Base de données

PSI/PSI⋆

I Introduction

Une base de données est une structure logicielle permettant de *sauvegarder* des données, souvent en grande quantité, avec les objectifs suivants :

- ▷ centraliser l'information
- ▷ accéder aux données de manière *isotrope*
- ▷ assurer l'indépendance des données et des traitements
- ▷ permettre des liaisons entre ensembles de données
- ▷ assurer l'intégrité des données
- ▷ assurer la sécurité des données
- ▷ permettre le traitement des données de manière partagée

⚛ Exemples.

- Données bancaires : clients, comptes, transactions, …
- Données d'un réseau social : utilisateurs, contenu, relations, …
- Données des ressources d'une entreprise : employés, services, matériel, identifiant, …
- Données d'un établissement scolaire : élèves, personnel, classes, notes, …
- Données scientifiques : acquisitions, résultats, …
- Données d'un jeu vidéo en ligne : utilisateurs, équipement, statistiques, …

🐾 🐾 🐾

Les données d'une base de données peuvent être structurées, semi-structurées ou brutes. Voici trois modèles de bases de données structurées :

- Modèle hiérarchique (basé sur des structures arborescentes)
- Bases de données réseaux (basées sur des structures en graphe ou en réseaux)
- Modèle relationnel (basé sur la théorie mathématique des relations)

Une base de données *relationnelle* est une base de données dans laquelle les données sont organisées sous forme de *relations* ou *tables*.

Un système de gestion de bases de données (SGBD) est un logiciel permettant d'*archiver*, de *retrouver*, de *mettre à jour*, de *traiter*, et plus généralement d'interagir avec les données sauvegardées dans une base de données.

⚛ Exemples.

MariaDB, MySQL, PostgreSQL, SQLite, MongoDB, Voldemort, …

🐾 🐾 🐾

II Tables (Relations)

Le terme *relation* vient du *modèle relationnel* sur lequel sont basées l'organisation et l'implémentation des bases de données relationnelles. Une relation est représentée sous la forme d'un tableau à deux dimensions, d'où le nom plus commun en pratiquer de *tables*.

❦ Définition.

Une *table* est représentée sous la forme d'un tableau à deux dimensions où chaque colonne correspond à un *attribut* et chaque ligne, appelée *enregistrement*, correspond à un objet de la relation. Les attributs consistent en un *nom*, que le décrit, et un *type de données*.

⚛ Exemples.

La table élève ci-dessous contient trois enregistrements décrits par quatre attributs.

nom	prenom	classe	note
Insuline	Nadine	T12	12
Velociraptor	Allan	T12	18
Caféine	Nadine	T7	10

❧ ❧ ❧

❦ Définition.

Le schéma d'une relation est la liste de ses attributs, *i.e.* leur nom et leur type.

⚛ Exemples.

Pour la table ci-dessus :

— `nom` (`VARCHAR`, chaîne de caractère)
— `prenom` (`VARCHAR`)
— `classe` (`VARCHAR`)
— `note` (`INT`)

❧ ❧ ❧

Dans les tables, les *clés* sont un moyen d'identifier de manière unique des éléments.

❦ Définition.

Une clé *primaire* est un *ensemble d'attributs* fixe et non vide, identifiant chaque enregistrement de manière unique.

⚛ Exemples.

- Le numéro d'étudiant d'un élève
- Le numéro de compte d'un compte bancaire

- Le numéro de sécurité sociale d'un salarié
- La longitude, la latitude et la date d'un séisme (3 attributs)
- La date, la maternité, le nom, le prénom, le ou les parents d'une naissance

Définition.

Une clé *étrangère* est un ensemble d'attributs d'une table $\mathcal{R}_1$ qui fait référence à un ensemble d'attributs *identiques, constituant une clé primaire* d'une autre table $\mathcal{R}_2$.

On dit que la table $\mathcal{R}_1$ référence la table $\mathcal{R}_2$.

Pour chaque enregistrement de $\mathcal{R}_1$, le tuple de valeurs des attributs constituant la clé étrangère doit être présent dans $\mathcal{R}_2$.

Exemples.

