'histoire

L'algèbre relationnelle est introduite en juin 1970 par Edgar Frank Codd dans la revue scientifique Communications of the ACM (Association for Computing Machinery). Ce modèle s'impose rapidement comme référence pour les SGBD. IBM met alors en chantier un langage visant à mettre œuvre le modèle de Codd : Donald D. Chamberlin and Raymond F. Boyce proposent ainsi, au début des années 1970, la première version du langage *Structured English Query Language* (SEQUEL) – devenu ensuite SQL. À la fin des années 1970, Relational Software Inc. (désormais connu sous le nom d'Oracle Corporation) lance la première implémentation commerciale de SQL sous le nom d'Oracle V2 pour les ordinateurs VAX.

SQL devient une recommandation de l'American National Standards Institute (ANSI) en 1986 puis une norme internationale reconnue par l'International Organization for Standardization (ISO) en 1987 (**SQL 86** ou **SQL1**). La norme internationale SQL est passée ensuite par un certain nombre de révisions, les premières normes, trop incomplètes, ayant été ignorées par les principaux SGBD. **SQL1** offrait ainsi des requêtes compilées puis exécutés depuis un programme d'application, mais ne proposait que des opérations ensemblistes restreintes. La norme **SQL2** (ou **SQL 92**) intégrait des requêtes dynamiques (exécution immédiate ou différée), différents types de jointures et des opérations ensemblistes élargies. La plupart des SGBD qui dominent le marché intègre l'ensemble minimum de fonctionnalités à respecter pour se dire de la norme SQL 2. L'année 1999 a vu l'arrivée de la norme **SQL 3** (ou **SQL 99**) avec la prise en compte des expressions rationnelles et de fonctions objets. L'évolution naturelle de SQL conduit à de nouveaux enrichissements, avec un élargissement des opérations ensemblistes et l'introduction de fonctions pour la manipulation de données XML.

Malgré l'existence de ces normes, il existe des différences non négligeables entre les syntaxes et les fonctionnalités des différents SGBD. Il n'est donc pas aisé d'écrire du code pleinement portable et le travail sur tel ou tel SGBD nécessite généralement quelques adaptations. Ainsi le type « chaîne de 42 caractères » s'écrit `varchar(42)` sous MySQL, `character varying(42)` sous PostgreSQL et `VARCHAR2(42)` sous Oracle.

Syntaxe SQL

Les exemples suivants sont donnés pour PostgreSQL sur une base de données baptisée impexp, constituée des deux tables suivantes :

`produit(`<u>`code_produit`</u>`,origine,prix_unitaire,nom_produit)`
`commande(`<u>`numero_cde`</u>`,code_produit,quantite,continent)`

Langage de requêtes

Sélection d'éléments dans une base de données (duplicatas possibles)
Exemple : `SELECT * FROM produit;`

Sélection d'éléments dans une base de données (éléments distincts)
Exemple : `SELECT DISTINCT nom_produit FROM produit;`

Sélection d'éléments en opérant des restrictions sur les lignes choisies
`SELECT liste_de_champs FROM table WHERE liste_de_contraintes;`

La liste des contraintes `liste_de_contraintes` que les lignes doivent respecter est exprimée sous la forme de conditions faisant intervenir :

- Des opérateurs de comparaison :
  - attribut ~ valeur
  - attribut1 ~ attribut2

  Où ~ ϵ { `=, <, <=, >, >=, <>` }
- Des opérateurs logiques :
  - Opérateur « et » : `condition_1 AND condition_2 AND condition_3`
  - Opérateur « ou » : `condition_1 OR condition_2 OR condition_3`
  - Opérateur « non » : `NOT condition_1`

Exemple : `SELECT * FROM produit WHERE code_produit>=200 AND prix_unitaire<>350;`

Il est également possible d'exprimer la restriction sous la forme d'un intervalle de recherche dans lequel les valeurs d'un attribut peuvent évoluer : attribut BETWEEN x AND y
Exemple : `SELECT * FROM produit WHERE prix_unitaire BETWEEN 150 AND 200;`

La recherche peut également être effectuée en exprimant qu'un attribut prend ses valeurs dans un certain ensemble :
Exemple : `SELECT * FROM commande WHERE continent IN ('Europe','Amérique','Asie');`

