LYCÉE LECONTE DE LISLE

Bases de données

Vincent Picard

1
Base de donnée relationnelle

Base de donnée relationnelle

- Une **base de donnée** permet de stocker un grand nombre d'informations sous une forme **structurée** qui garantit l'intégrité des informations conservées et permet efficacement d'en récupérer les données voulues.
- Une base de donnéee peut être **interrogée** à l'aide de requêtes qui permettent d'extraire les informations recherchées. Au programme, on étudiera le **langage de requêtes SQL** (*structured query language*)
- On étudiera uniquement les **bases de données relationnelles** c'est-à-dire structurées sous forme de plusieurs **tables** (ou **relations**)

Qu'est-ce qu'une table ?

Voici un exemple de table appelée `Produit` :

Produit		
`idProduit`	`Designation`	`Prix`
1	Savoyarde	13.00
2	Royale	9.00
3	Forézienne	12.00
4	Bretonne	11.00
5	Océane	12.00
6	Végétarienne	11.00
7	Canicatti	13.00

- Les **colonnes** correspondent aux **attributs** de la table. Chaque attribut possède un nom. Ici : `idProduit`, `Designation` et `Produit`.
- Les valeurs de chaque attribut sont d'un **type** bien défini appelé **domaine**. Ici : entier (idProduit), chaîne (Designation), flottant (Prix)
- Les **lignes** correspondent aux **enregistrements** de la table. Chaque enregistrement est une donnée sous forme d'un **n-uplet** qui contient la valeur prise pour chaque attribut. Ici, les enregistrements sont les pizzas proposées par le restaurant.

Autres tables de la base de donnée `pizza.sqlite`

- En plus de la table `Produit`, la base de donnée de gestion de pizzeria comporte les tables suivantes :

Client			
`idClient`	`Nom`	`Prénom`	`Adresse`
1	Dumbledore	Albus	28 av. de la Libération
2	Weasley	Fred	Chemin de traverse
3	McGonagall	Minerva	4 rue Privet Drive
4	Riddle	Tom	Allée des embrumes
5	Weasley	Georges	Chemin de traverse
6	Diggory	Cédric	Le terrier

Livreur		
`idLivreur`	`Nom`	`Prénom`
1	Granger	Hermione
2	Potter	Harry
3	Weasley	Ron

Véhicule		
`idVéhicule`	`Type`	`Kms`
1	Scooter	25000
2	Voiture	42000
3	Scooter	17000
4	Scooter	5000

- Et enfin une table pour enregistrer le traitement des commandes de Pizza :

Commande							
`id`	`idClient`	`idProduit`	`idLiveur`	`idVehicule`	`Date`	`Heure`	`Livraison`
1	1	7	2	4	13-04	19:00	19:45
2	5	2	1	2	13-04	18:00	18:30
3	3	3	3	4	14-04	12:00	`NULL`
4	5	1	2	1	14-04	12:00	12:30
5	4	6	2	2	14-04	14:00	`NULL`
6	1	5	1	4	14-04	14:00	14:30
...							

- Remarquer qu'on utilise les numéros de client, de livreur et de produit pour se référer aux informations des autres tables.

Clés primaires, clés étrangères

Pour une table donnée :

- Une **clé primaire** est un attribut, ou un ensemble d'attributs, qui permettent d'identifier un enregistrement de manière unique dans cette table.

Candidat		
`id`	`Nom`	`Prénom`
84302	Holmes	Sherlock
18391	Fletcher	Jessica
19385	Columbo	Franck

  - `id` est une clé primaire
  - `Nom` n'est pas une clé primaire
  - `(Nom, Prénom)` n'est pas une clé primaire

- Une **clé étrangère** est un attribut qui sert à faire référence à un enregistrement d'une autre table.

Convocation		
`idConvoc`	`Candidat`	`Epreuve`
789Y	18391	Info B
981C	18391	Maths 2
171D	84302	Physique 1

  - `idConvoc` est une clé primaire
  - `Candidat` n'est pas une clé primaire
  - `Candidat` est une clé étrangère faisant référence à un candidat précis de la table Candidat via sa clé primaire.