Soit une première table `reseau` ayant pour schéma :

— `adresse` (`VARCHAR`)—clé étrangère
— `id_gaz` (`INT`)
— `id_eau` (`INT`)
— `id_electricite` (`INT`)

Soit la table `kebab` qui référence `reseau` et ayant pour schéma :

— `id` (`INT`)—clé primaire
— `adresse` (`VARCHAR`)—clé étrangère faisant référence à `reseau.adresse`.
— `note` (`INT`)

III Requêtes SQL de Recherche sur *une* Table

On se concentre exclusivement sur le langage SQL utilisé par de nombreux SGBD. Seules les requêtes de recherche dans la base de données sont abordées.

Dans la suite, un élément de syntaxe écrit entre crochets (`[ ... ]`) correspond à un élément de syntaxe optionnel.

Une requête se construit, de manière non-linéaire, comme un ensemble de *clauses* et se termine par un `;`.

Pour un résumé de la syntaxe *c.f.* l'annexe et le diagramme de syntaxe SQL en fin de ce document. Ici est détaillée la sémantique des différentes clauses.

Soit `R` une table.

3.1 Base et Projection

Une requête de recherche commence *toujours* par une clause suivant le modèle suivant :

```
SELECT e_1 [AS a_1], e_2 [AS a_2], ..., e_m [AS b_m] FROM R ;
```

avec $\{e_1, ..., e_m\}$ des expressions sur les attributs de `R` et $\{a_1, ..., a_m\}$ d'éventuels noms pour désigner le résultat de ces expressions.

Le plus souvent, les expressions sont simplement l'attribut recherché. On appelle cette opération une *projection*.

```
SELECT login, password FROM users ;
```

Le mot clé **AS** permet de nommer le résultat d'une expression. Il donne plus de lisibilité ou permet d'utiliser le résultat d'un calcul dans une autre clause de la requête évaluée après le calcul.

```
SELECT compte1 AS courant, compte2 AS epargne, nom, prenom FROM clients ;
```

Le langage SQL supporte les calculs sur les entiers et les flottants, un certain nombre de fonctions mathématiques, la concaténation sur les chaînes de caractères, la conversion de type, etc. (*c.f. annexe pour une liste non exhaustive*)

```
SELECT eleve,
       (note_math + note_info * 12 + note_physique) / 14 AS moyenne,
       FROM eleves ;
SELECT CONCAT(nom, '␣', prenom) FROM employe ;
```

On peut utiliser le symbole * pour obtenir, sans renommage, tous, les attributs de chaque enregistrement.

```
SELECT * FROM seismes ;
SELECT *, ( note_math + note_info ) / 2 AS moyenne FROM eleves ;
```

On peut ajouter le mot clé **DISTINCT** pour éliminer les éventuels doublons du résultat.

```
SELECT DISTINCT client FROM comptes ;
```

3.2 Sélection

Une sélection consiste à renvoyer uniquement les enregistrements vérifiant une condition introduite dans la clause **WHERE**.

```
SELECT ... FROM R WHERE condition;
```

La syntaxe des conditions pourra être trouvée en annexe.

```
SELECT login, password FROM users
       WHERE login = "Nadine" AND password = "azerty" ;
SELECT eleve,
       (note_math + note_info * 12 + note_physique) / 14 AS moyenne,
       FROM eleves WHERE moyenne >= 12;
```

3.3 Agrégats

Après sélection et avant projection et calculs, une clause **GROUP BY** peut servir à regrouper des enregistrements en paquets, dénommés *agrégats*.

```
SELECT F_1 [AS a_1], ..., F_m [AS a_m]
       FROM R
       [ WHERE condition ]
       GROUP BY e_1, ..., e_n
       [ HAVING condition ];
```

$\{e_1, ..., e_n\}$ sont des expressions sur les attributs de R. Les enregistrements d'un même agrégat partagent une valeur commune sur le résultat de $(e_1, ..., e_n)$.

