Chapitre 6

Le langage SQL

Sommaire

Ce chapitre présente le langage SQL d'interrogation et de manipulation de données (insertion, mise-à-jour, destruction). La syntaxe est celle de la norme SQL2, implantée plus ou moins complètement dans la plupart des principaux SGBDR. Certaines fonctionnalités (*triggers* par exemple) sont en cours de normalisation dans SQL3, mais existent déjà dans quelques systèmes en raison de leur importance. Ils seront présentés brièvement.

SQL est un langage *déclaratif* qui permet d'interroger une base de données sans se soucier de la représentation interne (physique) des données, de leur localisation, des chemins d'accès ou des algorithmes nécessaires. A ce titre il s'adresse à une large communauté d'utilisateurs potentiels (pas seulement des informaticiens) et constitue un des atouts les plus spectaculaires (et le plus connu) des SGBDR. On peut l'utiliser de manière interactive, mais également en association avec des interfaces graphiques, des outils de *reporting* ou, très généralement, des langages de programmation.

Ce dernier aspect est très important en pratique car SQL ne permet pas de faire de la programmation au sens courant du terme et doit donc être associé avec un langage comme le C, le COBOL ou JAVA pour réaliser des traitements complexes accédant à une base de données. L'interface de SQL et du langage C sera présentée dans le chapitre 8.

Philippe Rigaux (rigaux@lri.fr), Cours de bases de données, 2003

6.1 Requêtes simples SQL

Dans tout ce chapitre on va prendre l'exemple de la (petite) base de données présentée dans le chapitre sur l'algèbre.

6.1.1 Sélections simples

Commençons par les requêtes les plus simples : la figure 5.1 montre une instance de la base pour les relations `Station` et `Activite`. Première requête : on souhaite extraire de la base le nom de toutes les stations se trouvant aux Antilles.

```
SELECT nomStation
FROM   Station
WHERE  region = 'Antilles'
```

Ce premier exemple montre la structure de base d'une requête SQL, avec les trois clauses `SELECT`, `FROM` et `WHERE`.

- `FROM` indique la (ou les) tables dans lesquelles on trouve les attributs utiles à la requête. Un attribut peut être 'utile' de deux manières (non exclusives) : (1) on souhaite afficher son contenu, (2) on souhaite qu'il ait une valeur particulière (une constante ou la valeur d'un autre attribut).
- `SELECT` indique la liste des attributs constituant le résultat.
- `WHERE` indique les conditions que doivent satisfaire les n-uplets de la base pour faire partie du résultat.

Dans l'exemple précédent, l'interprétation est simple : on parcourt les n-uplets de la relation `Station`. Pour chaque n-uplet, si l'attribut `region` a pour valeur 'Antilles', on place l'attribut `nomStation` dans le résultat [1]. On obtient donc le résultat :

nomStation
Venusa
Santalba

Le résultat d'un ordre SQL est toujours une relation (une table) 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 en algèbre relationnelle. En fait on dispose d'un peu plus de liberté que cela. On peut :

- Renommer les attributs
- Appliquer des fonctions aux valeurs de chaque tuple.
- Introduire des constantes.

Les fonctions applicables aux valeurs des attributs sont par exemple les opérations arithmétiques (+, *, ...) pour les attributs numériques ou des manipulations de chaîne de caractères (concaténation, sous-chaînes, mise en majuscule, ...). Il n'existe pas de norme mais la requête suivante devrait fonctionner sur tous les systèmes : on convertit le prix des activités en euros et on affiche le cours de l'euro avec chaque tuple.

```
SELECT libelle, prix / 6.56, 'Cours de l'euro = ', 6.56
FROM   Activite
WHERE  nomStation = 'Santalba'
```

1. Attention, ce n'est pas forcément ainsi que la requête est *exécutée* par le système.

Ce qui donne le résultat :

libelle	prix / 6.56	'Cours de l'euro ='	'6.56'
Kayac	7.62	'Cours de l'euro ='	6.56

Renommage

Les noms des attributs sont par défaut ceux indiqués dans la clause SELECT, même quand il y a des expressions complexes. Pour renommer les attributs, on utilise le mot-clé AS.

```
SELECT libelle, prix / 6.56 AS prixEnEuros,
          'Cours de l'euro = ', 6.56 AS cours
FROM   Activite
WHERE  nomStation = 'Santalba'
```

On obtient alors :

libelle	prixEnEuros	'Cours de l'euro ='	cours
Kayac	7.69	'Cours de l'euro ='	6.56

Remarque : Sur certains systèmes, le mot-clé AS est optionnel.