- Attention, on remarque qu'un nom d'attribut peut correspondre à une clé primaire dans une table et à une clé étrangère dans une autre table : il faut bien comprendre de quelle table on parle.

Schéma relationnel

- Le **schéma relationnel** d'une base de donnée est sa structure sous forme d'une liste de tables pour lesquelles on précise les attributs, les domaines, et si possible les clés primaires et les clés étrangères.
- Le schéma de la base de données complète de gestion de la pizzeria est le suivant.

  Client(idClient ENTIER, Nom TEXTE, Prénom TEXTE, Adresse TEXTE)

  Produit(idProduit ENTIER, Désignation TEXTE, Prix FLOTTANT)

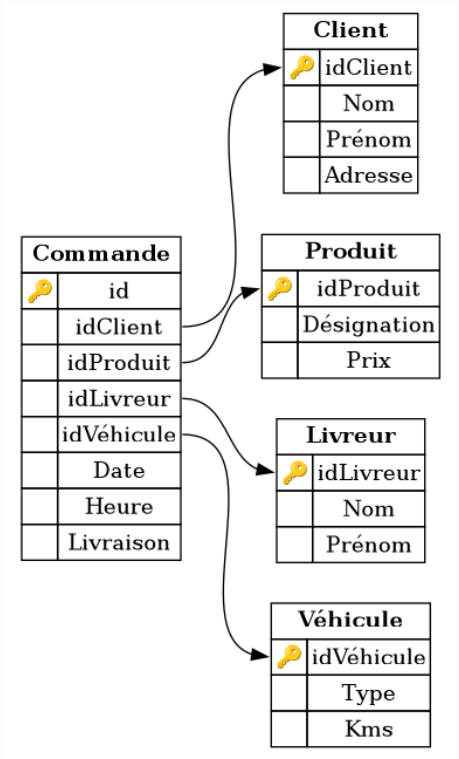
  Livreur(idLivreur ENTIER, Nom TEXTE, Prénom TEXTE)

  Véhicule(idVéhicule ENTIER, Type TEXTE, Kms ENTIER)

  Commande(id ENTIER, *idClient* ENTIER, *idProduit* ENTIER, *idLivreur* ENTIER, *idVéhicule* ENTIER, Date DATE, Heure HEURE, Livraison HEURE)

  Nous avons décidé de souligner les clés primaires et d'écrire en italique les *clés étrangères*.

Schéma relationnel : avec un diagramme !



2
Requêtes en SQL : sur une seule table

Langage SQL

- Le langage SQL permet d'écrire informatiquement des **requêtes** portant sur une base de donnée relationnelle.
- Le résultat d'une requête SQL se présente sous la forme d'une table.
- Le format type d'une requête SQL est le suivant :

```
SELECT colonnes FROM table WHERE condition;
```

  - `colonnes` : liste des colonnes que l'on souhaite afficher dans le résultat. Si on veut toutes les colonnes on écrit `*`
  - `table` : nom de la table sur laquelle on travaille
  - `condition` : test booléen portant sur les attributs de la table. Dans le résultat on ne garde que les enregistrements pour lesquels le test vaut vrai
  - la partie `WHERE condition` est facultative, si elle est omise alors toutes les lignes sont conservées dans le résultat.

- Exemple :

```
SELECT Designation, Prix FROM Produit WHERE Prix > 10.0;
```

Opérations unaires sur une table

- L'opération consistant à ne conserver qu'un ensemble de colonnes s'appelle la **projection**, on l'obtient avec une requête de type :

```
SELECT col1, col2, ..., coln FROM table;
```

- L'opération consistant à ne conserver que les lignes d'une table vérifiant une condition s'appelle la **sélection**, on l'obtient avec une requête de type :

```
SELECT * FROM table WHERE condition;
```

- L'opération consistant à **renommer** les noms de certains attributs s'obtient avec le mot clé `AS` :

```
SELECT col1 AS nom1, col2 AS nom2, ..., coln AS nomn FROM table
```

- En SQL, il est possible de combiner toutes ces opérations en une seule requête comme on l'a montré précédemment.

