## 2. SQL

« Structured Query Language »
Le langage SQL d'interrogation et de manipulation de données suit la syntaxe de la norme SQL2 implantée dans la plupart des SGBDR.
SQL est un langage déclaratif qui permet d'interroger une base de données sans se soucier de la représentation interne des données, de leur localisation, des chemins d'accès ou des algorithmes nécessaires.
On peut l'utiliser de manière interactive mais aussi en association avec des interfaces graphiques, des outils de reporting ou des langages de programmation.
Rappel :

Station(**nomStation**, capacite, lieu, region, tarif)
Activite(*nomStation*, **libelle**, prix)
Client(**idClient**, nom, prenom, ville, region, solde)
Sejour(*idClient*, **nomStation, debut**, nbPlaces)

**Requêtes simples**

Sélections simples

```
SELECT …
FROM …
WHERE …
```

- FROM indique la ou les tables dans lesquelles on trouve les attributs utiles à la requête,
- SELECT indique la liste des attributs constituant le résultat (= projection),
- WHERE indique les conditions que doivent satisfaire les tuples de la base pour faire partie du résultat (= sélection).

Le résultat d'un ordre SQL est toujours une relation dont les attributs sont ceux spécifiés dans la clause SELECT. On peut donc considérer en première approche ce résultat comme un 'découpage' horizontal et vertical de la table indiquée dans le FROM, similaire à une utilisation combinée de la sélection et de la projection.
On peut renommer les attributs (en utilisant le mot-clé optionnel AS), appliquer des fonctions aux valeurs de chaque tuple et introduire des constantes.

```
SELECT libelle, prix/6,56 AS prixEnEuros
FROM Activite
WHERE nomStation = 'Santalba'
```

La spécification de clé permet d'éviter les doublons dans les relations stockées, mais ils peuvent apparaître dans le résultat d'une requête. Pour éviter d'obtenir deux tuples identiques, on utilise le mot-clé DISTINCT, mais cette opération peut être coûteuse.

```
SELECT DISTINCT libelle
FROM Activite
```

Il est possible de trier le résultat d'une requête avec la clause ORDER BY

```
SELECT *    (* : tous les attributs)
FROM Station
ORDER BY tarif, nomStation
```

Pour trier en ordre descendant, on utilise le mot-clé DESC après la liste des attributs.
Dans la clause WHERE, on spécifie une condition booléenne (AND, OR, NOT, =, <, <=, >, >=, <>) portant sur les attributs des relations du FROM. Pour obtenir une recherche par intervalle, on utilise le mot-clé BETWEEN.

Pour les chaînes de caractères, attention à la différence entre chaînes de longueur fixe et celles de longueur variable ; et attention à la différence majuscule/minuscule.
SQL fournit des options de recherche par motif à l'aide de la clause LIKE : '_' désigne n'importe quel caractère et '%' n'importe quelle chaîne.
Une date est spécifiée par le mot-clé DATE au format 'aaaa-mm-jj'

```
SELECT IdClient
FROM Sejour
WHERE debut BETWEEN DATE '1999-07-01' AND DATE '2006-07-01'
```

On admet que la valeur de certains attributs soit inconnue en utilisant le mot-clé NULL. On ne peut donc lui appliquer aucune des opérations ou comparaisons usuelles. Toutes les opérations appliquées à NULL retourne le résultat NULL. Toute comparaison avec NULL donne UNKNOWN.
TRUE = 1, FALSE = 0 et UNKNOWN = ½.
x AND y = min(x,y)
x OR y = max(x,y)
NOT x = 1-x
Les conditions exprimées dans une clause WHERE sont évaluées pour chaque tuple qui est conservé si cette évaluation donne TRUE. La présence d'une valeur nulle dans une comparaison a donc souvent le même effet que si cette comparaison échoue et renvoie FALSE. La présence de NULL peut avoir des effets surprenants.
NULL est un mot-clé pas une constante. Le prédicat pour tester l'absence de valeur dans une colonne est 'x IS NULL' (ou 'x IS NOT NULL'). Dans la mesure du possible, il faut éviter NULL en spécifiant la contrainte NOT NULL ou en donnant une valeur par défaut.