Doublons

L'introduction de fonctions permet d'aller au-delà de ce qui est possible en algèbre relationnelle. Il existe une autre différence, plus subtile : SQL permet l'existence de doublons dans les tables (il ne s'agit donc pas d'ensemble au sens strict du terme). La spécification de clés permet d'éviter les doublons dans les relations stockées, mais il peuvent apparaître dans le résultat d'une requête. Exemple :

```
SELECT libelle
FROM   Activite
```

donnera autant de lignes dans le résultat que dans la table Activite.

libelle
Voile
Plongee
Plongee
Ski
Piscine
Kayac

Pour éviter d'obtenir deux tuples identiques, on peut utiliser le mot-clé DISTINCT.

```
SELECT DISTINCT libelle
FROM   Activite
```

Attention : l'élimination des doublons peut être une opération coûteuse.

Tri du résultat

Il est possible de trier le résultat d'un requête avec la clause ORDER BY suivie de la liste des attributs servant de critère au tri. Exemple :

```
SELECT   *
FROM     Station
ORDER BY tarif, nomStation
```


trie, en ordre ascendant, les stations par leur tarif, puis, pour un même tarif, présente les stations selon l'ordre lexicographique. Pour trier en ordre descendant, on ajoute le mot-clé DESC après la liste des attributs.

Voici maintenant la plus simple des requêtes SQL : elle consiste à afficher l'intégralité d'une table. Pour avoir toutes les lignes on omet la clause WHERE, et pour avoir toutes les colonnes, on peut au choix lister tous les attributs ou utiliser le caractère '*' qui a la même signification.

```
SELECT *
FROM   Station
```

6.1.2 La clause WHERE

Dans la clause WHERE, on spécifie une condition booléenne portant sur les attributs des relations du FROM. On utilise pour cela de manière standard le AND, le OR, le NOT et les parenthèses pour changer l'ordre de priorité des opérateurs booléens. Par exemple :

```
SELECT nomStation, libelle
FROM   Activite
WHERE  nomStation  = 'Santalba'
AND    (prix > 50 AND prix < 120)
```

Les opérateurs de comparaison sont ceux du Pascal : $<$, $<=$, $>$, $>$, $=$, et $<>$ pour exprimer la différence ($!=$ est également possible). Pour obtenir une recherche par intervalle, on peut également utiliser le mot-clé BETWEEN. La requête précédente est équivalente à :

```
SELECT nomStation, libelle
FROM   Activite
WHERE  nomStation = 'Santalba'
AND    prix BETWEEN 50 AND 120
```

Chaînes de caractères

Les comparaisons de chaînes de caractères soulèvent quelques problèmes délicats.

1. Il faut être attentif aux différences entre chaînes de longueur fixe et chaînes de longueur variable. Les premières sont complétées par des blancs (' ') et pas les secondes.

2. Si SQL ne distingue pas majuscules et minuscules pour les mot-clés, il n'en va pas de même pour les valeurs. Donc 'SANTALBA' est différent de 'Santalba'.

SQL fournit des options pour les recherches par motif (*pattern matching*) à l'aide de la clause LIKE. Le caractère '_' désigne n'importe quel caractère, et le '%' n'importe quelle chaîne de caractères. Par exemple, voici la requête cherchant toutes les stations dont le nom termine par un 'a'.

```
SELECT nomStation
FROM   Station
WHERE  nomStation LIKE '%a'
```

Quelles sont les stations dont le nom commence par un 'V' et comprend exactement 6 caractères ?

```
SELECT nomStation
FROM   Station
WHERE  nomStation LIKE 'V_____'
```

Dates

Une autre différence avec l'algèbre est la possibilité de manipuler des dates. En fait tous les systèmes proposaient bien avant la normalisation leur propre format de date, et la norme préconisée par SQL2 n'est de ce fait pas suivie par tous.

