Support de cours SQL

Anne Etien

October 22, 2019

Contents

1

Introduction

1.1 Historique

En 1970, le docteur Codd, un chercheur d'IBM à San José aux États-Unis, propose une nouvelle manière d'aborder le traitement automatique de l'information, se basant sur la théorie de l'algèbre relationnel (théorie des ensembles et logique des prédicats). Cette proposition est faite afin de garantir une plus grande indépendance entre la théorie et l'implémentation physique des données au sein des machines.

De 1972 à 1976, IBM invente le langage d'interrogation SEQUEL (Structured English QUEry Language) qui deviendra ensuite SQL *Structured Query Language* c'est à dire Langage d'Interrogation Structuré.

En 1979, Relational Software, Inc. (actuellement Oracle Corporation) présenta la première version commercialement disponible de SQL, rapidement imité par d'autres fournisseurs.

SQL a été adopté comme recommandation par l'Institut de normalisation américaine (ANSI) en 1986, puis comme norme internationale par l'ISO en 1987 sous le nom de ISO/CEI 9075 - Technologies de l'information - Langages de base de données - SQL. De nouvelles versions sont apparues ensuite. On notera par exemple SQL-92 aussi nommé SQL2 sorti en 1992 qui apporte des révisions majeures au langage. SQL-99 (alias SQL3) en 1999 et dernièrement en 2011 SQL:2011.

Même si SQL est considéré comme le standard de toutes les bases de données relationnelles et commercialisées, il n'en reste pas moins vrai que chaque éditeur tend à développer son propre dialecte, c'est à dire à rajouter des éléments hors de la norme. Soit fonctionnellement identiques mais de syntaxe différentes soit fonctionnellement nouveau (le CONNECT BY d'Oracle pour la récursivité par exemple). Il est alors très difficile de porter une base de données SQL d'un serveur à l'autre. C'est un moindre mal si l'on a respecté au maximum la norme, mais bien souvent lorsque l'élément normatif est présent dans le SGBDR avec un élément spécifique,

c'est l'élément spécifique qui est proposé et documenté au détriment de la norme !

1.2 Caractéristiques

SQL est à la fois :

- un langage d'interrogation de la base (commande `SELECT`)
- un langage de manipulation des données (LMD) ; (commandes `UPDATE`, `INSERT`, `DELETE`)
- un langage de définition des données (LDD) ; (commandes `CREATE`, `ALTER`, `DROP`),
- un langage de contrôle de l'accès aux données (LCD)[1] ; (commandes `GRANT`, `REVOKE`),
- un langage de contrôle de transaction des données (LCT)[2] ; (commandes `SET TRANSACTION`, `BEGIN`, `COMMIT`, `ROLLBACK`).

1.3 Création, destruction et accès à la base

A Polytech Lille, nous utiliserons le SGBD PostgreSQL. Il faudra donc dans un premier temps exporter la variable `PGHOST` en tapant la commande suivante dans un terminal (attention à ne pas mettre d'espace autour du signe =) :

```
export PGHOST=serveur-etu.polytech-lille.fr
```

La manipulation de la base elle même se fait par des commandes Unix dans un terminal :

- Création d'une base (Les fragments de commande entre [ ] sont facultatifs)

  ```
  createdb nomBase [ -U comptePostgres ]
  ```

  L'option `-U` est facultative en générale, mais obligatoire pour les personnes dont le login Polytech contient un − et lorsque vous vous connecterez à Postgres depuis les comptes examen car contrairement à vous, on ne leur a pas créé de compte Postgres.

  Mon login Polytech ne contient pas de −, il correspond donc exactement à mon compte Postgres. Si par hasard, votre login Polytech contient un −, comme dans `un-exemp`, votre compte Postgres est obtenu

[1] Cet aspect du langage sera abordé en GIS4.
[2] Cet aspect du langage aussi.

en remplaçant le − par _ soit `un_exemp`. Ainsi pour moi, et si je veux créer la base `aetienHotel`[3] il s'agira de faire :

```
createdb aetienHotel -U aetien
```

- Accès à une base : `psql nomBase [ -U comptePostgres ]`

  Soit pour moi, et si je veux accéder à la base `aetienHotel`. Bien évidemment cette base aura été créée préalablement, (et pas détruite).

  ```
  psql aetienHotel -U aetien
  ```

- Destruction d'une base :

  ```
  dropdb nomBase [ -U comptePostgres ]
  ```

  Soit pour moi, et si je veux détruire la base `aetienHotel`

  ```
  dropdb aetienHotel -U aetien
  ```

1.4 Remarques préliminaires sur les SGBDR

Voici quelques points qu'il convient d'avoir à l'esprit lorsque l'on travaille sur des bases de données :

- Il existe une grande indépendance entre la couche abstraite que constitue le SGBDR et la couche physique que sont le ou les fichiers constituant une base de données. En l'occurrence il est déraisonnable de croire que les fichiers sont arrangés dans l'ordre des colonnes lors de la création des tables et que les données sont rangées dans l'ordre de leur insertion ou de la valeur de la clef.

- Il n'existe pas d'ordre spécifique pour les tables dans une base ou pour les colonnes dans une table, même si le SGBDR en donne l'apparence en renvoyant assez généralement l'ordre établi lors de la création (notamment pour les colonnes). Par conséquent les tables, colonnes, index... doivent être repérés par leur nom qui est donc unique dans l'espace de nommage associé (le schéma pour la table, la table pour la colonne...).

  Cependant, une table ou une colonne ne peut être utilisée avant d'avoir été préalablement définie.

- Les données sont systématiquement présentées sous forme de tables, et cela quel que soit le résultat attendu. Pour autant la table constituée par la réponse n'est pas forcément une table persistante, ce qui signifie que si vous voulez conserver les données d'une requête pour en faire

[3] Afin d'être sûr que votre nom de base est unique on y fera figurer votre login.

usage ultérieurement, il faudra créer un objet dans la base (table ou vue) afin d'y placer les données extraites.

