Base de données et SQL

Thibault PENNING

Les bases de données répondent à plusieurs besoins qui incluent :

- Stocker un grand nombre d'informations
- Les lier entre eux
- Les modifier
- Faire des recherches avancées

Définition 0.1 (Base de donnée)**.** Ensemble de données modélisant les objets d'une partie du monde réel et servant de support à une application informatique.

On peut citer en plus que :

- Les données sont non indépendantes
- La base de données est interrogeable sur le contenu (avec des critères)

Le modèle relationnel

Quelques définitions

Le modèle relationnel a été proposé par Edgar Franck CODD en 1970.

Dans le modèle relationnel, les données sont représentées sous forme de relations entre des données relevant du même concept.

Définition 0.1 (Entité)**.** Un objet que l'on veut modéliser dans une base de données, est une entité.

Exemple 0.1. Avec un livre : un *tuple* (ISBN, titres, auteurs, valeurs)

Définition 0.2 (Domaine)**.** Ensemble de valeurs caractérisées par un nom.

Les données des bases de données prennent donc leur valeur dans les ***domaines***.

On peut citer des domaines comme ENTIER, BOOLEEN ou CHAINE (de caractères), mais aussi des domaines plus spécifiques comme DATE (3 chiffres, jours, mois et années) ou COULEURS = {Blanc, Noir, Rose et Bleu}.

Définition 0.3 (Relations)**.** Une relation étant l'ensemble fini d'entité relevant du même concept. Les entités sont décrites par certaines de leurs caractéristiques. Une relation est identifiée par un nom.

Une relation peut être vue comme une collection de tubes dont chaque index est pris dans le même domaine (entre chaque tuple).

Il est donc possible de les représenter sous forme 2D appelée **Table** avec comme colonne correspondant au domaine et chaque ligne une entité de la relation.