$F_1, ..., F_m$ sont des expressions basées uniquement sur :

- une des expressions $e_1, ..., e_m$ commune à chaque agrégat
- une fonction d'agrégation, F, qui prend en entrée une ou plusieurs expressions sur les attributs de R, $b_1, ..., b_q$ et qui effectue un calcul sur les colonnes $b_1, ..., b_q$ de chaque agrégat

Une liste non exhaustive des fonctions d'agrégation est donnée en annexe.

```
SELECT classe, AVG(note_info) FROM eleve GROUP BY classe ;
SELECT annee, ville, COUNT(*), MAX(magnitude)
       FROM seismes
       WHERE magnitude >= 3
       GROUP BY annee, ville ;
SELECT niveau, COUNT(DISTINCT user), SUM(point * niveau_classe) / 12
       FROM score
       GROUP BY niveau, FLOOR(point / 100) ;
```

Une clause **WHERE** fait une sélection *en amont* de l'agrégation. On peut faire une sélection *en aval* de l'agrégation avec la clause **HAVING** qui peut utiliser les alias ainsi que des fonctions d'agrégations.

```
SELECT ville, MAX(magnitude) AS max_mag
       FROM seisme
       GROUP BY ville
       HAVING max_mag >= 7 AND COUNT(*) > 10 ;
```

3.4 Tri et Troncature

Avant ou après agrégation, on peut ajouter une clause **ORDER BY** pour trier les enregistrements du résultat selon une ou plusieurs expressions basées sur des attributs, ou des fonctions d'agrégation, si agrégation il y a eu. Le tri sera toujours effectué après sélection, agrégation et calculs éventuels. Le tri a pour utilité possible : la présentation des résultats finaux ou une troncature éventuelle.

```
SELECT ...
       FROM R
       [ WHERE condition ]
       [ GROUP BY ... [ HAVING condition ] ]
       ORDER BY e_1, ..., e_m ;
```

```sql
SELECT nom, moyenne FROM eleve WHERE moyenne >= 12 ORDER BY nom ;
SELECT ville, MAX(magnitude) AS max_mag
       FROM seisme
       GROUP BY ville
       HAVING max_mag >= 7
       ORDER BY max_mag, COUNT(*)
```

Par défaut le tri est fait dans l'ordre croissant. On peut ajouter le mot clé **DESC** pour un tri dans l'ordre décroissant.

```sql
SELECT user, points, FROM score ORDER BY points DESC ;
```

On peut tronquer le résultat renvoyé par une requête. Les mots clés **LIMIT** `n` et `OFFSET m` permettent respectivement de limiter le nombre d'enregistrements du résultat à `n` et d'ignorer les `m` premiers. À noter que `OFFSET` ne peut être utilisé qu'en complément de **LIMIT**.

```sql
SELECT ...
       FROM R
       [ WHERE condition ]
       [ GROUP BY ... [ HAVING condition ] ]
       [ ORDER BY e_1, ..., e_m ]
       LIMIT p [ OFFSET q ];
```

```sql
SELECT amplitude FROM signal ORDER BY amplitude LIMIT 12 OFFSET 21 ;
```

IV Requête SQL de Recherche *entre* Tables

4.1 Jointures

Le produit cartésien d'une table `R1` avec une seconde table `R2` consiste en une table `R` dont les colonnes sont celles de `R1`, puis celles de `R2`, et dont les enregistrements sont exactement ceux dont les colonnes de `R1` correspondent à un enregistrement de `R1` et les colonnes de `R2` à un enregistrement de `R2`

Le produit cartésien des deux tables suivantes …

nom	prenom
Insuline	Nadine
Velociraptor	Allan

classe	effectif
T12	42
T7	12

… est la suivante.

nom	prenom	classe	effectif
Insuline	Nadine	T12	42
Velociraptor	Allan	T7	12
Insuline	Nadine	T7	12
Velociraptor	Allan	T12	42

On peut réaliser le produit cartésien de plusieurs tables en SQL

```sql
SELECT ... FROM R1, R2, ..., Rn;
```

Toutefois, le produit cartésien n'est en pratique jamais utilisé. Il est en effet très rare que l'on souhaite générer l'ensemble des combinaisons d'enregistrement possible. Dans le cas d'une sélection, il est possible d'améliorer les performances en utilisant une *jointure*.

Une jointure (interne) entre deux tables est une opération permettant de combiner deux tables autour d'un attribut (plus généralement d'une condition). Les lignes du résultat de la jointure de `R1` avec `R2` sur la condition `c` correspondent aux enregistrements du produit cartésien de `R1` avec `R2` qui vérifient la condition `c`. Dans le cas de la jointure autour d'un attribut `a`, il s'agit des enregistrements de `R1` et `R2` qui ont en commun l'attribut `a`.

```
SELECT ... FROM R1 JOIN R2 ON c JOIN R3 ON c1 ... JOIN Rn ON cn ;
```

Le mot clé, **`JOIN ... ON ...`** fait partie de la clause **`FROM`**. On peut donc le faire suivre par les clauses **`WHERE`** et toutes celles qui peuvent s'ajouter ensuite.