- La logique sous-jacente aux bases de données repose sur l'algèbre relationnel lui-même basé sur la théorie des ensembles.
- Du fait de l'existence d'optimiseurs, la manière d'écrire une requête à peu d'influence en général sur la qualité de son exécution. Dans un premier temps il vaut mieux se consacrer à la résolution du problème que d'essayer de savoir si la clause "bidule" est plus gourmande en ressource lors de son exécution que la clause "truc". Néanmoins l'optimisation de l'écriture des requêtes sera abordée dans un article de la série.

  Le langage est insensible à la casse ce qui signifie que les commandes peuvent être tapées en minuscule ou majuscule invariablement. En revanche, les données sont sensibles à la casse ("case-sensitive"). Donc si dans la base, mon nom est sous la forme Etien, et que vous le cherchez sous la forme etien, vous ne le trouverez pas. Une chaine de caractères s'écrit entre simple quotes en SQL. Attention, ce qui est écrit entre double quote est considéré comme un identifier (noms de tables, de colonnes...) sensible à la casse.

1.5 Les types de données

Dernier point que nous allons aborder dans ce premier chapitre, les différents types de données spécifiés par SQL.

1.5.1 Types alphanumériques

- `CHARACTER` (ou `CHAR`) : valeurs alpha de longueur fixe
- `CHARACTER VARYING` (ou `VARCHAR` ou `CHAR VARYING`) : valeur alpha de longueur maximale fixée que l'on doit spécifier. Dans le cadre de ce cours, on pourra utiliser `VARCHAR` pour les types alphanumériques en précisant bien la taille maximale de la chaîne.

  Exemple : `nom_client VARCHAR(32)`

1.5.2 Types numériques

- `NUMERIC` (ou `DECIMAL` ou `DEC`) : nombre décimal
- `INTEGER` (ou `INT`): entier long
- `SMALLINT` : entier court. Dans les bases de données réelles, on évitera de choisir ce type pour un attribut clé, car ce type est réservé aux entier de 0 à 127.

- `FLOAT` : réel à virgule flottante dont la représentation est binaire à échelle et précision obligatoire
- `REAL` : réel à virgule flottante dont la représentation est binaire, de faible précision
- `DOUBLE PRECISION` : réel à virgule flottante dont la représentation est binaire, de grande précision
- `BIT` : chaîne de bit de longueur fixe
- `BIT VARYING` : chaîne de bit de longueur maximale

Pour les types réels NUMERIC, DECIMAL, DEC et FLOAT, on doit spécifier le nombre de chiffres significatifs et la précision des décimales après la virgule.

Exemple : `NUMERIC (15,2)` signifie que le nombre comportera au plus 15 chiffres significatifs dont deux décimales.

1.5.3 Types temporels

- `DATE : date du calendrier grégorien`
- `TIME : temps sur 24 heures`
- `TIMESTAMP : combiné date temps`
- `INTERVAL : intervalle de date / temps`

ATTENTION : Le standard ISO adopté pour le SQL repose sur le format AAAA-MM-JJ. Il est ainsi valable jusqu'en l'an 9999... ou AAAA est l'année sur 4 chiffres, MM le mois sur deux chiffres, et JJ le jour. Pour l'heure le format ISO est hh:mm:ss.nnn (n étant le nombre de millisecondes). Il est possible de changer de format.

Exemple : `2018-03-26 22:54:28.123` est le 26 mars 2018 à 22h 54m, 28s et 123 millisecondes.

Mais peu de moteurs de requêtes l'implémente de manière aussi formelle...

1.5.4 Types " BLOBS " (hors du standard SQL 2)

Longueur maximale prédéterminée, donnée de type binaire, texte long voire formaté, structure interprétable directement par le SGBDR ou indirectement par add-on externes (image, son, vidéo...). Attention : ne sont pas normalisés !

On trouve souvent les éléments suivants :

- `TEXT` : suite longue de caractères de longueur indéterminé

- `IMAGE` : stockage d'image dans un format déterminé
- `OLE` : stockage d'objet OLE (Windows)

1.5.5 Autres types courants, hors norme SQL 92

- `BOOLEAN` (ou `LOGICAL`) : curieusement le type logique (ou encore booléen) est absent de la norme. On peut en comprendre aisément les raisons... La pure logique booléenne ne saurait être respectée à cause de la possibilité offerte par SQL de gérer les valeurs nulles. On aurait donc affaire à une logique dite " 3 états " qui n'aurait plus rien de l'algèbre booléenne. La norme passe donc sous silence ce problème et laisse à chaque éditeur de SGBDR le soin de concevoir ou non un booléen " à sa manière ". On peut par exemple implémenter un tel type de données, en utilisant une colonne de type caractère longueur 1, non nul et restreint à deux valeurs (V / F ou encore T / F).
- `MONEY` : est un sous type du type `NUMERIC` avec une échelle maximale et une précision de deux chiffres après la virgule.
- `BYTES` (ou `BINARY`) : Type binaire (octets) de longueur devant être précisée. Permet par exemple le stockage d'un code barre.

2

Langage de définition des données

Syntaxe PostgreSQL (très simplifiée)

- pour spécifier qu'un élément est facultatif mais présent en 0 ou 1 occurrence, on utilise [ ]
- pour spécifier qu'un élément est facultatif mais présent en 0 ou plusieurs occurrences, on utilise {}
- pour spécifier qu'un élément est répété plusieurs fois on utilise, *
- pour spécifier un choix, on utilise |

Dans ce chapitre, on se focalise sur la définition de table, mais il est également possible de créer / supprimer / modifier des domaines (exemple entiers positifs) ou de créer / supprimer des vues (requêtes `SELECT` stockée et nommée dans la base).

2.1 Création de table - commande `CREATE TABLE`

Rappel : chaque relation / table est définie par un nom de relation et une liste d'attributs. Chaque attribut est défini par un nom et un type de données.

Dans la plupart des SGBD, le nom de la table doit commencer par une lettre et posséder 254 colonnes maximum par table.