**Requêtes sur plusieurs tables**

- Jointure

On donne simplement la liste des tables concernées dans la clause FROM et on exprime les critères de rapprochement entre ces tables dans la clause WHERE.
On construit le produit cartésien des tables du FROM, en préfixant chaque attribut par le nom ou le synonyme de sa table pour éviter les ambiguïtés. Il n'y a alors plus qu'une seule table sur laquelle on interprète l'ordre SQL comme dans le cas d'une requête simple.

« Nom des clients avec le nom des stations où ils ont séjourné ? »

```
SELECT nom, station
FROM Client, Sejour
WHERE client.idClient = sejour.idClient
```

« Nom des clients habitant Paris, les stations où ils ont séjourné avec la date et le tarif hebdomadaire pour chaque station ? »

```
SELECT nom, nomStation, début, tarif
FROM Client, Sejour, Station
WHERE ville = 'Paris'
AND client.idClient = sejour.idClient
AND station.nomStation = sejour.nomStation
```

« Couples de stations situées dans la même région ? »

```
SELECT s1.nomStation, s2.nomStation
FROM Station s1, Station s2
WHERE s1.region = s2.region
```

Tout se passe comme si on devait faire la jointure entre deux versions de la table Station.

- Union, intersection et différence

On construit deux requêtes dont les résultats ont même arité et on les relie par un des mots-clé UNION, INTERSECT ou EXCEPT.

« Tous les noms des régions dans la base ? »

```
SELECT region FROM Station
UNION
SELECT region FROM Client
```

« Régions où on trouve à la fois des clients et des stations ? »

```
SELECT region FROM Station
INTERSECT
SELECT region FROM Client
```

« Régions où on trouve des stations mais pas des clients ? »

```
SELECT region FROM Station
EXCEPT
SELECT region FROM Client
```

La norme SQL2 spécifie que les doublons doivent être éliminés du résultat lors des trois opérations ensemblistes. Certains systèmes ne suivent pas cette norme. L'union ne peut être exprimé autrement qu'avec UNION. Par contre, INTERSECT peut être exprimé avec une jointure et la différence s'obtient plus aisément à l'aide des requêtes imbriquées.

**Requêtes imbriquées**

Rappel

Station(**nomStation**, capacite, lieu, region, tarif)
Activite(*nomStation*, **libelle**, prix)
Client(**idClient**, nom, prenom, ville, region, solde)
Sejour(*idClient*, **nomStation, debut**, nbPlaces)

- Conditions portant sur des relations

Cela permet d'offrir une alternative syntaxique à l'expression de jointures pour lesquelles le résultat est constitué avec les attributs d'une seule table, l'autre ne servant que pour exprimer des conditions.

« Nom des stations où ont séjourné des clients parisiens ? »

```
SELECT nomStation
FROM Client, Sejour
WHERE ville = 'Paris'
AND client.idClient = sejour.idClient
```

⇒

```
SELECT nomStation
FROM Sejour
WHERE idClient IN (SELECT idClient
                   FROM Client
                   WHERE ville = 'Paris')
```

IN exprime la condition d'appartenance de idClient à la relation formée par la requête imbriquée.
Sur une relation R construite avec une requête imbriquée, on peut exprimer les conditions suivantes :

- EXISTS R, renvoie True si R n'est pas vide, False sinon (ou NOT EXISTS).
- t IN R, renvoie True si le tuple t appartient à R, False sinon (ou NOT IN).
- v cmp ANY R, renvoie True si la comparaison de l'attribut v avec un moins un des tuples de R est vérifiée, False sinon, cmp ∈ {<, >, =, …}.
- v cmp ALL R, renvoie True si la comparaison de l'attribut v avec tous les tuples de R est vérifiée, False sinon, cmp ∈ {<, >, =, …}.