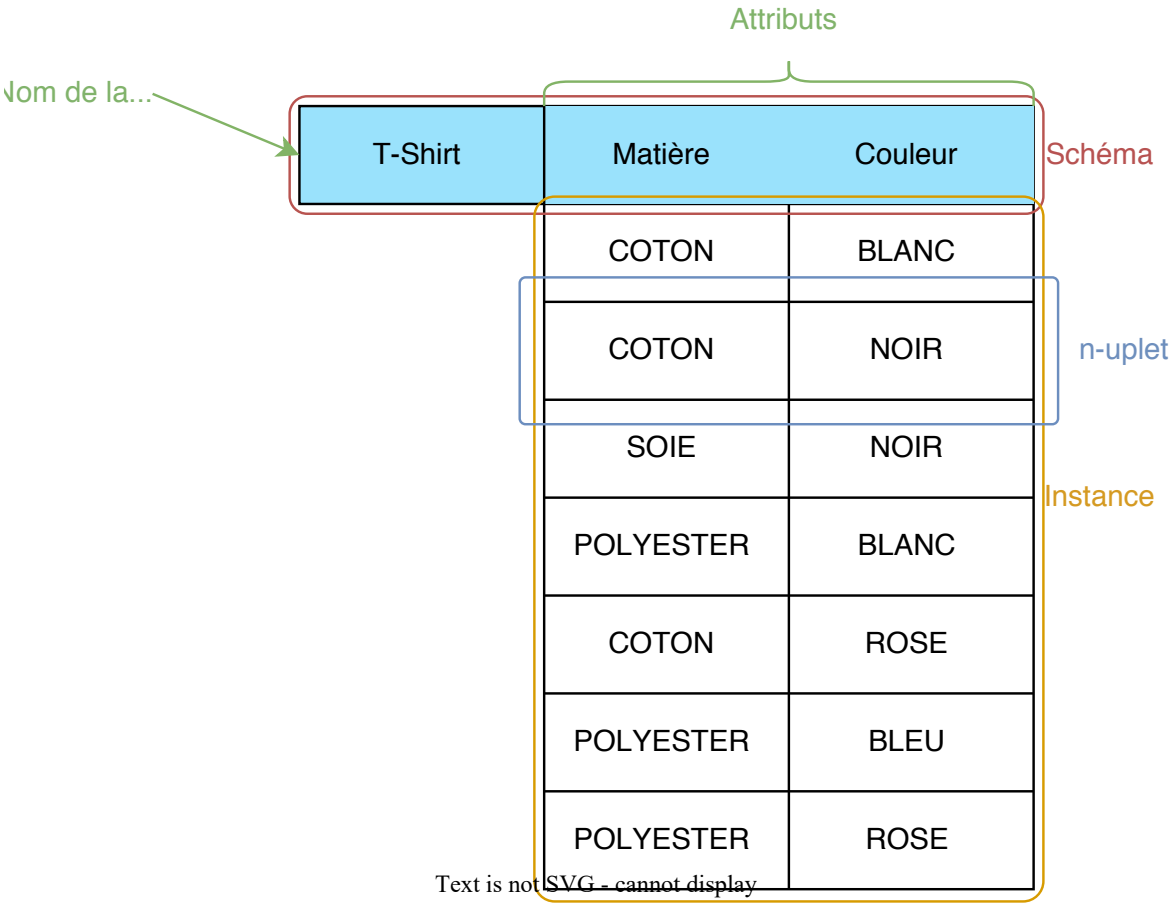

Figure 1: Example d'une table d'une relation

Définition 0.4 (Attribut)**.** Caractéristiques décrivant une entité (une des colonnes) d'une relation. Elle est identifiée par un nom

Exemple 0.2. Attribut *Matière* de la relation T-shirt.

> **Note**
>
> Le nom est souvent celui du domaine.

Définition 0.5 (Schéma (d'une relation))**.** Ensemble des attributs d'une relation (ainsi que leur domaine). On associe souvent au schéma le nom d'une relation.

Voir Figure 1

> **Note**
>
> Le schéma est en invariant d'une relation, peu importe les tubes ajoutés ou supprimés , ...

Exemple 0.3. `T-Shirt(Couleur:Chaîne, Matière:Chaîne, Tour_taille:Flottant, Année_crétion:Date)`

Définition 0.6 (Instance)**.** Ensemble des n-uplets d'une relation à un instant donné.

Voir Figure 1

> **Note**
>
> - L'ordre des nuplay n'a pas d'importance
> - Pas de doublons.

Contrainte d'intégrité

Afin de garantir la cohérence d'une base de données, on se munit de contraintes que la base de données et ces données doivent respecter.

Si une donnée briserait une contrainte de la base de données, alors cette dernière ne doit pas être ajoutée ou modifiée.

Contrainte d'unicité

Définition 0.7 (Surclé (ou super clé))**.** Ensemble d'attributs permettant d'identifier de manière unique une relation, peu importe l'instance de la base de données.

i.e $\forall r, l \in \text{Attributs}, r \neq s$ ssi $\text{Instance}[r] \neq \text{Instance}[s]$

Définition 0.8 (Clé)**.** Surclé de taille minimale, i.e., ensemble minimal d'attributs permettant d'identifier de manière unique une relation.

> **!** Important
>
> Toute relation dont comportait au moins une clé.

Définition 0.9. Clé principal choisie pour le rôle d'identification.

> **i** Note
>
> Il doit toujours exister une clé primaire dans une base de données relationnelle.

> **i** Note
>
> On indique une clé primaire en soulignant les attributs qui la composent.
>
> **Exemple 0.4.** Pour une relation *Eleve* ayant comme clé primaire l'ensemble {**nom**, **prenom**, **naissance**}
> Eleve(**nom**:*Chaine*, **prenom**:*Chaine*, **classe**:*Chaine*, **naissance**:*Date*, **moyenne_NSI**:*Flottant*)

Définition 0.10. On appelle cette contrainte (existence d'une clé primaire), la **contrainte d'unicité**

Contraindre référentielle

Le modèle relationnel a pour but de tirer des relations non seulement dans les tables, mais aussi entre les tables.

Exemple 0.5. Par exemple, si l'on veut gérer les commandes d'un magasin de T-shirts.

Client(nom:*Chaîne*, **adresse**:*Chaîne*, **bon_client**:*Bool*)

Commande(num_commande:*Entier*, **client**:?, **t-shirt**:?)

T-Shirt(**couleur**:*Couleur*, **tissu**:**Tissu**, **date_sortie**:*Date*)

On aimerait lier les commandes aux clients et t shirts pour pouvoir envoyer les bonnes commandes sont redondances. Une des idées serait de lier les attributs ***clients*** et ***shirts*** de commande à des entitées existante dans les relations clients et t-shirts.