Une date est spécifiée en SQL2 par le mot-clé `DATE` suivi d'une chaîne de caractères au format 'aaaa-mm-jj', par exemple `DATE '1998-01-01'`. Les zéros sont nécessaires afin que le mois et le quantième comprennent systématiquement deux chiffres.

On peut effectuer des sélections sur les dates à l'aide des comparateurs usuels. Voici par exemple la requête 'ID des clients qui ont commencé un séjour en juillet 1998'.

```
SELECT idClient
FROM   Sejour
WHERE  debut BETWEEN DATE '1998-07-01' AND  DATE '1998-07-31'
```

Les systèmes proposent de plus des fonctions permettant de calculer des écarts de dates, d'ajouter des mois ou des années à des dates, etc.

6.1.3 Valeurs nulles

Autre spécificité de SQL par rapport à l'algèbre : on admet que la valeur de certains attributs soit inconnue, et on parle alors de *valeur nulle*, désignée par le mot-clé `NULL`. Il est très important de comprendre que la 'valeur nulle' n'est en fait pas une valeur mais une absence de valeur, et que l'on ne peut donc lui appliquer aucune des opérations ou comparaisons usuelles.

- Toute opération appliquée à `NULL` donne pour résultat `NULL`.
- Toute comparaison avec `NULL` donne un résultat qui n'est ni vrai, ni faux mais une troisième valeur booléenne, `UNKNOWN`.

Les valeurs booléennes `TRUE`, `FALSE` et `UNKNOWN` sont définies de la manière suivante : `TRUE` vaut 1, `FALSE` 0 et `UNKNOWN` 1/2. Les connecteurs logiques donnent alors les résultats suivants :

1. x `AND` $y = min(x, y)$
2. x `OR` $y = max(x, y)$
3. `NOT` $x = 1 - x$

Les conditions exprimées dans une clause `WHERE` sont évaluées pour chaque tuple, et ne sont conservés dans le résultat que les tuples pour lesquels cette évaluation donne `TRUE`. La présence d'une valeur nulle dans une comparaison a donc souvent (mais pas toujours !) le même effet que si cette comparaison échoue et renvoie `FALSE`.

Voici une instance de la table `SEJOUR` avec des informations manquantes.

SEJOUR			
idClient	station	début	nbPlaces
10	Passac	1998-07-01	2
20	Santalba	1998-08-03	
30	Passac		3

La présence de `NULL` peut avoir des effets surprenants. Par exemple la requête suivante

```
SELECT station
FROM   Sejour
WHERE  nbPlaces <= 10 OR nbPlaces >= 10
```


devrait en principe ramener toutes les stations de la table. En fait 'Santalba' ne figurera pas dans le résultat car `nbPlaces` est à `NULL`.

Autre piège : `NULL` est un mot-clé, pas une constante. Donc une comparaison comme `nbPlaces = NULL` est incorrecte. Le prédicat pour tester l'absence de valeur dans une colonne est x `IS NULL` (et son inverse `IS NOT NULL`). La requête suivante sélectionne tous les séjours pour lesquels on connaît le nombre de places.

```
SELECT *
FROM   Sejour
WHERE  nbPlaces IS NOT NULL
```

La présence de `NULL` est une source de problèmes : dans la mesure du possible il faut l'éviter en spécifiant la contrainte `NOT NULL` ou en donnant une valeur par défaut.

6.2 Requêtes sur plusieurs tables

Les requêtes SQL décrites dans cette section permettent de manipuler simultanément plusieurs tables et d'exprimer les opérations binaires de l'algèbre relationnelle : jointure, produit cartésien, union, intersection, différence.

6.2.1 Jointures

La jointure est une des opérations les plus utiles (et donc une des plus courantes) puisqu'elle permet d'exprimer des requêtes portant sur des données réparties dans plusieurs tables. La syntaxe pour exprimer des jointures avec SQL est une extension directe de celle étudiée précédemment dans le cas des sélections simples : 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`.