La différence s'exprime facilement avec NOT IN ou NOT EXISTS.

- Sous-requêtes corrélées

La sous-requête est basée sur une ou plusieurs valeurs issues des relations de la requête principale.
« Quels sont les clients (nom, prénom) qui ont séjourné à Santalba ? »

⇒

```
SELECT nom, prenom
FROM Client
WHERE EXISTS (SELECT *
              FROM Sejour
              WHERE nomStation = 'Santalba'
              AND client.idClient = sejour.idClient)
```

« Dans quelle station pratique-t-on une activité au même prix qu'à Santalba ? »

⇒

```
SELECT nomStation
FROM Activite A1
WHERE EXISTS (SELECT *
              FROM Activite A2
              WHERE nomStation = 'Santalba'
              AND A1.libelle = A2.libelle
              AND A1.prix = A2.prix)
```

**Fonctions d'agrégation**

Ces fonctions s'appliquent à une colonne en général de type numérique :

- COUNT qui compte le nombre de valeurs non nulles,
- MAX, MIN
- AVG qui calcule la moyenne des valeurs de la colonne,
- SUM qui effectue le cumul.

```
SELECT COUNT(nomStation), AVG(tarif), MIN(tarif), MAX(tarif)
FROM Station
```

« Combien de places a réservé M. Kerouac pour l'ensemble des séjours ? »

```
SELECT SUM(nbPlaces)
FROM Client, Sejour
WHERE nom = 'Kerouac'
AND client.idClient = sejour.idClient
```

On ne peut pas utiliser simultanément dans la clause SELECT des fonctions d'agrégation et des noms d'attributs, sauf dans le cas d'un GROUP BY.

La clause GROUP BY

Cette clause consiste à partitionner le résultat en groupes, en associant les tuples partageant la même valeur pour une ou plusieurs colonnes.
« Afficher les régions avec le nombre de stations »

```
SELECT region, COUNT(nomStation)
FROM Station
GROUP BY region
```

Ici, on construit un groupe pour chaque région.
« Donner le nombre de places réservées par client »

```
SELECT nom, SUM(nbPlaces)
FROM Client, Sejour
WHERE client.idClient=sejour.idClient
GROUP BY id, nom
```

La clause HAVING
On peut faire porter des conditions sur les groupes avec la clause HAVING. La clause WHERE ne peut exprimer des conditions que sur les tuples pris un à un.
« Nombre de places réservées par client, pour les clients ayant réservé plus de 10 places ? »

```
SELECT nom, SUM(nbPlaces)
FROM Client, Sejour
WHERE client.idClient=sejour.idClient
GROUP BY nom
HAVING SUM(nbPlaces) >= 10
```

**Mises à jour**

Insertion

```
INSERT INTO R(A1, A2, …, An) VALUES (v1, v2, …,vn)
```

R est le nom de la relation, $A_1$, $A_2$, …, $A_n$ sont les noms des attributs dans lesquels on souhaite placer une valeur (les autres attributs seront à NULL ou à la valeur par défaut) et $v_1$, $v_2$, …,$v_n$ sont les valeurs.

```
INSERT INTO Client (id, nom, prenom)
VALUES (40, 'Dupont', 'Jean')
```

Les attributs ville et region seront à NULL, solde sera à 0.
Il est également possible d'insérer dans une table le résultat d'une requête. La partie VALUES est remplacée par la requête.
Exemple : on crée une table Sites et on souhaite y copier les couples (lieu, region) déjà existant dans la table Station.

```
INSERT INTO Sites (lieu, region)
SELECT lieu, region FROM Station
```

Destruction

```
DELETE FROM R
WHERE condition
```

Condition est une condition valide pour toute clause WHERE.
« Destruction de tous les clients dont le nom commence par 'M' »

```
DELETE FROM Client
WHERE nom like 'M%'
```

Modification

```
UPDATE R SET A1=v1, A2=v2, …, An=vn
WHERE condition
```

« Augmenter les prix des activités de la station Passac de 10% »

```
UPDATE Activite
SET prix=prix*1.1
WHERE nomStation='Passac'
```

Toutes les mises à jour ne deviennent définitives qu'à l'issue d'une validation par COMMIT. Entre-temps, elles peuvent être annulées par ROLLBACK.