2.1.1 Grammaire de la commande `CREATE TABLE`

La commande `CREATE TABLE` est suivi du nom de la table `tableName` puis entre parenthèses, la description des différents éléments ainsi que les contraintes de tables.

```
CREATE TABLE tableName (
   tableElement {, tableElement}
   {, tableConstraint } )
```

La table est constituée d'au moins un `tableElement`. Avec

```
< tableElement> ::= <columnName> <type> [ <columnConstraint>*]
```

Un `tableElement` est donc un nom de colonne suivi d'un type et éventuellement des contraintes de colonne. Comme expliqué à la section 1.5, le type de la colonne s'écrit comme suit :

```
type ::= VARCHAR <longueur> | INT | REAL | DATE ...
```

2.1.2 Contrainte d'attribut

Une contrainte d'attribut n'implique qu'un seul attribut.

```
columnConstraint ::= [ CONSTRAINT constraintName ] [NOT NULL],
[UNIQUE], [DEFAULT <valeurOuExpression>], [CHECK <valeurOuExpression>],
[PRIMARY KEY | REFERENCES tableName [ ( column [, ...  ]  )  ]]
```

Elle permet de spécifier par exemple que :

- la valeur `NULL` est impossible pour cet attribut (`NOT NULL`)
- chaque valeur de cet attribut est unique (`UNIQUE`)
- une valeur par défaut (`DEFAULT`)
- la valeur de l'attribut doit vérifier une condition (`CHECK`). Cette condition est vérifiée à chaque modification ou insertion. Cette contrainte peut éventuellement impliquer plusieurs attributs (par exemple pour comparer que la date de début est bien antérieure à la date de fin). Dans ce cas, elle devient une contrainte de table, et est alors spécifiée après la déclaration des attributs avec les autres contraintes de table.
- l'attribut est clé primaire (`PRIMARY KEY`)
- une contrainte référencielle : l'attribut est une clé étrangère référençant un autre attribut dans une autre table (`REFERENCES <tableName> [(columnName)]`)

Les colonnes de la clé primaire sont automatiquement `UNIQUE NOT NULL`.

2.1.3 Contrainte de table

Une contrainte de table concerne plusieurs attributs. En particulier, c'est de cette façon qu'on définit les clés composées.

```
tableConstraint::= [ CONSTRAINT constraintName ] PRIMARY KEY
( columnName [, ...  ]  )  | FOREIGN KEY ( columnName [, ...  ]
)  REFERENCES tableName [ ( column [, ...  ]  )  ]
```

Quelques précisions sur les clés étrangères :

- Il peut y avoir plusieurs clés étrangères dans une même table.
- Une clé étrangère peut être NULL voire non unique.
- possibilité de modifer automatiquement les clés étrangères en cas de changement de la clé primaire associée :

  ON DELETE { RESTRICT | CASCADE | SET NULL | SET DEFAULT}

  ON UPDATE { RESTRICT | CASCADE | SET NULL | SET DEFAULT}

  - RESTRICT est la valeur par défaut si rien n'est précisé. Si l'on essaye de supprimer ou de mettre à jour une valeur référencée par une clé étrangère, l'action est avortée et on obtient une erreur.
  - Le comportement CASCADE est le plus risqué (et le plus violent !). En effet, ON DELETE CASCADE supprime purement et simplement toutes les lignes qui référençaient la valeur supprimée ! ON UPDATE CASCADE remplace la valeur modifiée par la nouvelle valeur dans toutes les tables référençant l'attribut clé dont la valeur est modifée.
  - Si on choisit SET NULL, alors tout simplement, NULL est substitué aux valeurs dont la référence est supprimée ou modifiée.
  - Si on choisit SET DEFAULT alors la valeur par défaut est substituée aux valeurs dont la référence est supprimée ou modifiée. Bien évidemment, il faut avoir précisé une valeur par défaut, sinon, ce sera NULL.

Attention : ces déclencheurs agissent uniquement en cas d'opération sur la clé primaire associée !

2.1.4 Exemple

On illustre ici les différentes façons de spécifier qu'un attribut est la clé d'une relation, avec une contrainte d'attribut, une contrainte de table, avec nommage de la contrainte... On remarque que toute instruction se termine par ;.

L'ordre de définition des tables est important uniquement car il n'est pas possible de référencer une table ou une colonne qui n'a pas été préalablement définie. L'ordre de définition des colonnes au sein d'une table n'a aucune importance. Néanmoins, le(s) premier(s) attribut(s) correspond(ent) souvent à la clé.

2.2 Modification d'une table - commande ALTER

Il est possible de modifier une table déjà créée. Par exemple :

```
CREATE TABLE utilisateur (
num_u INTEGER PRIMARY KEY,
nom VARCHAR(30),
prenom VARCHAR(30) );

CREATE TABLE auteur (
num_a INTEGER,
nom VARCHAR(30),
CONSTRAINT cleAuteur PRIMARY KEY (num_a) );

CREATE TABLE emprunte (
num_l INTEGER REFERENCES livre,
num_u INTEGER REFERENCES utilisateur,
PRIMARY KEY (num_l, num_u));
```

```
CREATE TABLE editeur (
num_e INTEGER PRIMARY KEY,
nom VARCHAR(30),
adresse VARCHAR(30),
codePostal integer,
ville VARCHAR(30));

CREATE TABLE livre (
num_l INTEGER,
titre text,
nAuteur INTEGER REFERENCES auteur
PRIMARY KEY (num_l));
```

- pour ajouter une colonne `ALTER TABLE nomTable ADD [ COLUMN ] nomColonne type`
- pour renommer une colonne `ALTER TABLE nomTable RENAME [ COLUMN ] nomColonne TO nouveauNom`. Attention, aux erreurs si la colonne est référencée par exemple dans une contrainte d'intégrité.
- pour renommer la table elle même `ALTER TABLE nomTable RENAME TO nomTable`. Attention, aux erreurs si la table est référencée par exemple dans une contrainte d'intégrité.
- pour supprimer une colonne `ALTER TABLE nomTable DROP [ COLUMN] nomColonne`. Attention, aux erreurs si la colonne est référencée par exemple dans une contrainte d'intégrité.
- pour changer le propriétaire de la table `ALTER TABLE nomTable OWNER to nouveauProprietaire`.
- pour ajouter une contrainte de table et en particulier une clé étrangère `ALTER TABLE nomTable ADD tableContrainte`. C'est de cette façon qu'on fera une clé étrangère sur la table elle même ou un cycle sur des clés étrangères, puisque la table doit être créée pour être référencée ce qui n'est pas possible en cas de cycle avec les méthodes vues précédemment.
- ...(tout ce qu'il est possible de faire à la construction dans la commande `CREATE` peut l'être également une fois la table créée avec la commande `ALTER`).