Pour cela, nous pouvons référencer dans la relation commande, une entité existante grâce à leurs clés (primaire) (de leur relation d'origine, **nom** par exemple pour les clients).

Définition 0.11 (Clé étrangère)**.** Ensemble d'attributs qui apparaît comme clé dans une autre relation (en général primaire).

> Mise en garde
>
> Une clé étrangère n'est pas une clé dans la relation actuelle. Elle n'est une clé que dans la relation qu'elle référence.

La **contrainte référentielle** stipule qu'une clé étrangère doit apparaître comme clé primaire dans une autre relation. Elle contraint les liens entre relations.

Contraint de domaine

De manière générale, les domaines sont souvent représentés par des types de base (chaine entiers, booléens, flottants, date, temps). Mais ces types peuvent être trop permissifs. Par exemple, il n'existe pas de T-shirt avec un tour de taille de moins 2 cm, ou de 1000 cm. Alors que ces valeurs sont tout à fait légales si nous décidons de représenter un tour de taille par un entier ou un flottant.

Ainsi, les types de base ne sont pas toujours suffisants, ils permettent d'indiquer la représentation mémoire que prendra la valeur, mais pas la cohérence des données avec le monde réelle.

Pour palliasa par exemple, pour le tour de taille, il serait pratique de définir une valeur cohérente (entre 30 et 200 par exemple). Toute autre valeur serait aberrante et ne serait pas considérée. Ainsi, nous garantirons que les données présentant la base de données représente bien le monde réel comme la base de données est supposé le faire.

Définition 0.12. C'est ici qu'apparaît la **contrainte de domaine** qui impose que les valeurs d'un attribut vérifient une assertion logique.

Exemple 0.6. On peut citer comme assertion logique :

- Pour restreindre le domaine du tour de taille $30 \leq \text{tour\_taille} \leq 200$
- Pour décrire le domaine des couleurs $\text{Couleur} \in \{Rose, Bleu, Noir, Blanc\}$

Un language de requète : le SQL

Définition 0.1 (SGBD)**.** Un SGBD ou *système de gestion de base de données* est un ensemble d'outils logiciels permettant de gérer des bases de données stockées sur un support informatique.

En pratique, un SGBD est un logiciel (comme MySQL, MariaDB ou PostgreSQL) ou librairie comme (SQLite) par laquelle le développeur passe afin d'interagir avec une base de données. L'avantage étant que l'abstraction matérielle, mais aussi les différentes optimisations, etc, ne sont pas du ressort du développeur, mais celui du créateur de la base de données permettant ainsi à ce dernier de se concentrer exclusivement sur les manipulations voulues sur les données.

Maintenant que nous avons vu les BDD relationnel, nous allons nous doter d'un langage capable d'interagir avec un SGBD relationnelle. Le langage le plus commun pour cela est le SQL. C'est ce que l'on appelle un langage de définition des données.

Le SQL est composé de bout de code. que l'on appelle requête. En général, une requête permet de rechercher des données dans une base de données (interrogation sur son contenu.). Elle peut aussi créer de nouvelles relations, supprimer, modifier des données, …

Le développeur enverra ainsi des requêtes SQL qui sera interprétées par la SGBD, qui exécutera elle-même l'action. Il est intéressant de noter que, par exemple, les contraintes précédemment cité seront gérées par le SGBD lors de l'exécution de des requêtes. Nous verrons cela dans la Section .

Une requête commence par une clause(`SELECT`, `INSERT`, …) et se termine par un point-virgule (`;`). Les commentaire commence par deux tirets `--`.

Le SQL n'est pas sensible à la casse en ce qui concerne les clauses. En revanche, il est courant de noter ces dernières tout en majuscules afin de différencier la partie purement du langage et celle des définitions spécifique des requête. De même, le SQL n'est pas sensible au retour à la ligne. Seul le point virgule marque la fin d'une requête.

> **!** La valeur NULL
>
> Lorsqu'un n-uplet ne doit pas contenir de valeur, SQL fournit une valeur par défaut appelée `NULL`. Cette valeur (comme `None` en Python) marque ainsi l'absence d'information. Par défaut, tout attribut, peu importe son domaine, peut contenir la valeur `NULL`. Cela mène bien souvent à des résultats non prévus, Car la valeur de nul a des comportements spécifiques en fonction des requêtes. Ainsi, il est conseillé de ne pas autoriser la valeur `NULL` (nous verrons comment le faire). Et dans le cas où cette dernière est souhaitée de bien faire attention lors de la création des requêtes.