## 3. Création de schémas relationnels

*Exemple :*

Film (**IdFilm**, Titre, Annee, Genre, Resume, IdMES)
Artiste (**IdArt**, NomArt, PrenomArt, NaissArt)
Cinema (**NomCine**, Ville, Rue, Numero)
Horaire (**IdHor**, HeureDebut, HeureFin)
Role (**IdFilm, IdArt**, NomRole)
Salle (**NomCine, No**, Capacite, Climatise)
Seance (**IdFilm**, **NomCine, NoSalle, IdHor**, Tarif)

**Utilisateurs**

L'accès à une base de données est restreint à des utilisateurs connus du SGBD et identifiés par un nom et un mot de passe. Chaque utilisateur se voit attribuer certains droits sur les schémas et les tables.

```
CONNECT utilisateur
```

Le propriétaire (concepteur) d'un schéma a tous les droits sur les éléments de ce schéma. Il définit les droits des autres utilisateurs sur les éléments de ce schéma.

```
GRANT <privilège>
ON <élément du schéma>
TO <utilisateur>
[WITH GRANT OPTION]
```

privilège = INSERT (insertion), UPDATE (modification), SELECT (recherche), DELETE (destruction), REFERENCE (pour permettre de faire référence à une table dans une contrainte d'intégrité), USAGE (pour permettre l'utilisation une définition ≠ table ou insertion).
Pour accorder un privilège, il faut en avoir le droit soit parce qu'on est propriétaire de l'élément du schéma, soit parce que le droit est accordé par la commande WITH GRANT OPTION.

```
GRANT SELECT ON Film TO Marc
```

On peut désigner tous les utilisateurs avec le mot-clé PUBLIC, et tous les privilèges avec l'expression ALL PRIVILEGES.

```
GRANT ALL PRIVILEGES ON Film TO PUBLIC
```

On supprime un droit avec la commande

```
REVOKE <privilège>
ON <élément du schéma>
FROM <utilisateur>
```

**Définition d'un schéma**

Un schéma est l'ensemble des déclarations décrivant une base de données au niveau logique. Outre les tables, un schéma peut comprendre des vues, des contraintes de différents types, des triggers (procédures déclenchées par certains événements), etc.
On crée un schéma en lui donnant un nom puis en donnant la liste des commandes créant les éléments.

```
CREATE DATABASE officiel_des_spectacles
CREATE TABLE Film …
CREATE VIEW …
```

CREATE ASSERTION …
CREATE TRIGGER …

Pour choisir le schéma courant (déjà créé), on utilise la commande

```
USE officiel_des_spectacles
```

**Contraintes et assertions**

Ces contraintes portent le plus souvent sur la restriction des attributs à un ensemble de valeurs, mais on peut trouver des contraintes plus complexes faisant référence à d'autres relations. Une contrainte s'exprime par la commande CHECK (condition).

```
CREATE TABLE Salle
(NomCine      VARCHAR (30) NOT NULL,
No            INTEGER,
Capacite      INTEGER CHECK (Capacite < 300),
Climatise     CHAR(1) CHECK (Climatise IN ('O', 'N')),
PRIMARY KEY (NomCine, No),
FOREIGN KEY NomCine REFERENCES Cinema)
```

Au lieu d'associer une contrainte à un attribut particulier, on peut la définir globalement avec CONSTRAINT en donnant un nom à chaque contrainte.