2.3 Suppression d'une table - commande DROP

Bien évidemment, seule une table préalablement créée peut être détruite. L'option IF EXISTS permet de ne pas provoquer d'erreur si on cherche à supprimer une table qui n'existe pas. Par ailleurs, par défaut, il n'est pas possible de supprimer une entité et donc une table référencée par une autre. L'ordre de suppression des tables est donc l'inverse de celui de création sauf si elles ont été modifiées pour ajouter des contraintes référentielles. Sinon, il est possible d'utiliser l'option CASCADE.

```
DROP TABLE [ IF EXISTS ] <tableName> [ CASCADE | RESTRICT ]
```

3

Langage de manipulation des données

3.1 Insertion de lignes - commande `INSERT`

Pour insérer des lignes dans une table, on utilise la commande `INSERT`. Syntaxe (simplifiée) :

```
INSERT INTO tableName [ ( columnName {, ...} ) ]
VALUES ( expression , ... ) | SELECT query
```

Les valeurs doivent être fournies dans l'ordre de déclaration des attributs de la liste ou, s'il n'y en a pas, celui défini à la création. Si la liste d'attributs est incomplète, les attributs non spécifiés sont insérés avec des valeurs `NULL`.

Ci dessous quelques exemples :

- `INSERT INTO auteur VALUES (1, "Uderzo");` pour insérer une ligne complète.
- `INSERT INTO auteur (nom, num_a) VALUES ("Franquin", 2);` pour insérer une ligne complète en spécifiant les colonnes.
- `INSERT INTO livre(num_l, titre) VALUES (1, "L'Odysée d'Astérix");` pour insérer une ligne incomplète.

Il est aussi possible d'ajouter des lignes en fonction du résultat d'une requête `SELECT`. Bien évidemment dans ce cas, il faut que le résultat de la requête `SELECT` ait autant de colonne que ceux de la table dans laquelle on insère les lignes (ou de ceux spécifiés) et que les types des colonnes dans les deux cas soient compatibles.

```
INSERT INTO livre SELECT ...
```

3.2 Suppression de lignes - commande `DELETE`

Attention à ne pas confondre `DELETE` et `DROP`.

Syntaxe : `DELETE FROM tableName [ WHERE condition ]`

La condition `WHERE` est aussi complexe que celle de la commande `SELECT` que nous allons voir dans le chapitre suivant.

Exemples :

- `DELETE FROM utilisateur ;` comme il n'y a pas de condition, toutes les lignes de la table seront supprimées.
- `DELETE FROM livre where num_l=1 ;` Seules les lignes vérifiant la condition seront supprimées.

Ici comme la condition concerne une valeur de la clé, il n'y aura qu'une seule ligne supprimée dans la table `livre`. Si on avait spécifié `ON DELETE CASCADE` dans la définition des tables, on peut supprimer beaucoup de choses.

3.3 Modification de lignes - commande `UPDATE`

Syntaxe : `UPDATE tableName SET col = expression {, col = expression} [ WHERE condition ]`

Exemples :

- `UPDATE livre SET auteur=1 WHERE num_l=1;` modifie une colonne, pour une ligne.
- `UPDATE auteur SET nom="Victor Hugo", num_a=4 where num_a=3;` modifie plusieurs colonnes pour une ligne :
- `UPDATE personnel SET salaire=salaire+0.10*salaire;` modifie toutes les lignes de la table car aucune condition n'est spécifiée.

L'expression caractérisant la modification à effectuer peut être une constante, une expression arithmétique ou le résultat d'un `SELECT`.

4

Langage d'interrogation de la base - équivalence avec l'algèbre relationnelle

Les requêtes d'interrogation ou de consultation de la base représentent la majorité des requêtes. Une seule commande, `SELECT` est utilisée. Elle encapsule complètement l'algèbre relationnelle.

4.1 Exemple

auteur

num_a	nom
1	Albert Uderzo
2	Victor Hugo
3	J.K. Rowling

editeur

num_e	nom	ville
1	Albert-René	Bruxelles
2	Gallimard	Paris
3	Folio	Paris

livre

num_l	titre	auteur
1	Le fils d'Asterix	1
2	Les misérables	2
3	Notre dame de Paris	2
4	Harry Potter à l'école des sorciers	3
5	Harry Potter et la chambre des secrets	3

utilisateur

num_u	nom	prenom
1	Caron	Olivier
2	Kessaci	Marie-Éléonore
3	Janot	Stéphane
4	Etien	Anne

emprunt

num_l	num_u
1	1
2	4
4	1

editePar

num_l	num_e	dateEdition
1	1	1998-03-24
2	3	1940-02-02
3	2	1967-06-12
4	2	1999-03-01
5	2	2000-02-01
3	3	2015-06-02
2	2	2018-08-12

4.2 Syntaxe partielle de la commande `SELECT`

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ] * |
expression [ AS outputName ] [, ...]
   [ FROM fromItem [, ...] ]
   [ WHERE condition ]
   [ GROUP BY expression [, ...] ]
   [ HAVING condition [, ...] ]
   [ { UNION | INTERSECT | EXCEPT [ ALL ] } select ]
   [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...]
]
   [ FOR UPDATE [ OF tablename [, ...] ] ]
   [ LIMIT { count | ALL } [ { OFFSET | , } start ]]
```

4.3 Consultation simple d'une table

Syntaxe : `SELECT col1, col2, ..., coln FROM nomTable`
Variante usuelle pour consulter toute la table : `SELECT * FROM nomTable`

Exemple : `SELECT * FROM utilisateur ;`

num_u	nom	prenom
1	Caron	Olivier
2	Kessaci	Marie-Éléonore
3	Janot	Stéphane
4	Etien	Anne

4.4 Expression d'une projection

Syntaxe : `SELECT` col_{i1}`,` col_{i2}`, ...,` col_{ip} `FROM nomTable`
Exemple : `SELECT prenom, nom FROM utilisateur ;`

prenom	prenom
Olivier	Caron
Marie-Éléonore	Kessaci
Stéphane	Janot
Anne	Etien

On peut inverser l'ordre de présentation ce qui n'a aucun impact sur le calcul. Si on veut l'équivalent de l'algèbre relationnelle pure : il faut ajouter la clause `DISTINCT` (qui permet de supprimer les doublons)

4.5 Expression d'une restriction

Introduction clause `WHERE` et utilisation des opérateurs booléens `and`, `or` et `not` pour séparer les différentes conditions.

Exemple :