Création d'une table

Pour créer une relation ou table en SQL, on utilise la syntaxe suivante :

```
CREATE TABLE Nom( attribut1, attribut2, ..., [contraintes1], [contrainte2], .. );
```

```
CREATE TABLE Eleves(nom VARCHAR(100), prenom VARCHAR(100), age INT,
                    PRIMARY KEY (nom, prenom));
```

```
CREATE TABLE Classe_Terminal(numero INT PRIMARY KEY, nombre_eleves INT);
```

Avec cette syntaxe, nous créons ainsi une relation à partir de son schéma.

Les types

Lors de la création, plusieurs informations sont passées. L'une d'entre elles est le type de données ou domaines de l'atribu. Ces domaines dépendent grandement des SGBD. Nous parlerons ici des plus courantes, celles qui sont implémentées dans la plupart des SGBD.

Parmi cela, nous avons :

- Les **booléen* avec `BOOLEAN` avec comme valeur `TRUE`et `FALSE`
- Les données numérique :
  - `INTERGER`ou `INT` pour les entier signer sur 32 bits. (On trouve aussi `BIGINT` et `SMALLINT` pour 64 et 16 bits)
  - `DECIMAL(NB_CHRIFFRE, NB_DECIMAUX)` pour des nombre décimal (nombre a virgule fixe), où le premier élément paramètere représente le nombre total de chiffre utilisé, et le second le nombre de chiffres après la virgule (par exemple `DECIMAL(5,2)` pour 159.63).
  - `FLOAT`(ou `REAL`) pour les nombre flottant en 32 bits et `DOUBLE PRECISION` (ou `DOUBLE`) pour les nombre en 64 bits.
- Les données textuelle :
  - `CHAR(n)` pour représenter une chaine de n charactère. Si la chaine possède moins de caractère des espace sont ajouté. Pour evité cela `VARCHAR(n)` prend en compte des caractère de taille variable allant jusqu'à n.
  - `TEXT` qui se comporte comme `VARCHAR` avec un n fixé au maximum (il peut y avoir en pratique où non une limite).
- Les donées temporelle (ISO 8601) :
  - `DATE`pour représenter un jour avec le format `aaaa-mm-jj`

- `TIME`pour représenter une heure sur le format `HH:MM:SS`
- `DATETIME` pour une date et une heure
- `TIMESTAMP`qui a une date et une heure renseigner de manière automatique

Les options

> **NOT NULL**
>
> Il est possible d'ajouter l'option `NOT NULL` afin de préciser que l'attribut ne peut pas contenir la valeur `NULL`. Cette option se rajoute juste après avoir précisé le type. Il est fortement conseillé d'utiliser cette option le plus de fois que possible afin d'éviter les soucis inerant à la valeur `NULL`.

> **Important**
>
> Aussi tout attribut faisant partie d'une clé devrait avoir l'option `NOT NULL`.

> **DEFAULT**
>
> Il est possible d'utiliser l'option `DEFAULT` suivie d'une valeur. Ainsi, lors de l'ajout d'un enregistrement dans la base de données, l'utilisateur n'a pas besoin de renseigner la valeur prise par un attribut contenant l'option `DEFAULT`. Dans ce cas, la valeur prise sera la valeur renseignée lors de la création du schéma (après le `DEFAULT`).
> Il est utile de préciser que l'option default n'empêche pas une valeur d'être à `NULL`. En effet, lors d'une requête, l'utilisateur peut préciser explicitement que la valeur sera `NULL`. Ainsi, au besoin, il est utile de déclarer les 2 options .

Au final, la définition d'un attribut lors de la création d'une table suit le schéma suivant :

```
nom_attribut type [NOT NULL] [DEFAULT valeur]
```

Les contraintes

En plus des attributs, il est aussi possible de définir lors de la création d'un schéma des coCall of Duty, Modern Warfare ? ntraintes d'intégrité.