Pour être sûr que la jointure soit possible, elle sera *systématiquement faite autour d'une clé étrangère*.

```
SELECT eleve.nom, prof.nom
       FROM eleve JOIN prof ON eleve.id_prod_principal = prof.id
       WHERE classe = "T12"
       ORDER BY eleve.nom
```

On notera qu'il est possible de joindre une classe avec elle-même.

```
SELECT eleve1.nom, eleve2.nom
       FROM eleve as eleve1 JOIN eleve as eleve2
            ON eleve1.prenom = eleve2.prenom
```

4.2 Requêtes Multiples

Le résultat d'une requête est une table. La table résultat peut donc être utilisée dans de nouvelles requêtes, pour une sélection ou une jointure.

De plus, si la table résultat est réduite à une colonne, elle peut être assimilée à une liste et peut ainsi être utilisé pour vérifier l'appartenance d'une valeur à la liste (**`in`**). Si la table est deplus réduite à une ligne, elle peut être assimilée à une valeur, et peut être utilisée en tant que telle dans une condition.

On appelle cela, *l'imbrication* de requête.

```
SELECT COUNT(*) / (SELECT COUNT(*) FROM eleve) FROM eleve WHERE moyenne >= 12 ;
SELECT nb_habitant FROM villes WHERE ville IN
    (SELECT ville FROM seisme GROUP BY ville HAVING MAX(magnitude) >= 7) ;
SELECT AVG(moyennes_infos) AS moyenne_inf_classe
       FROM (SELECT AVG(notes_info) AS moyennes_info
                    FROM eleves
                    GROUP BY numero_etudiant )
       GROUP BY classe ;
```

Il existe aussi des opérations ensemblistes effectives sur deux requêtes ayant des tables résultats dont le nombre de colonnes est égal.

- ▷ `requete1` **`UNION`** `requete2` : renvoie l'union des enregistrements.
- ▷ `requete1` **`INTERSECT`** `requete2` : renvoie l'intersection des enregistrements.

▷ `requete1` **`EXCEPT`** `requete2` : renvoie la différence des résultats

```
SELECT nom, prenom FROM baccalaureat WHERE moyenne >= 12
       INTERSECT
SELECT nom, prenom FROM eiffel ;
```

V Modèle Entité — Relation

Le modèle *entité-relation* est un modèle de données reposant sur deux concepts de base :

- ▷ les *entités* : des objets possédant un certain nombre de caractéristiques.
- ▷ les *associations* : des ensemble étiquetés d'au moins deux entités représentant la relation entre ces entités.

⚛ Exemples.

Un séisme est une entité caractérisée par sa magnitude, sa localisation géographique, sa profondeur, l'équipement qui l'a enregistré.

Une ville est une autre entité caractérisée par son nom, le pays dans lequel elle se situe, son nombre d'habitants, …

On peut créer une association « le séisme `S` a touché la ville `V` ».

❧ ❧ ❧

On retiendra trois types d'associations :

- ▷ Relation *Parent-Enfant* (1-∗) : l'entité *parente* peut-être associée à aucune ou plusieurs entités *enfants*. L'entité *enfant* est associée à exactement une entité *parente*. *e.g.* : un *élève* est dans une *classe*.
- ▷ Relation *symétrique* (1-1) : une entité est associée exactement à une autre entité. *e.g.* : un numéro d'étudiant correspond exactement à un numéro de sécurité sociale.
- ▷ Relation *multiple* (∗-∗) : les entités peuvent être associées à plusieurs autres entités et vice et versa. *e.g.* : les élèves participent aux cours de plusieurs professeurs et les professeurs font cours avec plusieurs élèves.

Les entités modélisent les tables dans la base de donnée. L'intérêt de ce modèle est *in fine* d'aider à la construction de ces tables et de savoir comment joindre deux tables (entités) en relation. Les associations entre deux tables `R1` et `R2` se traduisent alors comme ceci :

- ▷ 1-∗ : signifie qu'il faut une clé étrangère dans `R1` vers `R2`. La jointure se fait de `R1` sur `R2` autour de la clé étrangère.
- ▷ 1-1 : signifie que la jointure se fait autour des clés primaires.
- ▷ ∗-∗ : est difficilement représentable par des clés étrangères. On doit alors créer une table intermédiaire, `A`, représentant l'association et créer deux associations 1-∗ et $∗$-1 entre `R1` et `A` puis entre `A` et `R2`.