```
SELECT * FROM livre WHERE auteur=2;
```

num_l	titre	auteur
2	Les misérables	2
3	Notre dame de Paris	2

4.6 Expression d'une jointure

Il existe plusieurs façons d'exprimer une jointure.

4.6.1 Expression d'un produit cartésien

De la même façon qu'en algèbre relationnelle, le produit cartésien n'a pas de sens en soi.

Utilisation de la clause `FROM`

Exemple : `SELECT * FROM utilisateur, livre ;` ... (20 lignes)

```
SELECT distinct editeur.nom AS nomEditeur , auteur.nom AS nomAuteur
FROM editeur, auteur;
```

Dans ce deuxième exemple, il est impératif dans la clause `SELECT` de préfixer le nom des colonnes par le nom de la table afin d'éviter toute ambiguité. En effet, les deux tables `editeur` et `auteur` ont toutes les deux une colonne `nom`. Si on ne précise par la table le système de gestion de base de données ne sait pas de laquelle on parle. Le vrai nom d'une colonne s'écrit `nomTable.nomColonne` Le nom de la table peut-être ommis quand il n'y a aucune ambiguïté. Il peut être fastidieux d'utiliser le nom complet des tables. Il est donc possible d'utiliser un renommage des tables, afin d'y faire référence dans une autre clause.

```
SELECT distinct e.nom AS nomEditeur , a.nom AS nomAuteur FROM
editeur e, auteur a;
```

Cette requête est complètement équivalente à la précédente.

Note : dans la clause `FROM`, contrairement à la clause `SELECT`, le mot clé `AS` n'est pas obligatoire pour le renommage. On a donc dans la clause `FROM` : `nomTable alias`. Bien entendu, l'alias doit être unique au sein de la requête.

4.6.2 Expression de la jointure - équivalent à l'algèbre

De la même faqn qu'en algèbre relationnelle, la jointure a pour but de relier avec cohérence plusieurs tables. Souvent, il s'agit de relier les clés étrangères d'une table, avec les clé primaires qu'elles représentent dans les autres tables. On utilise alors un prédicat de jointure qui comme en algèbre s'écrit `colonne opérateur colonne`. Il apparaît dans la clause `WHERE`.

Exemple : Lister les titres des livres et le nom de leur auteur.

```
SELECT titre, nom FROM auteur, livre
   WHERE auteur.num_a = livre.auteur;
```

titre	nom
Le fils d'Asterix	Albert Uderzo
Les misérables	Victor Hugo
Notre dame de Paris	Victor Hugo
Harry Potter à l'école des sorciers	J.K. Rowling
Harry Potter et la chambre des secrets	J.K. Rowling

Bien évidemment, on peut faire plusieurs jointures. On remarquera que lorsqu'on a n tables, on a **au moins** $n-1$ jointures.

Exemple : Lister les titres des livres édités chez Folio.

```
SELECT titre FROM editeur e, livre l, editePar ep
   WHERE e.num_e = ep.num_e AND l.num_l = ep.num_l
      AND e.nom = 'Folio';
```

titre
Les misérables
Notre dame de Paris

Il est également possible de faire une jointure sur la même table. Bien évidemment comme on utilise deux instances de la même table, on est obligé de les renommer.

Exemple : liste de couples de livres ayant le même auteur

```
SELECT l1.titre, l2.titre FROM livre l1, livre l2
   WHERE l1.auteur=l2.auteur AND l1.titre > l2.titre;
```

Nous avons ici deux prédicats de jointure. Le premier permet d'exprimer que les deux livres ont le même auteur. Le deuxième permet de ne pas garder en résultat les instances de livres (car un livre dans la table `l1` a le même auteur que ce même livre dans la table `l2`), ni d'obtenir les résultats en double.

titre	titre
Notre dame de Paris	Les misérables
Harry Potter et la chambre des secrets	Harry Potter à l'école des sorciers

4.6.3 Expression de la jointure - apport de SQL2

SQL2 permet la séparation des critères de jointure et de restriction Les jointures sont exprimées dans la clause `FROM`.

Jointure interne Cette jointure est dite *Jointure interne* et s'effectue avec le mot clé `INNER JOIN`. On obtient alors exactement le même résultat qu'en notation SQL-89.

Pour les exemples suivants, on ajoute :

```
INSERT INTO livre VALUES (6, Le livre inconnu, NULL) ;
INSERT INTO auteur VALUES (4, Paltoquet) ;
```

Afin de lister les titres des livres et le nom de leur auteur et de façon strictement équivalente à

```
SELECT titre, nom FROM auteur, livre
   WHERE auteur.num_a = livre.auteur;
```

on a

```
SELECT titre, nom FROM livre INNER JOIN auteur
   ON livre.auteur=auteur.num_a ;
```

La jointure interne ne fournit en résultat **que** les lignes qui vérifient le critère de jointure. Souvent, on souhaite que même les lignes sans correspondance soit aussi présentes ! Il existe pour cela la notion de table directrice

titre	nom
Le fils d'Asterix	Albert Uderzo
Les misérables	Victor Hugo
Notre dame de Paris	Victor Hugo
Harry Potter à l'école des sorciers	J.K. Rowling
Harry Potter et la chambre des secrets	J.K. Rowling

afin de garantir que tous les enregistrements apparaissent dans la table résultante.

Les trois syntaxes suivantes sont équivalentes.