> **Note**
>
> Dans toute la suite nous noterons qu'il est possible de donner un nom à une contrainte, en ajoutant `CONSTRAINT nom` devant une contrainte

Contraintes d'unicité

Pour la contrainte d'unicité, nous allons simplement définir quelle est la clé primaire. La syntaxe est la suivante :

```
PRIMARY KEY (attr1, attr2, ...)
```

On indique donc quel sont les attribut formant cette clé. Il est aussi possible de précisé cette clé, si elle n'est composé d'un seul attribut, d'ajouter l'option `PRIMARY KEY`lors de la définition de l'attribut.

> Mise en garde
>
> En SQL, il est possible de ne pas préciser de clé primaire. Cela n'est évidemment pas une pratique encouragée car elle peut entraîner beaucoup de soucis. De plus, comme nous n'avons vu dans la définition, dans une base de données relationnelle, une relation ne doit pas posséder de doublons. Or, sans clé primaire, il n'est pas possible de vérifier la contrainte d'unicité. Ainsi, lors de toutes vos créations de table, il vous sera demandé d'utiliser une clé primaire.

Il est aussi possible de définir d'autres contraintes d'unicité afin de déclarer d'autres clés. Etant donné qu'une seule clé primaire ne peut exister, la syntaxe est différente, mais très proche. On utilise la contrainte dit `UNIQUE` :

```
UNIQUE (attr1, attr2, ...)
```

Contrainte référentielle

Pour définir une contrainte référentielle, il convient de définir une clé étrangère.

Pour cela, la syntaxe est la suivante :

```
FOREIGN KEY (attr1, attr2, ...) REFERENCES Autre_Relation(attr1_autre, attr2_autre, ...)
```

Ainsi, cette contrainte lie des attributs d'une relation (ou table en SQL) courante, avec les attributs d'une relation déjà préalablement définie. L'ordre des attributs de la table actuelle doit correspondre à l'ordre des attributs de la table référencée. Le nom de la relation après référence et le nom de la table référencée déjà existante. Il est à noter que le nom de la table actuelle est le nom de la table référencée ne sont pas forcément les mêmes. Seul l'ordre de la définition des données est important et permettra au SGBD de comprendre quels ailes prévu sont liés entre eux.

Contraint de domaine

Avant de contraindre le domaine, escuel permet d'ajouter une assertion logique que doit vérifier un attribut. Pour cela, nous utilisons la syntaxe `CHECK` :

```
CHECK (assertion)
```

Exemple 0.1.

```
CHECK (tour_taille BETWEEN 30 AND 200)
```

Nous verrons plus de syntaxe, des assertions logiques dans la Section . Retenez simplement la clause `CHECK`.

Modification d'une table

Il peut être utile de modifier une table a posteriori.

Suppression d'une table

Il est par exemple possible de supprimer une table avec la clause `DROP TABLE`:

```
DROP TABLE TShirt;
```

Nous indiquons ainsi le nom de la relation à supprimer.

Modification d'une table

Il est possible de modifier un schéma de plusieurs manières possibles. Ces manières ne sont pas a priori au programme et sont fortement déconseillées. En effet, la modification d'une table a des impacts très importants. Cela impactera possiblement les données stockées, mais aussi les requête créées.

Nous citerons cependant une possibilité, celle d'ajouter une nouvelle contrainte d'intégrité :

```
ALTER TABLE nom
ADD CONSTRAINT nom_contraint CONTRAINTE_A_AJOUTER;
```

La syntaxe de la contrainte à ajouter est la même que les syntaxes vues précédemment.

Modification des données

Une fois notre table relation créée, nous pouvons enfin modifié l'instance.

Ajout d'une donnée

Afin d'ajouter une donnée, nous utilisons la syntaxe suivante :

```
INSERT INTO Relation VALUES (val_attr1, val_attr2, ...);
```

Cette syntaxe utilise le nom de la relation ou table déjà créée, et suppose que l'on ajoute dans l'ordre les valeurs pour tous les attributs.

Cependant, il est possible, notamment dans le cas de l'utilisation d'une option défaut que l'on ne veuille pas préciser toutes les valeurs.

Pour cela, nous pouvons utiliser la syntaxe suivante :

```
INSERT INTO Relation(attr_a, attr_b, ...) VALUES (val_attr_a, val_attr_b, ...)
```

Cette 2e syntaxe a aussi pour intérêt de s'assurer que les valeurs sont insérées dans les bons attributs. Si la relation se venait à changer ou qu'un développeur oublie le schéma, Cette syntaxe générera une erreur ou ajoutera au bon endroit les bonnes valeurs.