Expressions booléennes

- Pour écrire les tests booléens en SQL on peut utiliser :
  - le test d'**égalité** : `=`
  - le test de **non égalité** : `<>`
  - les **comparaisons** : `<`, `<=`, `>`, `>=`
  - les **opérateurs booléens** : `AND`, `OR`, `NOT`
- Exemple :

```
SELECT nom, prenom, age, taille FROM Patient WHERE age >= 18 AND taill
```

Opérations ensemblistes sur les résulats

- Lorsque le résultat de deux requêtes sont **compatibles** : c'est-à-dire comporte le même ensemble d'attributs, on peut réaliser des opérations ensemblistes.
  - `UNION` : qui réalise l'**union** des enregistrements des 2 résultats
  - `INTERSECT` : qui réalise l'**intersection** des enregistrements des 2 résultats
  - `EXCEPT` : qui réalise la **différence ensembliste** des enregistrements des 2 résultats
- La **syntaxe** est la suivante :

```
requete1
UNION / INTERSECT / EXCEPT
requete2;
```

- **Exemple :** nom et prénoms des personnes qui fréquentent le restaurant (livreurs ou clients) :

```
SELECT Nom, Prénom FROM Client
UNION
SELECT Nom, Prénom FROM Livreur;
```

Élimination des doublons

- Par défaut, SQL garde dans le résultat des requêtes les lignes ayant mêmes valeurs.
- Utiliser `SELECT DISTINCT` au lieu de `SELECT` dans une requête SQL permet d'obtenir une table de résultat ne comportant pas de doublon.
- Exemple : on veut la liste sans doublon des prénoms des clients du restaurant

```
SELECT DISTINCT prenom FROM Client;
```

Tri des résultats

- Il est possible de trier les lignes du résultat par ordre de valeurs croissantes d'une certaine colonne :

```
SELECT ... FROM ...  WHERE ... ORDER BY col ASC;
```

- ou par ordre decroissant des valeurs dans la colonne choisie :

```
SELECT ... FROM ... WHERE ... ORDER BY col DESC;
```

- Si l'ordre n'est pas précisé, SQL considère par défaut qu'il s'agit de l'ordre est croissant (`ASC`)

- Exemple : liste des pizzas par ordre décroissant de prix :

```
SELECT Designation AS Pizza, Prix FROM Produit ORDER BY Prix DESC;
```

Troncature des résultats

- Il est possible de demander de ne retourner que les n premiers enregistrements du résultat avec le mot clé `... LIMIT n`
- Exemple : les 3 pizzas les plus chères :

```
SELECT Designation, Prix FROM Produit ORDER BY Prix DESC LIMIT 3;
```

- En complément de `... LIMIT n` on peut aussi demander d'afficher les n premiers résultats en ignorant les k premiers avec `... LIMIT n OFFSET k`:

```
SELECT Designation, Prix FROM Produit ORDER BY Prix DESC LIMIT 3 OFFSET 2;
```

affichera les 3e, 4e et 5e pizza par ordre de prix.

3
Requêtes en SQL : sur plusieurs tables

Requêtes sur plusieurs tables

- Lorsque la requête nécessite de croiser les informations provenant de plusieurs tables, les opérations précédentes ne suffisent plus.
- Exemple : "somme totale dépensée par Dumbledore dans la pizzeria ?"
- Deux opérations sont à connaître :
  - La **produit cartésien** de tables qui réalise tous les croisements possibles : peu utilisée en pratique...
  - La **jointure** de tables qui réalise les croisements vérifiant une certaine condition : **très utilisée** en pratique !

Le produit cartésien de tables

Le **produit cartésien** de tables T_1 et T_2 noté $T_1 \times T_2$ contient tous les n-uplets de la forme :

$$(a_1, \ldots, a_p, b_1, \ldots b_q)$$

où $(a_1, \ldots, a_p)$ est un n-uplet de la table T_1 et $(b_1, \ldots, b_q)$ un n-uplet de la table T_2.

- Intuition : on réalise tous les croisements possibles !