```
SELECT * FROM auteur INNER JOIN livre ON livre.auteur=auteur.num_a;

SELECT * FROM auteur INNER JOIN livre USING (num_a, auteur);
SELECT * FROM t1 NATURAL INNER JOIN t2;
```

La commande `NATURAL [INNER] JOIN` permet de faire une jointure naturelle entre 2 tables. Cette jointure s'effectue à la condition qu'il y ai des colonnes du même nom et de même type dans les 2 tables. Le résultat d'une jointure naturelle est la création d'un tableau avec autant de lignes qu'il y a de paires correspondant à l'association des colonnes de même nom.

Jointure externe `LEFT OUTER JOIN` permet d'indiquer que c'est la table de gauche qui est directrice. Les colonnes manquantes sont complétées à `NULL`.

```
SELECT titre, nom FROM livre LEFT OUTER JOIN auteur
   ON livre.auteur=auteur.num_a ;
```

titre	nom
Le fils d'Asterix	Albert Uderzo
Les misérables	Victor Hugo
Notre dame de Paris	Victor Hugo
Harry Potter à l'école des sorciers	J.K. Rowling
Harry Potter et la chambre des secrets	J.K. Rowling
Le livre inconnu	

Il est possible de préciser que c'est la table de gauche qui est directrice.

```
SELECT titre, nom FROM livre RIGHT OUTER JOIN auteur
   ON livre.auteur=auteur.num_a ;
```

Et également une jointue externe complète où chaque table est directrice.

```
SELECT titre, nom FROM livre FULL OUTER JOIN auteur
   ON livre.auteur=auteur.num_a ;
```

Note : `INNER` et `OUTER` sont facultatifs.

titre	nom
Le fils d'Asterix	Albert Uderzo
Les misérables	Victor Hugo
Notre dame de Paris	Victor Hugo
Harry Potter à l'école des sorciers	J.K. Rowling
Harry Potter et la chambre des secrets	J.K. Rowling
	Paltoquet

titre	nom
Le fils d'Asterix	Albert Uderzo
Les misérables	Victor Hugo
Notre dame de Paris	Victor Hugo
Harry Potter à l'école des sorciers	J.K. Rowling
Harry Potter et la chambre des secrets	J.K. Rowling
Le livre inconnu	
	Paltoquet

4.6.4 Expression d'une union - `UNION`

Syntaxe : `<requête> UNION <requête>`

Les résultats des deux requêtes doivent avoir la même structure (même nombre de colonnes, même type et dans le même ordre).

Exemple : Liste des noms d'auteur et d'éditeur.

```
SELECT nom FROM auteur UNION SELECT nom FROM editeur ;
```

nom
Albert Uderzo
Victor Hugo
Albert-René
J.K. Rowling
Folio
Gallimard

Note : C'est la première fois dans ce cours qu'on voit l'utilisation d'une sous-requête. La portée des tables ou des variables est celle de la requête dans laquelle elle est définie. Ici, il n'y a donc pas de confusion possible entre le champ `nom` de la table `auteur` et le champ `nom` de la table `editeur`.

4.6.5 Expression d'une intersection - `INTERSECT`

Syntaxe : `<requête> INTERSECT <requête>`

Les résultats des deux requêtes doivent avoir la même structure (même nombre de colonnes, même type et dans le même ordre).

Exemple : Liste des livres publiés chez Folio et chez Gallimard.

```
SELECT num_l, titre FROM editeur e, livre l, editePar ep
   WHERE e.num_e = ep.num_e AND l.num_l = ep.num_l
      AND e.nom = 'Folio'
INTERSECT
SELECT num_l, titre FROM editeur e, livre l, editePar ep
   WHERE e.num_e = ep.num_e AND l.num_l = ep.num_l
      AND e.nom = 'Gallimard';
```

num_l	titre
2	Les misérables
3	Notre dame de Paris

Note : Il est possible dans chacune des deux sous-requêtes de renommer la table `editeur` par `e`, puisque les deux requêtes sont indépendantes l'une de l'autre de part la syntaxe de l'opérateur `INTERSECT`.

La requête ci-dessus est équivalente à :

```
SELECT num_l, titre FROM editeur e, livre l, editePar ep
   WHERE e.num_e = ep.num_e AND l.num_l = ep.num_l
      AND e.nom = 'Folio'
      AND l.num_l IN
         (SELECT num_l FROM editeur e1, livre l1, editePar ep1
               WHERE e1.num_e = ep1.num_e AND l1.num_l = ep1.num_l
                  AND e1.nom = 'Gallimard');
```

ou à

```
SELECT num_l, titre FROM editeur e, livre l, editePar ep
   WHERE e.num_e = ep.num_e AND l.num_l = ep.num_l
      AND e.nom = 'Folio'
      AND EXISTS
         (SELECT * FROM editeur e1, livre l1, editePar ep1
               WHERE e1.num_e = ep1.num_e AND l1.num_l = ep1.num_l
                  AND e1.nom = 'Gallimard'
                  AND l.num_l = l1.num_l);
```

Attention à la structure du `IN` et du `EXISTS`.

Pour le `IN`, on a dans la clause `WHERE`: `attribut1 IN (SELECT attribut2 ...` avec `attribut1` et `attribut2` deux attributs ou ensemble d'attributs de même structure (nombre, type et ordre en cas d'ensemble identiques).

Pour le `EXISTS`, on a dans la clause `WHERE`: `EXISTS (SELECT * ...WHERE attribut1 = attribut2` avec `attribut1` qui provient de la première sous-requête et `attribut2` de la deuxième. Une fois encore `attribut1` et `attribut2` sont deux attributs ou ensemble d'attributs de même structure (nombre, type et ordre en cas d'ensemble identiques).

Note : Dans chacune des deux requêtes ci-dessus, les sous-requêtes sont imbriquées. Il n'est donc pas possible de renommer la table `editeur` par `e` dans la sous requête. La portée du renommage de la table `editeur` par `e`

est celle de la requête dans laquelle elle est définie soit ici la totalité de la requête. En effet, cette dernière inclut la sous requête, qui du coup n'est pas indépendante contrairement au cas de la requête utilisant l'opérateur `INTERSECT`.

4.6.6 Expression d'une différence - `EXCEPT`

Syntaxe : `<requête> EXCEPT <requête>`

Les résultats des deux requêtes doivent avoir la même structure (c'est à dire, même nombre de colonnes, même type et dans le même ordre).

Exemple : Liste des livres publiés chez Gallimard mais pas chez Folio.