C'est donc cette 2e syntaxe, certes plus verbeuse, qui est à utiliser.

Suppression d'une donnée

Pour supprimer une ou plusieurs données, nous devons sélectionner ces données. Pour cela, nous précisons la relation et l'insertion logique que les données à supprimer doivent vérifier. Tout comme précédemment, la syntaxe exacte des assertions logiques seront vues dans la Section . La syntaxe de suppression utilisant la clause `DELETE` est la suivante :

```
DELETE FROM Relation WHERE assertion;
```

Exemple 0.2.

```
DELETE FROM Eleve WHERE nom="Thibault";
```

Modification d'une donnée

Tout comme la suppression, la modification, demande d'abord de sélectionner les données à modifier. Elle demande aussi les attributs à modifier et leur valeur après la modification. Ma syntaxe utilisant `UPDATE` est la suivante :

```
UPDATE Relation
SET attr1=val_attr1, attr2=val_attr2, ...
WHERE assertion;
```

Exemple 0.3.

```
UPDATE TShirt
SET couleur="NOIR"
WHERE matiere="SOIE"
```

Recherche (ou selection) d'une information

Les requêtes les plus courantes dans une base de données sont les sélections qui permettent une recherche d'informations.

Une sélection permet d'obtenir les enregistrements existants dans une instance d'une base de données respectant critères donnés.

La plus simple syntaxe pour la sélection est la suivante :

```
SELECT *
FROM Eleve;
```

Cette dernière permet de retourner tous les enregistrements avec tous leurs attributs de la table `Eleve`. Ici, le caractère `*` sert à indiquer que nous recherchons tous les attributs des enregistrements.

Une syntaxe alternative pour ne demander que certaines colonnes d'une table est la suivante :

```
SELECT attr1, attr2, ...
FROM Relation;
```

C'est Rocket simple n'ont pas d'assertion logique. Ainsi par défaut, tous les enregistrements seront sélectionnés. Pour rajouter une assertion musique, nous allons ajouter la clause `WHERE` suivi de l'assertion logique voulu.

La syntaxe complète d'une sélection typique est donc la suivante :

```
SELECT attr1, attr2, ...
FROM Relation
WHERE assertion
```

Les assertions logiques

Il existe différentes manières d'exprimer une assertion logique :

- Opérateur de comparaison permettant de comparer en terme à une constante à l'aide d'un opérateur parmis : `< >`, `<=`, `>=`, `=`, `<>` (*"différent de"*).
- Un opérateur d'intervalle `BETWEEN low AND high`, marchand sur les constantes numériques mais aussi sur les dates par exemple.
- Un opérateur de comparaison de chaînes de caractères `LIKE` permettant de tester l'existence d'une sous-chaine.
- Un opérateur de test d'appartenance `IN` testant si une valeur appartient à une liste de constantes (exprimer entre parenthèse, `(1,2,3)`).
- Un opérateur de test de *"nullité"* `IS NULL` (ou `IS NOT NULL`), permettant de tester si une valeur est à `NULL`.

> **!** Test de nullité
>
> L'insertion logique `= NULL` n'est pas équivalente au test de nullité. En effet, si l'un des valeurs est mieux, alors l'opérateur de comparaison ne sait pas quoi faire. Il est donc important de régulièrement faire des tests de nullité, pour les attributs n'ayant pas l'option `NOT NULL`.

On a plus de l'opérateur `LIKE`, on peut utiliser le caractère `%` ou le caractère `_`.

Le caractère `%` permet d'indiquer que nous pouvons ajouter un nombre arbitraire (potentiellement infini) de caractères à sa place. Ainsi, `LIKE "a%"`cherche tout les mot commençant par `a`. Et `LIKE "%a"`, tout ceux finissant par `a`. Il est possible d'utiliser plusieurs fois le caractère afin de chercher des sous-chaine plus complexe, par exemple `LIKE "%abc%"` cherchera tous les maux contenant ou en leur sein `abc`, au début, au milieu ou à la fin . Le caractère `_` Se comporte de la même manière que le caractère `%` Mais ne représente qu'un seul caractère.