« Toute salle de plus de 300 places doit être climatisée ».

```
CONSTRAINT clim CHECK (Capacite < 300 OR Climatise = 'O')
```

On peut modifier ou détruire une contrainte : ALTER TABLE … DROP CONSTRAINT ...

```
ALTER TABLE Salle DROP CONSTRAINT clim
```

**Vues**

Création et interrogation

Une requête produit toujours une relation. On peut donc ajouter au schéma des tables « virtuelles » qui ne sont rien d'autres que le résultat de requêtes stockées appelées « vues ». Le résultat de la requête est réévalué à chaque fois qu'on accède à la vue. Une vue est donc dynamique. Une vue n'existe pas physiquement et permet d'obtenir une représentation différente des tables.

```
CREATE VIEW nom_vue
AS <requête>
[WITH CHECK OPTION]
```

« Vue ne contenant que les cinémas parisiens limités à leur nom et leur nombre de salles »

```
CREATE VIEW ParisCinemas
AS    SELECT NomCine, COUNT(*) AS NbSalles
      FROM Cinema c, Salle s
      WHERE Ville = 'Paris'
      AND c.NomCine = s.NomCine
      GROUP BY c.NomCine
```

L'intérêt d'une vue est de donner une représentation « dénormalisée » de la base en regroupant des informations par des jointures.

« Créer une vue Casting donnant explicitement les titres des films, leur année et les noms et prénoms des acteurs »

```
CREATE VIEW Casting (film, annee, acteur, prenom)
AS    SELECT Titre, Annee, NomArt, PrenomArt
      FROM Film f, Role r, Artiste a
      WHERE f.IdFilm = r.IdFilm
      AND a.IdArt = r.IdArt
```

On peut utiliser les vues et les tables dans des requêtes SQL.

« Quels acteurs ont tourné un film en 1997 ? »

```
SELECT acteur, prenom
FROM Casting
WHERE annee = 1997
```

On peut donner des droits en lecture sur cette vue.

```
GRANT SELECT ON Casting TO PUBLIC
```

Modification

Il s'agit de modifier la table d'origine qui sert de support à la vue. Les règles définissant les vues modifiables sont très strictes :

1. la vue doit être basée sur une seule table,
2. toute colonne non référencée dans la vue doit pouvoir être mise à NULL ou disposer d'une valeur par défaut,
3. on ne peut pas mettre à jour un attribut qui résulte d'un calcule ou d'une opération.

L'option WITH CHECK OPTION (à la création de la vue) permet de garantir que toute ligne insérée dans la vue satisfait les critères de sélection de la vue.

Pour supprimer la vue, on utilise DROP

```
DROP VIEW ParisCinemas
```

**Triggers**

C'est une procédure qui est déclenchée par des événements de mise à jour spécifiés par l'utilisateur et ne s'exécute que quand une condition est satisfaite.

Possibilité de manipuler simultanément les valeurs ancienne et nouvelle de la donnée modifiée, servant de test sur l'évolution de la base.

La base de donnée est dynamique : une opération sur la base peut en déclencher d'autres, qui elles-mêmes peuvent entraîner en cascade d'autres réflexes. Mais attention aux boucles sans fin.

```
CREATE TRIGGER nom-trigger
<quand> <événement> ON <table>
[FOR EACH ROW [WHEN <condition>]]
BEGIN
      <action>
END;
```

quand = BEFORE ou AFTER.

événement = DELETE, UPDATE, ou INSERT (séparé par des OR).

FOR EACH ROW = en son absence, le trigger est déclenché une fois pour toute requête modifiant la table ; sinon on peut référencer les anciennes et les nouvelles valeurs du tuple courant avec new.attribut et old.attribut.

action = implantée dans un langage dépendant du SGBD. Elle peut contenir des ordres SQL mais pas de mise à jour de la table courante.

```
CREATE TRIGGER CumulCapaciteGlobal
AFTER UPDATE OR INSERT OR DELETE ON Salle
BEGIN
      UPDATE Cinema c
      SET Capacite = (SELECT SUM (capacite)
                      FROM Salle s
                      WHERE c.NomCine = s.NomCine) ;
END;
```