Il est parfois utile de pouvoir effectuer une recherche spécifiant non la valeur exacte d'un attribut mais des éléments de sa forme (par exemple : « afficher tous les produits dont la dénomination se termine par un 'e' ») : attribut `LIKE 'chaine'`
Dans ce cas, `'chaine'` peut inclure des jokers :

- `%` : remplace n'importe quel nombre de caractères
- `_` : remplace un caractère

Exemple : `SELECT * FROM produit WHERE nom_produit LIKE '%ane';`

Enfin, il est quelques fois nécessaire de rechercher les lignes pour lesquelles la valeur d'un certain attribut a été (respectivement n'a pas été) spécifiée : attribut `IS NOT NULL` (respectivement `IS NULL`).
Exemple : `SELECT * FROM produit WHERE nom_produit IS NOT NULL;`
`SELECT * FROM produit WHERE prix_unitaire IS NULL;`

Jointure
Une jointure permet de combiner les données apparaissant dans des tables différentes. Lorsqu'on définit une jointure, on préfixe chacun des champs par sa table d'origine, ce qui permet de sélectionner exactement les attributs à obtenir ou les champs à comparer.

Exemples : `SELECT produit.nom_produit, commande.numero_cde FROM produit, commande WHERE produit.code_produit=commande.code_produit;`

Il existe d'autres types de jointures permettant, par exemple, de conserver dans la jointure tous les éléments d'une des tables (et pas seulement les éléments dont les valeurs relatives à la colonne ayant servi à la jointure étaient égales).

Fonctions
Compter le nombre d'éléments correspondant à certaines spécifications :
Exemple : `SELECT COUNT(*) FROM produit WHERE prix_unitaire>=200;`

Sommer les valeurs d'un attribut :
Exemple : `SELECT SUM(quantite) FROM commande WHERE code_produit=2 AND continent='Asie';`

Obtenir la moyenne des valeurs d'un attribut :
Exemple : `SELECT AVG(prix_unitaire) FROM produit;`

Obtenir les valeurs minimales (ou maximales) d'un attribut :
Exemples : `SELECT MIN(prix_unitaire) FROM produit;`
`SELECT MAX(prix_unitaire) FROM produit;`

Transformer en majuscules tous les caractères d'une expression :
Exemple : `SELECT * FROM produit WHERE UPPER(nom_produit) LIKE UPPER('%ane');`

Agrégation de données
Exemples : `SELECT origine, AVG(prix_unitaire) FROM produit GROUP BY origine;`
`SELECT origine, AVG(prix_unitaire) FROM produit GROUP BY origine HAVING MIN(prix_unitaire)>=100;`

Tri de données
Exemple : `SELECT * FROM produit WHERE nom_produit LIKE '%e' ORDER BY prix_unitaire DESC;`
`ASC` (par défaut) permet de trier en ordre croissant tandis que `DESC` trie en ordre décroissant.

Imbrication de requêtes
Afin de gagner en finesse dans les interrogations effectuées sur une base, il est souvent utile d'imbriquer des requêtes.
Exemple : `SELECT * FROM produit WHERE code_produit IN (SELECT code_produit FROM commande WHERE quantite>=1000);`

Les opérateurs `ANY` et `ALL` permettent de comparer un attribut à un ensemble de valeurs obtenues par imbrication de requêtes.
Exemples :
```
SELECT nom_produit FROM produit WHERE prix_unitaire < ANY ( SELECT prix_unitaire
FROM commande WHERE continent='Europe');
SELECT nom_produit FROM produit WHERE prix_unitaire < ALL ( SELECT prix_unitaire
FROM commande WHERE continent='Europe');
```

La condition `ANY (SELECT F FROM …)` est vraie si et seulement si la comparaison est vraie au moins pour une valeur du résultat du bloc `(SELECT F FROM …)`
La condition `ALL (SELECT F FROM …)` est vraie si et seulement si la comparaison est vraie pour toute valeur du résultat du bloc `(SELECT F FROM …)`

L'opérateur `EXISTS` permet de vérifier si le résultat d'une requête est non vide.
Exemple : `SELECT AVG(prix_unitaire) FROM commande WHERE EXISTS (SELECT * FROM commande WHERE continent='Afrique' AND quantite>=450);`