```
SELECT num_l, titre FROM editeur e, livre l, editePar ep
   WHERE e.num_e = ep.num_e AND l.num_l = ep.num_l
      AND e.nom = 'Gallimard'
EXCEPT
SELECT num_l, titre FROM editeur e, livre l, editePar ep
   WHERE e.num_e = ep.num_e AND l.num_l = ep.num_l
      AND e.nom = 'Folio';
```

num_l	titre
4	Harry Potter à l'école des sorciers
5	Harry Potter et la chambre des secrets

La requête ci-dessus est équivalente à :

```
SELECT num_l, titre FROM editeur e, livre l, editePar ep
   WHERE e.num_e = ep.num_e AND l.num_l = ep.num_l
      AND e.nom = 'Gallimard'
      AND l.num_l NOT IN
         (SELECT num_l FROM editeur e1, livre l1, editePar ep1
               WHERE e1.num_e = ep1.num_e AND l1.num_l = ep1.num_l
                  AND e1.nom = 'Folio');
```

ou à

```
SELECT num_l, titre FROM editeur e, livre l, editePar ep
   WHERE e.num_e = ep.num_e AND l.num_l = ep.num_l
      AND e.nom = 'Gallimard'
      AND NOT EXISTS
         (SELECT * FROM editeur e1, livre l1, editePar ep1
               WHERE e1.num_e = ep1.num_e AND l1.num_l = ep1.num_l
                  AND e1.nom = 'Folio'
                  AND l.num_l = l1.num_l);
```

Attention à la structure du `NOT IN` et du `NOT EXISTS`.

Pour le `NOT IN`, on a dans la clause `WHERE`: `attribut1 NOT IN (SELECT attribut2 ...` avec `attribut1` et `attribut2` deux attributs ou ensemble

d'attributs de même structure (nombre, type et ordre en cas d'ensemble identiques).

Pour le `NOT EXISTS`, on a dans la clause `WHERE`: `NOT EXISTS (SELECT * ...WHERE attribut1 = attribut2` avec `attribut1` qui provient de la première sous-requête et `attribut2` de la deuxième. Une fois encore `attribut1` et `attribut2` sont deux attributs ou ensemble d'attributs de même structure (nombre, type et ordre en cas d'ensemble identiques).

4.7 Expression d'une division - deux `NOT EXISTS`

Principe : Ensemble des éléments X pour lesquels, il n'existe pas de Y qui ne soient pas reliés à X. (avec "qui ne soient pas relié à X" = pour lesquels il n'existe pas de relation entre X et Y.

Syntaxe :

```
SELECT
FROM
WHERE
NOT EXISTS
```
} *ce que l'on veut en sortie*

```
(SELECT
FROM
WHERE
NOT EXISTS
```
} *ce par quoi on divise*

```
(SELECT
FROM
WHERE...));
```
} *ce qui permet de faire le lien*

Exemple : Liste des utilisateurs ayant emprunté tous les livres.

En sortie on veut des utilisateurs. On divise par des livres. Et la table `emprunt` permet de faire le lien.

```
SELECT nom, prenom
FROM utilisateur u
WHERE
NOT EXISTS
   ( SELECT *
   FROM livre l
   WHERE
   NOT EXISTS
      ( SELECT *
      FROM emprunt e
      WHERE e.num_l = l.num_l
      AND e.num_u = u.num_u ));
```

Il est également possible d'ajouter des conditions sur les utilisateurs ou les livres. Dans le premier cas, la restriction se trouvera dans le premier `WHERE` juste avant le premier `NOT EXISTS`. On n'oubliera pas alors d'ajouter un `AND` avant le `NOT EXISTS`. Dans le second cas, la restriction se trouvera dans le deuxième `WHERE` juste avant le deuxième `NOT EXISTS`. On n'oubliera pas là encore d'ajouter un `AND` avant le `NOT EXISTS`. La dernière requête est toujours la même (même s'il y a des restrictions) et ne contient que les tables qui permettent de faire le lien avec les sous requêtes précédentes.

Note : La différence, l'union, la division sont des exemples d'utilisation de requêtes `SELECT` imbriquées. En fait, il est possible d'imbriquer une requête `SELECT` dans à peu près n'importe quelle clause `FROM`, `WHERE`...

5

Langage d'interrogation de la base - Apport du SQL à l'algèbre relationnelle

5.1 Traîtement des chaînes de caractères

5.1.1 Opérateur `LIKE`

Caractères spéciaux : % remplace de 0 à plusieurs caractères et ? remplace exactement un caractère.

Exemple : Tous les titres dont le titre commence par un H.

```
SELECT DISTINCT titre FROM livre WHERE titre LIKE 'H%';
```

titre
Harry Potter à l'école des sorciers
Harry Potter et la chambre des secrets

5.1.2 Opérateur de comparaison et de concaténation

Comparaison Il existe plusieurs opérateurs de comparaison sur les chaînes de caractères qui sont aussi applicables à `INTEGER`, `DATE` ou plus généralement à tout type sur lequel un ordre est défini.

Opérateurs disponibles : =, <>, >,<, >=, <= (ordre lexicographique)

Il est aussi possible de comparer des chaînes en utilisant la clause `BETWEEN`. Cette clause permet de vérifier si la valeur d'un attribut est comprise entre deux constantes

Exemple :