Prenons l'exemple de la requête suivante : *donner le nom des clients avec le nom des stations où ils ont séjourné*. Le nom du client est dans la table `Client`, l'information sur le lien client/station dans la table `Sejour`. Deux tuples de ces tables peuvent être joints s'ils concernent le même client, ce qui peut s'exprimer à l'aide de l'identifiant du client. On obtient la requête :

```
SELECT nom, station
FROM   Client, Sejour
WHERE  id = idClient
```

On peut remarquer qu'il n'y a pas dans ce cas d'ambiguité sur les noms des attributs : `nom` et `id` viennent de la table `Client`, tandis que `station` et `idClient` viennent de la table `Sejour`. Il peut arriver (il arrive de fait fréquemment) qu'un même nom d'attribut soit partagé par plusieurs tables impliquées dans une jointure. Dans ce cas on résout l'ambiguité en préfixant l'attribut par le nom de la table.

Exemple : afficher le nom d'une station, son tarif hebdomadaire, ses activités et leurs prix.

```
SELECT nomStation, tarif, libelle, prix
FROM   Station, Activite
WHERE  Station.nomStation = Activite.nomStation
```

Comme il peut être fastidieux de répéter intégralement le nom d'une table, on peut lui associer un synonyme et utiliser ce synonyme en tant que préfixe. La requête précédente devient par exemple : [2]

```
SELECT S.nomStation, tarif, libelle, prix
FROM   Station S, Activite A
WHERE  S.nomStation = A.nomStation
```

2. Au lieu de `Station S`, la norme SQL2 préconise `Station AS S`, mais le `AS` est parfois inconnu ou optionnel.

Bien entendu, on peut effectuer des jointures sur un nombre quelconque de tables, et les combiner avec des sélections. Voici par exemple la requête qui affiche le nom des clients habitant Paris, les stations où ils ont séjourné avec la date, enfin le tarif hebdomadaire pour chaque station.

```
SELECT nom, station, debut, tarif
FROM   Client, Sejour, Station
WHERE  ville = 'Paris'
AND    id = idClient
AND    station = nomStation
```

Il n'y a pas d'ambiguité sur les noms d'attributs donc il est inutile en l'occurence d'employer des synonymes. Il existe en revanche une situation où l'utilisation des synonymes est indispensable : celle ou l'on souhaite effectuer une jointure d'une relation avec elle-même.

Considérons la requête suivante : *Donner les couples de stations situées dans la même région*. Ici toutes les informations nécessaires sont dans la seule table `Station`, mais on construit *un tuple* dans le résultat avec *deux tuples* partageant la même valeur pour l'attribut `région`.

Tout se passe comme s'il on devait faire la jointure entre deux versions distinctes de la table `Station`. Techniquement, on résout le problème en SQL en utilisant deux synonymes distincts.

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

On peut imaginer que `s1` et `s2` sont deux 'curseurs' qui parcourent indépendamment la table `Station` et permettent de constituer des couples de tuples auxquels on applique la condition de jointure.

Interprétation d'une jointure

L'interprétation d'une jointure est une généralisation de l'interprétation d'un ordre SQL portant sur une seule table. Intuitivement, on parcourt tous les tuples définis par la clause `FROM`, et on leur applique la condition exprimée dans le `WHERE`. Finalement on ne garde que les attributs spécifiés dans la clause `SELECT`.

Quels sont les tuples définis par le `FROM` ? Dans le cas d'une seule table, il n'y a pas de difficulté. Quand il y a plusieurs tables, on peut donner (au moins) deux définitions équivalentes :

1. **Boucles imbriquées**. On considère chaque synonyme de table (ou par défaut chaque nom de table) comme une *variable tuple*. Maintenant on construit des boucles imbriquées, chaque boucle correspondant à une des tables du `FROM` et permettant à la variable correspondante d'itérer sur le contenu de la table.

   A l'intérieur de l'ensemble des boucles, on applique la clause `WHERE`.