En plus de cela, il est possible de combiner les assertions logiques à l'aide des opérateurs logiques suivants : `OR`, `AND` et `NOT`, Correspondant à l'opérateur logique OU, ET et NON. Attention à l'ordre des opérateurs. N'hésitez donc pas à utiliser des parenthèses afin de s'assurer de l'ordre voulu.

Exemple 0.4. Par exemple pour récupérer seulement les couleur des TShirt en soie :

```
SELECT couleur
FROM TShirt
WHERE matiere = "SOIE"
```

fonction de calculs

Il peut parfois être utile, non pas de retourner l'ensemble des entités correspondant à la recherche, mais une fonction dans laquelle l'ensemble des entités de la recherche sera l'argument. Par exemple, si nous souhaitons connaître le nombre de couleurs possiblent pour 1 TShirt en soie.

Nous pouvons citer comme fonction :

- `COUNT` Permettant de connaître le nombre de valeurs de l'ensemble.
- `SUM` Permettant de semer les valeurs ensemble.
- `AVG` Calculant la moyenne de l'ensemble.
- `MIN` et `MAX` Retournant respectivement l'élément minimum et maximum de l'ensemble.

Exemple 0.5. Ainsi, pour notre exemple précédent :

```
SELECT COUNT(couleur)
FROM TShirt
WHERE matiere = "SOIE"
```

Le résultat n'est pas l'ensemble des couleurs, mais bien le nombre de couleurs de cet ensemble.

Les agrégats

Un agrégat est un partitionnement de la table en plusieurs sous-tables en fonction d'une condition. Cette condition est les mêmes assertions logiques que précédemment. Une fois ces soutables créés, nous les passons dans les fonctions de calculs définis précédemment. Par exemple, nous pouvons grouper les différentes entités en fonction de leur matière, créant ainsi autant de sous table que de matière. Puis ensuite comptez les couleurs. La roquette résultante serait donc le nombre de couleurs pour toutes les matières de t-shirt.

Les agrégats utilisent la clause `GROUP BY`. Nous n'expliquerons pas ici la syntaxe complète.

Exemple 0.6. Par exemple, afin de réaliser l'exemple précédent.

```
SELECT COUNT(couleur)
FROM TShirt
GROUP BY matiere
```

Il est aussi bien évidemment possible d'utiliser la clause `WHERE` avec les agrégats.

Ordonnancement et unicité du résultat

Il peut aussi être utile de vouloir ordonner les résultats, ou de ne vouloir que des résultats distincts dans le cas par exemple, où l'un des attributs souhaité n'est pas une clé.

Afin de trier les données, nous utilisons la clause `ORDER BY attr sens`

Où `sens` vaut :

- `ASC`pour un tri croissant
- `DESC`pour un tri décroissant

Exemple 0.7. Pour trier les élèves en fonction de leurs notes dans le sens croissant :

```
SELECT prenom, nom
FROM Eleve
ORDER BY moyenne_NSI ASC
```

Dans le cadre où nous aurions plusieurs valeurs possibles, nous pouvons utiliser le mot clé `DISTINCT` juste après la clause `SELECT`

Exemple 0.8. Pour obtenir toutes les matières possibles e manière unique (sans cela nous aurions des répétition) :

```
SELECT DISTINCT matiere
FROM TShirt
```

Jointure

L'un des éléments les plus importants des sélections en SQL sont les jointures. Une jointure permet de fusionner 2 tables qui se référence, grâce aux clés primaires et aux clés étrangères.

Il y a plusieurs types de jointures et nous verrons la jointure la plus modulaire.

Cette dernière utilise la clause `JOIN ... ON`.

La syntaxe est la suivante, Pour créer une table fusionnant la table `Tabl1` et `Tabl2` sur les attributs en commun `attr`.

Exemple 0.9. Par exemple, pour connaître toutes les commandes et leurs adresses, nous pouvons utiliser :

```
SELECt adresse, num_commande
FROM Clients JOIN Commandes ON (Commandes.nom = Clients.nom AND Commandes.prenom = Clients.p
```

Il est possible d'avoir recours à des jointures plus compliquées car il suffit de passer une assertion logique. L'une des utilités, est notamment lorsque une clé étrangère et une clé primaire s'exécutent avec plusieurs attributs.