Alias
Dans le cas où les noms des tables d'une base de données sont complexes, il est possible d'alléger l'écriture des requêtes en utilisant des alias. Les deux requêtes suivantes sont ainsi équivalentes :
```
SELECT produit.nom_produit, commande.numero_cde FROM produit, commande WHERE
produit.code_produit=commande.code_produit;
SELECT P.nom_produit, C.numero_cde FROM produit P, commande C
WHERE P.code_produit=C.code_produit;
```

Langage de définition de données

Création d'une base de données

```
CREATE DATABASE impexp;
```

Suppression d'une base de données

```
DROP DATABASE impexp;
```

Création d'une table

```
CREATE TABLE nomtable(nom-col  type-col  [DEFAULT valeur] [ [CONSTRAINT]
contrainte-col] )*[ [CONSTRAINT] contrainte-table ]* |  AS requête-SQL );
```

Exemple :

```
CREATE TABLE produit(code_produit int NOT NULL, origine varchar(50),
prix_unitaire int, nom_produit varchar(50), PRIMARY KEY(code_produit));
```

Types (`type-col`)

	PostgreSQL
entier	`int`
réel	`float`
caractère	`char`
chaîne de caractères	`varchar(..)`
booléen	`bool`
date (YYYY-MM-DD)	`date`
heure (HH:MM:SS)	`time`
date et heure	`timestamp`

Contraintes sur une colonne (`contrainte-col`)

- `NOT NULL` : le champ doit obligatoirement avoir une valeur
- `PRIMARY KEY` : définit la clef primaire (auquel cas le champ doit être unique et obligatoirement avoir une valeur)
- `UNIQUE` : interdit qu'une colonne contienne deux valeurs identiques
- `REFERENCES nom-table [(nom-col)] [action]` : définit une référence à une autre colonne
- `CHECK (condition)` : donne une condition que la colonne doit vérifier

Contraintes sur une table (`contrainte-table`)

- `PRIMARY KEY (nom-col*)` : établit une clef primaire
- `UNIQUE (nom-col*)`: interdit qu'une colonne contienne deux valeurs identiques
- `FOREIGN KEY (nom-col*) REFERENCES nom-table [(nom-col*)] [action]` : définit une référence à une clef externe
- `CHECK (condition)` : donne une condition que les colonnes de chaque ligne doivent vérifier

Suppression d'une table

Exemple : `DROP TABLE impexp;`

Langage de manipulation de données

Insertion dans une table

Exemple :

```
INSERT INTO produit(nom_produit,code_produit,prix_unitaire,origine)
VALUES('papaye',300,4,'Antilles');
```

Mise à jour dans une table

Exemple :

```
UPDATE produit SET nom_produit='mangue' WHERE nom_produit='coco';
```

Suppression dans une table

Exemple :

```
DELETE FROM produit WHERE nom_produit='pomme';
```

Transactions en SQL

Problème

Lorsque plusieurs utilisateurs partagent une même base de données, il est important de s'assurer que les différentes requêtes qu'ils effectuent ne conduisent pas à des données conflictuelles. Il est également primordial de protéger les données contre les cas où des échecs interviendraient pendant qu'une opération est effectuée. C'est de ces contraintes que découle la notion de transactions.

Qu'est-ce qu'une transaction ?

Une opération sur une base de données se divise souvent en plusieurs autres opérations élémentaires sur la base.
Les transactions offrent un mécanisme de regroupement d'une série d'opérations sur une base de données dans une opération logique. L'ensemble des modifications peut alors être entièrement validé (instruction SQL : `COMMIT`) ou entièrement annulé (instruction SQL : `ROLLBACK`). Ce mécanisme garantit la cohérence de la base avant et après la transaction.

Envisageons en effet le cas où un incident survient pendant une transaction. Il suffit alors de réaliser un « retour en arrière » (`ROLLBACK`) afin « d'effacer » les opérations entreprises.
Par contre si tout s'est bien passé, la transaction sera validée (`COMMIT`) afin que les changements soit définitivement pris en compte dans la base de données.

Une transaction est donc une séquence d'opérations liées, comportant des mises à jour ponctuelles d'une base de données devant vérifier les propriétés d'**Atomicité**, de **Cohérence**, d'**Isolation** et de **Durabilité** (**ACID**).

Les propriétés des transactions : ACID

Atomicité (*Atomicity*) : une transaction doit se terminer par une instruction `COMMIT` ou `ROLLBACK`.

Cohérence (*Consistency*) : l'exécution d'une transaction doit garantir le passage de la base de données d'un état cohérent à un autre. En cas d'échec, l'état cohérent initial doit être restauré.