2. **Produit cartésien**. 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 ambiguités.

   On est alors ramené à la situation où il y a une seule table (le résultat du produit cartésien) et on interprête l'ordre SQL comme dans le cas des requêtes simples.

La première interprétation est proche de ce que l'on obtiendrait si on devait programmer une requête avec un langage comme le C ou Pascal, la deuxième s'inspire de l'algèbre relationnelle.

6.2.2 Union, intersection et différence

L'expression de ces trois opérations ensemblistes en SQL est très proche de l'algèbre relationnelle. On construit deux requêtes dont les résultats ont même arité (même nombre de colonnes et mêmes types d'attributs), et on les relie par un des mot-clé `UNION`, `INTERSECT` ou `EXCEPT`.

Trois exemples suffiront pour illustrer ces opérations.

1. Donnez tous les noms de région dans la base.

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

2. Donnez les régions où l'on trouve à la fois des clients et des stations.

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

3. Quelles sont les régions où l'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. Le coût de l'élimination de doublons n'étant pas négligeable, il se peut cependant que certains systèmes fassent un choix différent.

L'union ne peut être exprimée autrement qu'avec `UNION`. En revanche `INTERSECT` peut être exprimée avec une jointure, et la différence s'obtient, souvent de manière plus aisée, à l'aide des requêtes imbriquées.

6.3 Requêtes imbriquées

Jusqu'à présent les conditions exprimées dans le `WHERE` consistaient en comparaisons de valeurs scalaires. Il est possible également avec SQL d'exprimer des conditions sur des *relations*. Pour l'essentiel, ces conditions consistent en l'existence d'au moins un tuple dans la relation testée, ou en l'appartenance d'un tuple particulier à la relation.

La relation testée est construite par une requête `SELECT ... FROM ... WHERE` que l'on appelle sous-requête ou *requête imbriquée* puisqu'elle apparaît dans la clause `WHERE`

6.3.1 Conditions portant sur des relations

Une première utilisation des sous-requêtes est d'offrir une alternative syntaxique à l'expression de jointures. Les jointures concernées sont celles pour lesquelles le résultat est constitué avec des attributs provenant d'une seule des deux tables, l'autre ne servant que pour exprimer des conditions.

Prenons l'exemple suivant : on veut les noms des stations où ont séjourné des clients parisiens. On peut obtenir le résultat avec une jointure classique.

```
SELECT station
FROM   Sejour, Client
WHERE  id = idClient
AND    ville = 'Paris'
```

Ici, le résultat, `station`, provient de la table `SEJOUR`. D'où l'idée de séparer la requête en deux parties : (1) on constitue avec une sous-requête les `id`s des clients parisiens, puis (2) on utilise ce résultat dans la requête principale.

```
SELECT  station
```

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

Le mot-clé `IN` exprime clairement la condition d'appartenance de `idClient` à la relation formée par la requête imbriquée. On peut remplacer le `IN` par un simple '=' *si on est sûr que la sous-requête ramène un et un seul tuple*. Par exemple :

```
SELECT  nom, prenom
FROM    Client
WHERE   region = (SELECT region FROM Station
                  WHERE  nomStation = 'Santalba')
```

est (partiellement) correct car la recherche dans la sous-requête s'effectue par la clé. En revanche il se peut qu'aucun tuple ne soit ramené, ce qui génère une erreur.

Voici les conditions que l'on peut exprimer sur une relation R construite avec une requête imbriquée.

1. `EXISTS R`. Renvoie `TRUE` si R n'est pas vide, `FALSE` sinon.
2. t `IN R` où est un tuple dont le type est celui de R. `TRUE` si t appartient à R, `FALSE` sinon.
3. $v\ cmp$ `ANY R`, où cmp est un comparateur SQL ($<$, $<$, $=$, etc.). Renvoie `TRUE` si la comparaison avec *au moins un* des tuples de la relation unaire R renvoie `TRUE`.
4. $v\ cmp$ `ALL R`, où cmp est un comparateur SQL ($<$, $<$, $=$, etc.). Renvoie `TRUE` si la comparaison avec *tous* les tuples de la relation unaire R renvoie `TRUE`.