```
SELECT nom FROM utilisateur WHERE nom BETWEEN 'A%' and 'F%';
```

qui est équivalent à l'exemple suivant :

`SELECT nom FROM utilisateur WHERE nom >= '%' AND nom <= 'F%'`

De la même l'opérateur `BETWEEN` est applicable à tout type (integer, chaîne, date, ...) sur lequel un ordre est défini.

Concaténation Opérateur de concaténation ||.

5.1.3 Sémantique des différentes formes d'égalités

- `ville = 'Lille'` → égalité stricte
- `ville IN ('Lille','Lens')` → égalité avec valeur d'un ensemble
- `ville LIKE 'Li%'` → égalité approximative
- `ville SIMILAR TO '%(fr|uk)'` → égalité avec expression régulière
- `ville IS NULL` → test de méta-valeur

5.1.4 Fonction prédéfinies

Il existe des fonctions prédéfinies (ex : upper() pour mettre en majuscule ou lower() pour mettre en minuscule la chaîne de caractères passée en paramètre).

Exemple : `SELECT upper(prenom || || nom) FROM utilisateur;`

OLIVIER CARON
MARIE-ÉLÉONORE KESSACI
STÉPHANE JANOT
ANNE ETIEN

5.2 Présentation des données

De façon générale, la présentation des données n'a aucun impact sur le traitement algébrique des requêtes. L'ordre d'affichage des colonnes est celui spécifié dans la clause `SELECT` sinon c'est celui spécifié à la construction des tables. La clause distinct permet d'éviter les doublons. L'ordre d'affichage des lignes se fait grâce à la clause `ORDER BY`, cet ordre peut être multi-critères. Le $n$ième critère de tri est utilisé en cas d'égalité des précédents.

Syntaxe :

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

`ASC` signifie croissant, `DESC` décroissant. Enfin, il est possible d'utiliser un opérateur de comparaison particulier afin d'établir l'ordre.

Exemple : Affichage de l'ensemble du contenu de la table livre par ordre décroissant des auteurs et alphabétique croissant des titres.

`SELECT * FROM livre ORDER BY auteur DESC, titre ASC ;`

num_l	titre	auteur
4	Harry Potter à l'école des sorciers	3
5	Harry Potter et la chambre des secrets	3
2	Les misérables	2
3	Notre dame de Paris	2
1	Le fils d'Asterix	1

5.3 Opération de calcul

Les opérateurs arithmétiques : +,−, . . . s'applique sur des valeurs numériques ou des dates. Ils peuvent être appliqué dans les clauses `SELECT` ou `WHERE`. Exemple :

```
SELECT now()-dateEdition as duree, num_l FROM editePar ;
```

Au passage, on remarque dans cette requête :

- l'utilisation de la fonction `now()` qui donne la date actuelle
- le renommage de la première colonne en `duree` grâce au mot clé `AS`.

duree	num_l
1423 days 17:30:01	1
22658 days 16:30:01	2
12666 days 17:30:01	3
1081 days 17:30:01	4
744 days 17:30:01	5

5.4 Fonction de calcul ou agrégat

Une fonction de calcul ou agrégat est une fonction qui s'applique sur un ensemble de tuples (ou lignes) et qui renvoie une valeur unique. Le résultat est stocké dans une colonne correspondant au nom de la fonction (sauf si renommage). Contrairement aux fonctions classiques décrites à la section 5.1.4 qui renvoient une valeur par ligne, pour les fonctions agrégat, il y a toujours une seule ligne résultat.

Syntaxe : `nomFonction(nomColonne) ou nomFonction(*)`

Les fonctions standards sont `count, min, max, avg, sum` qui respectivement compte le nombre d'éléments, renvoie le minimum, le maximum ou la moyenne des valeurs ou calcule la somme des valeurs de la colonne.

Exemple avec renommage : Nombre de livres.

```
SELECT count(*) as nombre FROM livre;
```

nombre
5

Exemple sans renommage : `SELECT min(num_l), max(num_l), avg(num_l), sum(num_l) FROM livre;`

min	max	avg	sum
1	5	3.000	15

Cette requête n'a aucun sens en soit puisque calculer la moyenne des numéros de livre est inutile.

5.5 Calcul sur les groupes de lignes - commandes `GROUP BY` et `HAVING`

Il est possible de sélectionner des lignes pour appliquer un calcul grâce à la clause `GROUP BY`. Cette clause permet de partitionner la relation en sous-relations ayant les mêmes valeurs sur les attributs précisés on peut alors appliquer des fonctions agrégat à chaque sous-relation. On aura un résultat par sous-relation.

`SELECT ...FROM livre GROUP BY auteur;`

	auteur	titre	num_l
sous-relation →	a	x	2
	a	y	1
sous-relation →	b	x	2
	b	z	5
	b	t	3
sous-relation →	b	x	2
	c	u	4

Exemple : Nombre de livres par auteurs.

`SELECT auteur, count(*) AS nbreParAuteur FROM livre GROUP BY auteur;`

Note : Toutes les colonnes figurant dans un `GROUP BY` doivent apparaître dans la clause `SELECT`. Et inversement, toutes les colonnes figurant dans la clause `SELECT` doivent apparaître dans la clause `GROUP BY` ou doivent dépendre fonctionnellement des attributs dans la clause `GROUP BY` (c'est-à-dire que pour une valeur des attributs dans la clause `GROUP BY`, il n'y a qu'une et une seule valeur des attributs de la clause `SELECT`)

Il est possible d'imposer une condition aux groupes formés par la clause `GROUP BY` grâce à la clause `HAVING`. Cette clause permet de poser une con-

auteur	nbreParAuteur
1	1
2	2
3	2

dition portant sur chacune des sous-relations générées par le `GROUP BY`. Les sous-relations ne vérifiant pas la condition sont écartées du résultat.

Exemple : Nombre de livres par auteurs ayant écrit plusieurs livres.

```
SELECT auteur, count(*) AS nbreParAuteur FROM livre GROUP BY
auteur HAVING count(*)>1;
```

Note : Ne pas confondre avec la clause `WHERE`. La clause `WHERE`, la condition doit être vérifiée par chaque ligne. La clause `HAVING`, la condition doit être vérifiée par chaque **groupe de lignes**.

auteur	nbreParAuteur
2	2
3	2

Note 2 : Il est possible à la place de la valeur 1 de mettre une requête imbriquée. Par exemple, nombre de livres par auteurs ayant écrit plus de livres que l'auteur numéro 2.

```
SELECT auteur, count(*) AS nbreParAuteur FROM livre GROUP BY
auteur HAVING count(*)> (SELECT count(*) FROM livre GROUP BY auteur
WHERE auteur = 2);
```

auteur	nbreParAuteur
3	2

Note 3 : Les **fonctions agrégat** ne peuvent apparaître que dans **les clauses `SELECT` ou `HAVING`**. Corollaire, elles ne peuvent pas apparaître dans la clause `WHERE`.