VI Memento SQL

6.1 Sélection

```
SELECT a FROM R ;
SELECT a, b, c FROM R ;
SELECT a as nadine, b FROM R ;
SELECT * FROM R ;
SELECT a, b, c FROM R WHERE condition ;
SELECT (a+b) / 2 as moy,
       c * 12.3 as c2,
       FLOOR(c) as fc,
       CONCAT(e, "␣", f, a) as s
       FROM R ;
```

Quelques fonctions :

- ▷ `concat` : concatène ses arguments
- ▷ `abs` : valeur absolue
- ▷ `floor`, `ceil` : partie entière, partie entière supérieure
- ▷ `sqrt`, `pow`, `log`, `log10`, `log2`, `sin`, `cos`, `tan`, `exp`, ...

6.2 Conditions

Opérateur	Déscription	Exemple
`=`	égalité	~b = « nadine »
≠, `<>`	non égalité	~b != « nadine »
`>`	supérieur strict	`b > 12`
`<`	inférieur strict	`b < 12`
⩾	supérieur ou égal	`b ⩾ 12`
⩽	inférieur ou égal	`b ⩽ 12`
`IN`	appartenance à une liste	`b IN ("nadine", "tajine")`
`BETWEEN`	intervalle	`b BETWEEN 12 AND 21`
`LIKE`	correspondance	`b LIKE "%ine"`
`IS NULL`	est nulle	`b IS NULL`
`IS NOT NULL`	n'est pas nulle	`b IS NOT NULL`

L'opérateur **`LIKE`** permet de chercher des chaînes de caractère à l'aide de caractère *joker* :

- `%` : qui signifie « n'importe quelle séquence de caractères, éventuellement aucun »
- `_` : qui signifie « exactement un caractère »

```
SELECT a FROM R WHERE c1 AND c2 ;
SELECT a FROM R WHERE c1 OR c2 ;
```

6.3 Agrégats

```
SELECT MAX(a) FROM R WHERE condition GROUP BY b ;
SELECT a % 10 as b FROM R GROUP BY b ;
```

```
SELECT SUM(a), b FROM R GROUP BY b HAVING condition ;
SELECT AVG(a), b FROM R WHERE c1 GROUP BY b HAVING c2 ;
```

Quelques fonctions d'agrégation :

- ▷ **AVG** : moyenne
- ▷ **MAX** : maximum
- ▷ **MIN** : minimum
- ▷ **SUM** : somme
- ▷ **COUNT(*)** : cardinal
- ▷ **COUNT(DISTINCT** x**)** : cardinal d'enregistrements avec x distincts
- ▷ STRING_AGG : concaténation
- ▷ VAR : variance

6.4 Tri

```
SELECT a FROM R WHERE condition ORDER BY b ;
SELECT a FROM R WHERE condition ORDER BY b DESC ;
SELECT MAX(a) as c FROM R GROUP BY b ORDER BY c ;
SELECT MAX(a) as c FROM R GROUP BY b ORDER BY c DESC ;
SELECT MAX(a) as c FROM R GROUP BY b ORDER BY AVG(c) ;
```

6.5 Troncature

```
SELECT a FROM R WHERE condition LIMIT n ;
SELECT a FROM R WHERE condition LIMIT n OFFSET m ;
SELECT a FROM R WHERE condition ORDER BY b LIMIT n OFFSET m ;
```

6.6 Jointure

```
SELECT R1.a, R2.b
       FROM R1 JOIN R2 ON R1.C = R2.d
       WHERE condition ;
SELECT R1.a, R2.b
       FROM R AS R1 JOIN R AS R2 ON R1.C = R2.d ;
```

6.7 Opération Ensemblistes

```
SELECT a,b FROM R1 UNION SELECT a,b FROM R2 ;
SELECT a,b FROM R1 INTERSECT SELECT a,b FROM R2 ;
SELECT a,b FROM R1 EXCEPT SELECT a,b FROM R2 ;
```