Isolation (*Isolation*) : quand de multiples transactions sont lancées en parallèle, chacune d'entre elles ne doit pas être capable de voir les modifications incomplètes faites par les autres.

Durabilité (*Durability*) : lorsqu'une transaction est validée, le système doit garantir que les modifications seront conservées en cas de panne.

Implémentation pratique

Une transaction est réalisée en entourant les commandes SQL de la transaction avec les commandes `BEGIN` (ou, de manière équivalente, `BEGIN WORK` ou `START TRANSACTION`) et `COMMIT` (ou `ROLLBACK` pour restaurer l'état précédent de la base).

Exemple : problématique d'un virement bancaire

```
BEGIN;
UPDATE comptes SET balance = balance - 100.00 WHERE nom = 'Alice';
UPDATE comptes SET balance = balance + 100.00 WHERE nom = 'Bob';
COMMIT;
```

Par défaut, PostgreSQL (et MySQL) est lancé en mode d'**auto-validation** (`autocommit`). Cela signifie que PostgreSQL traite chaque instruction SQL comme étant exécutée dans une transaction. Si on ne lance pas une commande `BEGIN`, alors chaque instruction individuelle se trouve enveloppée avec un `BEGIN` et, en cas de succès, un `COMMIT` implicites. Un groupe d'instructions entouré par un `BEGIN` et un `COMMIT` est, quelques fois, appelé un **bloc transactionnel**.
Autrement dit, chaque modification effectuée est enregistrée immédiatement sur le disque par PostgreSQL.

Il est possible de configurer PostgreSQL en mode de non auto-validation grâce à la commande suivante :
`SET AUTOCOMMIT OFF;`

Points de sauvegarde

Il est possible de contrôler les instructions dans une transaction d'une façon plus granulaire avec l'utilisation des points de sauvegarde. Les points de sauvegarde permettent d'annuler des parties de la transaction tout en validant le reste. Après avoir défini un point de sauvegarde avec `SAVEPOINT`, il est possible, si nécessaire, d'annuler toutes les opérations effectuées depuis le point de sauvegarde en utilisant la commande `ROLLBACK TO`. Toutes les modifications de la transaction dans la base de données entre le moment où le point de sauvegarde est défini et celui où l'annulation est demandée sont annulées mais les modifications antérieures au point de sauvegarde sont conservées.

Après avoir annulé jusqu'à un point de sauvegarde, il reste défini. Il est donc possible d'annuler plusieurs fois de nouveau et de rester au même point. Par contre, quand on est sûr de ne plus avoir besoin d'annuler jusqu'à un point de sauvegarde particulier, celui-ci peut être libéré (à l'aide de la commande `RELEASE`) pour que le système puisse récupérer quelques ressources. Libérer un point de sauvegarde ou annuler les opérations jusqu'à ce point de sauvegarde libère évidemment tous les points de sauvegarde définis après lui.

Tout ceci survient à l'intérieur du bloc de transaction, donc ce n'est pas visible par les autres sessions de la base de données. Quand et si vous validez le bloc de transaction, les actions validées deviennent visibles en un seul coup aux autres sessions, alors que les actions annulées ne deviendront jamais visibles.

Exemple :
```
BEGIN;
UPDATE comptes SET balance = balance - 100.00 WHERE nom = 'Alice';
SAVEPOINT mon_pointdesauvegarde;
UPDATE comptes SET balance = balance + 100.00 WHERE nom = 'Bob';
ROLLBACK TO mon_pointdesauvegarde;
UPDATE comptes SET balance = balance + 100.00 WHERE nom = 'Bill';
COMMIT;
```

Quelques remarques sur ROLLBACK

Un `ROLLBACK` :
- Annule les écritures de la dernière transaction
- Annule les transactions qui utilisent les écritures de la transaction

Ce qui signifie qu'une transaction validée risque d'être annulée.

Il convient donc de respecter quelques **règles** de « bonne conduite » avant d'effectuer un `ROLLBACK`, autrement dit que les exécutions parallèles satisfassent les propriétés suivantes :
- Recouvrabilité, *i.e.* les exécutions doivent éviter l’annulation des transactions validées
- Pas d'annulations en cascade
- Exécution stricte, *i.e.* ne pas écrire simultanément la même donnée dans deux transactions