Livreur		
`idLivreur`	`Nom`	`Prénom`
1	Granger	Hermione
2	Potter	Harry
3	Weasley	Ron

Véhicule		
`idVéhicule`	`Type`	`Kms`
1	Scooter	25000
2	Voiture	42000
3	Scooter	17000
4	Scooter	5000

idLivreur	Nom	Prénom	idVéhicule	Type	Kms
1	Granger	Hermione	1	Scooter	25000
1	Granger	Hermione	2	Voiture	42000
1	Granger	Hermione	3	Scooter	17000
1	Granger	Hermione	4	Scooter	5000
2	Potter	Harry	1	Scooter	25000
2	Potter	Harry	2	Voiture	42000
2	Potter	Harry	3	Scooter	17000
2	Potter	Harry	4	Scooter	5000
3	Weasley	Ron	1	Scooter	25000
3	Weasley	Ron	2	Voiture	42000
3	Weasley	Ron	3	Scooter	17000
3	Weasley	Ron	4	Scooter	5000

Produit cartésien en SQL

- Pour travailler sur le produit cartésien de T_1 et T_2 on écrit en SQL

```
SELECT ... FROM T1, T2 WHERE ...
```

- Exemple : réaliser simplement le produit cartésien précédent s'obtient par :

```
SELECT * FROM Livreur, Véhicule;
```

- Le produit cartésien se généralise à n tables :

```
SELECT ... FROM T1, T2, ..., Tn WHERE ...
```

- Le produit cartésien est rarement utilisé car il réalise tous les croisements de n-uplets possibles ce qui n'a souvent pas de sens en pratique.

La notation `table.attribut`

- Lorsqu'un même nom d'attribut est utilisé dans deux tables différentes cela peut conduire a des ambiguités.
- Exemple : dans la table `Client` et la table `Livreur` il y a des attributs `Nom` et `Prénom`...
- Cela est particulièrement problématique lorsqu'on calcule des produits, par exemple :

```
SELECT * FROM Client, Livreur WHERE Nom='Weasley';
```

le test porte-t-il sur le nom de livreur ou de client ?

- Pour lever l'ambiguité on utilise la syntaxe `table.attribut` pour préciser de quel attribut on parle :

```
SELECT * FROM Client, Livreur WHERE Client.Nom='Weasley';
```

La jointure de tables

- Soit θ une propriété portant sur les attributs de la base de donnée.

La **θ-jointure** de tables T_1 et T_2 notée $T_1 \bowtie_\theta T_2$ contient tous les n-uplets de la forme :

$$(a_1, \dots, a_p, b_1, \dots b_q)$$

où

- $(a_1, \dots, a_p)$ est un n-uplet de la table T_1
- $(b_1, \dots, b_q)$ un n-uplet de la table T_2
- $(a_1, \dots, a_p, b_1, \dots b_q)$ vérifie la condition θ

- C'est un produit cartésien amélioré !
- La propriété θ est appelée **condition de jointure** : elle sert à ne conserver que les croisements de données qui ont un sens.

Jointure : exemple

- On veut une liste des commandes qui comporte les noms de pizza commandées
- Il faut croiser les informations de la table `Commande` et de la table `Produit`.
- On calcule donc la θ-jointure suivante :

$$\text{Commande} \bowtie_{\text{Commande.idProduit}=\text{Produit.idProduit}} \text{Produit}$$

Commande $\bowtie_{\text{Commande.idProduit}=\text{Produit.idProduit}}$ Produit										
id	idClient	Commande.idProduit	idLiveur	idVehicule	Date	Heure	Livraison	Produit.idProduit	Designation	Prix
1	1	7	2	4	13-04	19:00	19:45	7	Canicatti	13.00
2	5	2	1	2	13-04	18:00	18:30	2	Royale	9.00
3	3	3	3	4	14-04	12:00	NULL	3	Forézienne	12.00
4	5	1	2	1	14-04	12:00	12:30	1	Savoyarde	13.00
5	4	6	2	2	14-04	14:00	NULL	6	Végétarienne	12.00
6	1	5	1	4	14-04	14:00	14:30	5	Bretonne	11.00
...										