De plus, toutes ces expressions peuvent être préfixées par `NOT` pour obtenir la négation. Voici quelques exemples.

– *Où (station, lieu) ne peut-on pas faire du ski ?*

```
SELECT nomStation, lieu
FROM   Station
WHERE  nomStation NOT IN (SELECT nomStation FROM Activite
                          WHERE  libelle = 'Ski')
```

– *Quelle station pratique le tarif le plus élevé ?*

```
SELECT nomStation
FROM   Station
WHERE  tarif >= ALL (SELECT tarif FROM Station)
```

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

```
SELECT nomStation, libelle
FROM   Activite
WHERE  prix IN (SELECT prix FROM Activite
                WHERE  nomStation = 'Santalba')
```

Ces requêtes peuvent s'exprimer sans imbrication (exercice), parfois de manière moins élégante ou moins concise. La différence, en particulier, s'exprime facilement avec `NOT IN` ou `NOT EXISTS`.

6.3.2 Sous-requêtes correllées

Les exemples de requêtes imbriquées donnés précédemment pouvait être évalués indépendamment de la requête principale, ce qui permet au système (s'il le juge nécessaire) d'exécuter la requête en deux phases. Il peut arriver que la sous-requête soit basée sur une ou plusieurs valeurs issues des relations de la requête principale. On parle alors de *requêtes correllées*.

Exemple : *quels sont les clients (nom, prénom) qui ont séjourné à Santalba.*

```
SELECT nom, prenom
FROM   Client
WHERE  EXISTS (SELECT 'x' FROM Sejour
               WHERE  station = 'Santalba'
               AND    idClient = id)
```

Le `id` dans la requête imbriquée n'appartient pas à la table `Sejour` mais à la table `Client` référencée dans le `FROM` de la requête principale.

Remarque : on peut employer un `NOT IN` à la place du `NOT EXISTS` (exercice), de même que l'on peut toujours employer `EXISTS` à la place de `IN`. Voici une des requêtes précédentes où l'on a appliqué cette transformation, en utilisant de plus des synonymes.
Dans quelle station pratique-t-on une activité au même prix qu'à Santalba ?

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

Cette requête est elle-même équivalente à une jointure sans requête imbriquée.

6.4 Agrégration

Toutes les requêtes vues jusqu'à présent pouvaient être interprétées comme une suite d'opérations effectuées *tuple à tuple*. De même le résultat était toujours constitué de valeurs issues de tuples individuels. Les fonctionnalités *d'agrégation* de SQL permettent d'exprimer des conditions *sur des groupes de tuples*, et de constituer le résultat par *agrégation de valeurs* au sein de chaque groupe.

La syntaxe SQL fournit donc :

1. Le moyen de partitioner une relation en *groupes* selon certains critères.
2. Le moyen d'exprimer des conditions sur ces groupes.
3. Des fonctions d'agrégation.

Il existe un groupe par défaut : c'est la relation toute entière. Sans même définir de groupe, on peut utiliser les fonctions d'agrégation.

6.4.1 Fonctions d'agrégation

Ces fonctions s'appliquent à une colonne, en général de type numérique. Ce sont :

1. `COUNT` qui compte le nombre de valeurs *non nulles*.
2. `MAX` et `MIN`.
3. `AVG` qui calcule la moyenne des valeurs de la colonne.

4. SUM qui effectue le cumul.

Exemple d'utilisation de ces fonctions :

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

Remarque importante : 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, voir plus loin). La requête suivante est incorrecte (pourquoi ?).

```
SELECT nomStation, AVG(tarif)
FROM   Station
```

A condition de respecter cette règle, on peut utiliser les ordres SQL les plus complexes. Exemple : *Combien de places a réservé Mr Kerouac pour l'ensemble des séjours ?*.

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

6.4.2 La clause GROUP BY

Dans les requêtes précédentes, on appliquait la fonction d'agrégation à l'ensemble du résultat d'une requête (donc éventuellement à l'ensemble de la table elle-même). Une fonctionnalité complémentaire consiste à *partitioner* ce résultat en groupes, et à appliquer la ou les fonction(s) à chaque groupe.