Opération enssembliste

Il est possible de regrouper les ensembles 2 requête différentes, lorsque leur schéma est compatible, grâce à :

- `Req1 UNION Req2` pour fusionner les 2 résultats
- `Req1 INTERSECT Req2` pour ne garder que les résultats en commun
- `Req1 EXCEPT Req2` Pour supprimer des résultats de la requête 1, les résultats de la requête 2

Requêtes imbriquées

Il est possible de réutiliser le résultat d'une requête comme point de départ pour une autre requête. Cette dernière est utilisée dans la clause `WHERE` afin d'affiner l'assertion logique. Cela peut par exemple être particulièrement utile avec la clause `IN`. Il est aussi possible d'utiliser une requête comme ensemble de départ de la clause `FROM` permettant ainsi, par exemple, un double tri.

Le plus simple pour cela est d'expérimenter par vous même. Cette roquette imbriquée permet notamment de simplifier grandement l'utilisation de certaines requêtes, rendues beaucoup plus compliquées sans cela.

Pour aller plus loin sur les SGBD

Gestion des transacctions

Définition 0.1 (Transactions)**.** Séquence d'opérations liées comportant des mises à jour ponctuelles d'une base de données devant vérifier des propriétés.

En SQL, chaque requête est en général interpréter par le SGBD comme une seule transaction.

Ce comportement peux ne pas être souhaitable. Immaginé que l'on souhaite ajouter une commande d'un nouveau client. On va donc réaliser la création d'un client, puis d'une commande. Si l'une de ces requête échoue, l'autre n'as pas de sens. Hors étant indépendant, on ne connais pas l'ordre d'éxécution a priori, encore moins si ces dernière vont échouer, et encore moins si ces requêtes sont vouer, parce que lier, à l'échec (l'une entraine léchec de l'autre). Ainsi il convindrait de créer une unique transaction pour les deux.

> **Note**
>
> Ainsi une transaction est tout ou rien : soit toute les requête reussise, soit elle doivent toute échouer.

C'est possible. Pour cela on utilise la syntaxe suivante :

```
START TRANSACTION;
requete1;
requete2;
...
COMMIT;
```

Ici tout les requête entre `START TRANSACTION` et `COMMIT` font partie d'une même transaction. Lors que le SGBD voit passer `COMMIT` alors il tente d'appliquer les modifications. Si cela est possible, base est modifié, sinon la transaction est annulé.

Si pour une raison quelconque, vous souhaité annuler la transaction (peut importe si elle peux être réaliser) grâce à la clause `ROLLBACK`.

Le SGBD stocke donc dans une mémoire temporaire les modification de chaque requête, puis à la fin de la transaction, lors du `COMMIT` si tout est cohérent, alors le nouvelle état est écrit définitivement dans la base de données. Sinon les modification sont abandonné, et n'étant pas enregistré, l'état de la base de données est inchangé, et seras le même qu'au début de la transcation, comme si aucune requête n'as été éxécuté.

Propriété d'une base de donées

Comme vu précédement, afin de conserver une base de données cohérente, de nombreux concepteur de SGBD se dote de propriété que leur logiciel doivent respecter, afin de garantir un fonctionement cohérent.

Dans une base de données relationnelle, ces propriétés sont dit **ACID**. Cela est l'acronyme des 4 propriétés :

- **Atomiciter** : C'est le principe selon laquelle une transaction doit effectuer une mise à jour ou ne doit pas modifier la base de données. La transaction est ainsi réalisée dans son intégralité ou n'est pas réalisée du tout en cas de menace pour la base de données.
- **Cohérence** : La transaction doit respecter la cohérence d'une base de données. Si la transaction ne permet pas de respecter cette cohérence, alors elle ne doit pas être réalisée. Cela peut arriver non seulement lors d'une mauvaise manipulation, d'une requête, mais aussi, par exemple, lors d'un accès concurent et conflictuel.
- **Isolation** : Le résultat d'une transaction d'eau doit être visible aux autres transactions qu'une fois cette dernière validée. Cela permet notamment d'éviter les interférences. Ce principe remet en cause certains accès concurrents. Cela postule, par exemple, que 2 transactions effectuées de manière concurrente doivent produire le même effet que s'ils avaient été exécutés de manière séquentiel.
- **Durabilité** : Une transition est validée L'état de la base de données doit être modifié. Cela inclut par exemple que le système doit garantir que les données sont conservées en cas de panne d'un disque par exemple.