La jointure en SQL

- Pour travailler sur la θ-jointure de T_1 et T_2 on écrit en SQL

```
SELECT ... FROM T1 JOIN T2 ON theta WHERE ...
```

- Exemple : la jointure précédente s'obtient par :

```
SELECT * FROM Commande JOIN Produit ON Commande.idProduit=Pro-
duit.idProduit;
```

- La jointure se généralise à n tables :

```
SELECT ... FROM T1 JOIN T2 JOIN  ... JOIN Tn ON theta WHERE ...
```

- Comprendre les jointures est essentiel pour écrire des requêtes en SQL !
- **Souvent la condition de jointure consiste à faire correspondre une clé étrangère avec la clé primaire à laquelle elle fait référence.**

4
Calculs et fonctions d'agrégation

Exemple de travail

- On présentera les concepts sur la table suivante :

Pays	Continent	Habitants	PIB
France	Europe	67	2775
Royaume-Uni	Europe	66	2828
Allemagne	Europe	83	4000
Espagne	Europe	47	1425
États-Unis	Amérique	333	20494
Canada	Amérique	38	1711
Mexique	Amérique	128	1223

- Ce qui sera dit s'applique aussi pour une table construite à partir de jointures ou de produits cartésien de plusieurs tables.

Calculs sur les colonnes

- Il est possible de créer une colonne contenant le résultat d'un calcul effectué sur les attributs de la table.
- Exemple : obtenir le PIB / Habitant de chaque pays :

```
SELECT Pays, PIB / Habitants FROM Economie;
```

- On obtient le résultat suivant :

Pays	PIB / Habitants
France	41.42
Royaume-Uni	42.84
Allemagne	48.19
Espagne	30.31
États-Unis	61.54
Canada	45.03
Mexique	9.55

Fonctions d'agrégation

- Les **fonctions d'agrégation** servent à effectuer des opérations qui **agrègent** plusieurs lignes en une seule.
- C'est souvent le cas pour les **fonctions statistiques**.
  - par exemple, sommer les valeurs d'un ensemble de lignes va conduire à une seule valeur résultat sur une seule ligne.
- Au programme, vous devez savoir utiliser :
  - `SUM` : qui permet de sommer les valeurs d'un attribut
  - `AVG` : qui permet de moyenner les valeurs d'un attribut
  - `MIN` : qui permet de trouver la valeur minimale d'un attribut
  - `MAX` : qui permet de trouver la valeur maximale d'un attribut
  - `COUNT` : qui permet de compter un nombre de lignes

Sommes

- Pour sommer les valeurs de toutes les lignes d'un attribut, on créé une colonne nommée `SUM(attribut)`.
- `SUM` est une **fonction d'agrégation** : si la table comporte n lignes le résultat n'en comportera plus qu'un !
- **Exemple :** calculer la population mondiale

```
SELECT SUM(Habitants) FROM Economie;
```

- On obtient alors :

SUM(Habitants)
762

- Il peut être agréable de renommer la colonne concernée :

```
SELECT SUM(Habitants) AS PopMondiale FROM Economie;
```

Moyennes

- Pour calculer une moyenne des valeurs d'un attribut, on utilise la **fonction d'agrégation** `AVG` qui fonctionne de la même manière que `SUM`.
- **Exemple** : nombre moyen d'habitants par pays

```
SELECT AVG(Habitants) FROM Economie;
```

Minimum et maximum

- La **fonction d'agrégation** `MIN(attribut)` permet à partir de n lignes de ne conserver qu'une seule ligne pour laquelle la valeur de l'`attribut` est minimale.
- **Exemple :** plus petite population

```
SELECT MIN(Habitants) FROM Economie;
```

MIN(Habitants)
38

- Il peut être utile de savoir en plus pour quel pays ce minimum est atteint : pour ce la on peut afficher d'autres attributs de la ligne résultat :

```
SELECT Pays, MIN(Habitants) FROM Economie;
```

Pays	MIN(Habitants)
Canada	38

- La fonction `MAX` fonctionne de la même manière.