On construit les groupes en associant les tuples partageant la même valeur pour une ou plusieurs colonnes.

Exemple : afficher les régions avec le nombre de stations.

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

Donc ici on constitue un groupe pour chaque région. Puis on affiche ce groupe sous la forme d'un tuple, dans lequel les attributs peuvent être de deux types :

1. les attributs *dont la valeur est constante pour l'ensemble du groupe*, soit précisément les attributs du GROUP BY. Exemple ici l'attribut region ;
2. le résultat d'une fonction d'agrégation appliquée au groupe : ici COUNT(nomStation).

Bien entendu il est possible d'exprimer des ordres SQL complexes et d'appliquer un GROUP BY au résultat. De plus, il n'est pas nécessaire de faire figurer tous les attributs du GROUP BY dans la clause SELECT.

Exemple : on souhaite consulter le nombre de places reservées, *par client*.

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

L'interprétation est simple : (1) on exécute d'abord la requête SELECT ... FROM ... WHERE, puis (2) on prend le résultat et on le partitione, enfin (3) on calcule le résultat des fonctions.

A l'issue de l'étape (2), on peut imaginer une relation qui n'est pas en première forme normale : on y trouverait des tuples avec les attributs du GROUP BY sous forme de valeur atomique, puis des attributs de type *ensemble* (donc interdits dans le modèle relationnel). C'est pour se ramener en 1FN que l'on doit appliquer des fonctions d'agrégation à ces ensembles.

Exercice : pourquoi grouper sur id, nom ? Quels sont les autres choix possibles et leurs inconvénients ?

6.4.3 La clause HAVING

Finalement, 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.

Exemple : on souhaite consulter le nombre de places reservées, par client, *pour les clients ayant réservé plus de 10 places*.

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

On voit que la condition porte ici sur une propriété de l'ensemble des tuples du groupe, et pas de chaque tuple pris individuellement. La clause HAVING est donc toujours exprimée sur le résultat de fonctions d'agrégation.

6.5 Mises-à-jour

Les commandes de mise-à-jour (insertion, destruction, modification) sont considérablement plus simples que les requêtes.

6.5.1 Insertion

L'insertion s'effectue avec la commande INSERT dont la syntaxe est la suivante :

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

R est le nom d'une relation, et les A1, ... An sont les noms des attributs dans lequels on souhaite placer une valeur. *Les autres attributs seront donc à* NULL *(ou à la valeur par défaut)*. Tous les attributs spécifiés NOT NULL (et sans valeur par défaut) doivent donc figurer dans une clause INSERT.

Les v1, ... vn sont les valeurs des attributs. Exemple de l'insertion d'un tuple dans la table Client.

```
INSERT INTO Client (id, nom, prenom)
VALUES (40, 'Moriarty', 'Dean')
```

Donc, à l'issue de cette insertion, les attributs ville et region seront à NULL.

Il est également possible d'insérer dans une table le résultat d'une requête. Dans ce cas la partie VALUES ... est remplacée par la requête elle-même. Exemple : on a créé une table Sites (lieu, region) 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
```

Bien entendu le nombre d'attributs et le type de ces derniers doivent être cohérents.

6.5.2 Destruction

La destruction s'effectue avec la clause DELETE dont la syntaxe est :

```
DELETE FROM R
WHERE condition
```

`R` est bien entendu la table, et `condition` est toute condition valide pour une clause `WHERE`. En d'autres termes, si on effectue, avant la destruction, la requête

```
SELECT * FROM R
WHERE condition
```

on obtient l'ensemble des lignes qui seront détruites par `DELETE`. Procéder de cette manière est un des moyens de s'assurer que l'on va bien détruire ce que l'on souhaite....

Exemple : destruction de tous les clients dont le nom commence par 'M'.

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

6.5.3 Modification

La modification s'effectue avec la clause `UPDATE`. La syntaxe est proche de celle du `DELETE` :

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

`R` est la relation, les `Ai` sont les attributs, les `vi` les nouvelles valeurs et `condition` est toute condition valide pour la clause `WHERE`. Exemple : augmenter le prix des activités de la station Passac de 10%.

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

Une remarque importante : toutes les mises-à-jour ne deviennent définitives qu'à l'issue d'une validation par `commit`. Entretemps elles peuvent être annulées par `rollback`. Voir le cours sur la concurrence d'accès.

6.6 Exercices

Exercice 11 *Reprendre les expressions algébriques du premier exercice du chapitre algèbre, et les exprimer en SQL.*

Exercice 12 *Donnez l'expression SQL des requêtes suivantes, ainsi que le résultat obtenu avec la base du chapitre "Le langage SQL".*

1. *Nom des stations ayant strictement plus de 200 places.*
2. *Noms des clients dont le nom commence par 'P' ou dont le solde est supérieur à 10 000.*
3. *Quelles sont les régions dont l'intitulé comprend (au moins) deux mots ?*
4. *Nom des stations qui proposent de la plongée.*
5. *Nom des clients qui sont allés à Santalba.*
6. *Donnez les couples de clients qui habitent dans la même région. Attention : un couple doit apparaître une seule fois.*
7. *Nom des régions qu'a visité Mr Pascal.*
8. *Nom des stations visitées par des européens.*
9. *Qui n'est pas allé dans la station Farniente ?*
10. *Quelles stations ne proposent pas de la plongée ?*

11. *Combien de séjours ont eu lieu à Passac ?*
12. *Donner, pour chaque station, le nombre de séjours qui s'y sont déroulés.*
13. *Donner les stations où se sont déroulés au moins 3 séjours.*
14. (♣) *Les clients qui sont allés dans* toutes *les stations.*

Exercice 13 (Valeurs nulles) *On considère la table suivante :*

STATION				
NomStation	*Capacité*	*Lieu*	*Région*	*Tarif*
Gratuite	*80*	*Guadeloupe*	*Antilles*	
NullePart	*150*			*2000*

1. *Donnez les résultats des requêtes suivantes (rappel : le `'||'` est la concaténation de chaînes de caractères.).*

   (a)
   ```
   SELECT nomStation FROM Station
   WHERE tarif >200
   ```
   (b)
   ```
   SELECT tarif * 3 FROM Station
   WHERE nomStation LIKE '%l%' AND lieu LIKE '%'
   ```
   (c)
   ```
   SELECT ' Lieu = ' || lieu FROM Station
   WHERE capacite >= 100 OR tarif >= 1000
   ```
   (d)
   ```
   SELECT ' Lieu = ' || lieu FROM Station
   WHERE NOT (capacite <100 AND tarif <1000)
   ```

2. *Les deux dernières requêtes sont-elles équivalentes (i.e. donnent-elles le même résultat quel que soit le contenu de la table) ?*
3. *Supposons que l'on ait conservé une logique bivaluée (avec `TRUE` et `FALSE`) et adopté la règle suivante : toute comparaison avec un `NULL` donne `FALSE`. Obtient-on des résultats équivalents ? Cette règle est-elle correcte ?*
4. *Même question, en supposant que toute comparaison avec `NULL` donne `TRUE`.*

Exercice 14 *On reprend la requête constituant la liste des stations avec leurs activités, légèrement modifiée.*

```
SELECT S.nomStation, tarif, libelle, prix
FROM   Station S, Activites A, Sejour
WHERE  S.nomStation = A.nomStation
```

1. *La table `Sejour` est-elle nécessaire dans le `FROM` ?*
2. *Qu'obtient-on dans les trois cas suivants : (1) la table `Sejour` contient 1 tuple, (2) la table `Sejour` contient 100 000 tuples, (3) la table `Sejour` est vide.*
3. *Soit trois tables `R`, `S` et `T` ayant chacune un seul attribut `A`. On veut calculer `R ∩ (S ∪ T)`.*

   (a) La requête suivante est-elle correcte ? Expliquez pourquoi.
   ```
   SELECT R.A FROM R, S, T WHERE R.A=S.A OR R.A=T.A
   ```
   (b) Donnez la bonne requête.