Comptage avec `COUNT`

- La **fonction d'agrégation** `COUNT` permet de compter le nombre de lignes de la table. Elle possède plusieurs **variantes** :
  - `COUNT(*)` : compte simplement le nombre de lignes (noter qu'il est alors inutile de préciser un attribut)
  - `COUNT(attribut)` : compte le nombre de lignes pour lesquelles la valeur de `attribut` n'est pas `NULL`
  - `COUNT(DISTINCT attribut)` : compte le nombre de valeurs distinctes dans la colonne attribut.

- **Exemples**
  - Nombre de pays dans le monde :

```
SELECT COUNT(*) FROM Economie;
```

  - Nombre de commandes effectivement livrées dans la Pizzeria :

```
SELECT COUNT(Livraison) FROM Commande;
```

  - Nombre de noms de famille de clients :

```
SELECT COUNT(DISTINCT Nom) FROM Client;
```

Groupage avec GROUP BY

- La syntaxe

```
SELECT ... FROM ... WHERE ... GROUP BY attribut;
```

  permet de **regrouper** les lignes ayant même valeur pour `attribut`

- En présence de `GROUP BY` les fonction d'agrégation agissent **groupe par groupe** au lieu de s'appliquer sur la table en entier.

- **Exemple :** nombre d'habitants pour chaque continent :

```
SELECT SUM(Habitants) FROM Economie GROUP BY Continent;
```

SUM(Habitants)
263
499

- Le résultat possède deux lignes (appelées **agrégats**) car 2 groupes on été formés : 1 pour les lignes où `Continent` vaut `Europe`, et 1 pour les lignes où `Continent` vaut `Amérique`

- Lorsqu'on utilise `GROUP BY attribut` il est conseillé d'afficher également la colonne `attribut` afin de pouvoir distinguer les agrégats dans le résultat.

- **Exemple**

```
SELECT Continent, SUM(Habitants) FROM Economie GROUP BY Conti-
nent;
```

Continent	`SUM(Habitants)`
Europe	263
Amérique	499

Filtrage des agrégats avec HAVING

- **Une fois les fonctions d'agrégation appliquées** on peut vouloir filtrer les agrégats obtenus, cela s'obtient avec la syntaxe

```
SELECT ... FROM ... WHERE ... GROUP BY ... HAVING condition;
```

- **Exemple :** Liste des continents ayant une population totale supérieure à 300 millions d'habitants.

```
SELECT Continent, SUM(Habitants) FROM Economie GROUP BY Continent HAVING SUM(Habitants) >= 300;
```

Continent	SUM(Habitants)
Amérique	499

- **Attention :** ne pas confondre `WHERE` qui filtre les lignes **avant** d'appliquer les fonctions d'agrégation et `HAVING` qui filtre les agrégats !

Dans quel ordre SQL travaille-t-il ?

```
SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ... OR-
DER BY ... LIMIT ... OFFSET ...
```

1. `FROM` : choix de la table, ou des tables et calcul des jointures et produits cartésiens
2. `WHERE` : filtrage des lignes
3. `GROUP BY` : former les groupes de lignes
4. Calculs des fonctions d'agrégation sur chaque groupe
5. `HAVING` : filtrage des agrégats
6. `SELECT` : choix des colonnes à afficher
7. `ORDER BY` : tri du résultat
8. `LIMIT/OFFSET` : troncature du résultat

Exemples plus complexes

- Classer les livreurs par recette totale récupérée sur les commandes du midi (entre 12h et 14h).

```
SELECT Prénom, Nom, SUM(Prix) AS Recettes
FROM Commande JOIN Livreur JOIN Produit
ON Commande.idLivreur = Livreur.idLivreur
AND Commande.idProduit = Produit.idProduit
WHERE Heure >= '12:00' AND Heure <= '14:00'
GROUP BY Livreur.idLivreur
ORDER BY Recettes DESC;
```

Prénom	Nom	Recettes
Harry	Potter	24.0
Hermione	Granger	12.0
Ron	Weasley	12.0