Module 5 - Les différentes opérations de l'algèbre relationnelle en pratique: opérateurs, jointures et fonctions de groupes

Les différentes opérations de l'algèbre relationnelle en pratique

Nous avons vu au module 3 les différentes opérations définies en algèbre relationnelle :

- **restriction**
- **projection**
- **union**
- **intersection**
- **différence**
- **multiplication**
- **division**
- **jointure**

Restriction (`WHERE`)

La *restriction* est une des opérations de l'algèbre relationnelle (souvenez-vous !)

```
SELECT listeColonnes FROM nomTable [WHERE condition]
```

où `condition` peut-être composée de colonnes,d' expressions, de constantes liées deux à deux entre des **opérateurs**:

- de comparaison: >, = , < , <=, >=, <>(not)
- logiques: NOT, AND, OR
- intégrés: BETWEEN, IN, LIKE, IS NULL

> L'ordre de priorité des opérateurs logiques est NOT, AND et OR.

Projection (`SELECT col1,col2`)

La projection consiste à retenir dans la relation résultante *certains* attributs (colonnes) et pas d'autres.

```
SELECT attribut1, attribut2 ... FROM Relation;
```

Intersection (`INTERSECT`) *ET*

Définition: qui extrait des données présentes *simultanément* dans les deux tables. S'écrit en composant deux requêtes de *projection* (**l'intersection ne peut se faire que sur des relations ayant le même nombre d'attributs et attributs de même type !**)

Prenons comme exemple les deux relations suivantes:

- AvionsdeAF: **immat**, typeAvion, nbHVol
- AvionsdeSG: **immatriculation**, typeAv, prixAchat

```sql
--
CREATE TABLE AvionsdeAF
(immat CHAR(6), typeAvion CHAR(10), nbHVol DECIMAL(10,2),
CONSTRAINT pk_AvionsdeAF PRIMARY KEY (immat));

CREATE TABLE AvionsdeSING
(immatriculation CHAR(6), typeAv CHAR(10), prixAchat DECIMAL(14,2),
CONSTRAINT pk_AvionsdeSING PRIMARY KEY (immatriculation));

INSERT INTO AvionsdeAF VALUES ('F-WTSS', 'Concorde', 6570);
INSERT INTO AvionsdeAF VALUES ('F-GLFS', 'A320', 3500);
INSERT INTO AvionsdeAF VALUES ('F-GTMP', 'A340', NULL);

INSERT INTO AvionsdeSING VALUES ('S-ANSI', 'A320', 104500);
INSERT INTO AvionsdeSING VALUES ('S-AVEZ', 'A320', 156000);
INSERT INTO AvionsdeSING VALUES ('S-MILE', 'A330', 198000);
INSERT INTO AvionsdeSING VALUES ('F-GTMP', 'A340', 204500);
```

```
-- Quels sont les types d'avions que les 2 compagnies exploitent en commun ?
SELECT typeAvion FROM AvionsdeAF INTERSECT SELECT typeAv FROM AvionsdeSING;
-- Quels sont les avions que les 2 compagnies exploitent toutes les deux ? (immatriculation de l'appareil)
SELECT immat FROM AvionsdeAF INTERSECT SELECT immatriculation FROM AvionsdeSING;
```

> Seules des colonnes de même type doivent être comparées avec des opérateurs ensemblistes

> Pour en apprendre plus, voir la clause INTERSECT de la documentation officielle.

Union (`UNION, UNION ALL`) *OU*

Définition: qui fusionnent des données des deux tables

Continuons avec la base précédente.

```
-- Quels sont les types d'avions que les deux compagnies exploitent ? (résultats distincts par défaut)
SELECT typeAvion FROM AvionsdeAF UNION SELECT typeAv FROM AvionsdeSING;
-- Même requête avec les duplicatas.
SELECT typeAvion FROM AvionsdeAF UNION ALL SELECT typeAv FROM AvionsdeSING;
```

> Pour en apprendre plus, voir la clause UNION de la documentation officielle.

Différence (`SELECT DISTINCT + NOT IN`)

Définition: qui extrait des données présentes dans une table sans être présentes dans la deuxième table.

Pas d'opérateur dédié en MySQL, il faut concaténer deux requêtes avec `SELECT DISTINCT` et `NOT IN`.

```
-- Quels sont les types d'avions exploités par la compagnie 'AF' mais pas par 'SING' ?
SELECT DISTINCT typeAvion FROM AvionsdeAF WHERE typeAvion NOT IN (SELECT typeAv FROM AvionsdeSING);
-- Quels sont les types d'avions exploités par la compagnie 'SING' mais pas par 'AF' ?
SELECT DISTINCT typeAv FROM AvionsdeSING WHERE typeAv NOT IN (SELECT typeAvion FROM AvionsdeAF);
```

Produit cartésien ou multiplication (`CROSS JOIN`)

Définition: produit l'ensemble des combinaisons entre les enregistrements (lignes) de deux tables.

Ajoutons une relation `Pilote` dans notre base de données

```
CREATE TABLE Pilote
        (brevet VARCHAR(6) PRIMARY KEY,
         nom VARCHAR(16), nbHVol DECIMAL(7,2), compa CHAR(4));

INSERT INTO Pilote VALUES ('PL-1', 'Gratien Viel', 450, 'AF');
INSERT INTO Pilote VALUES ('PL-2', 'Richard Grin', 1000, 'SING');
INSERT INTO Pilote VALUES ('PL-3', 'Placide Fresnais', 2450, 'CAST');
INSERT INTO Pilote VALUES ('PL-4', 'Daniel Vielle', 5000, 'AF');
```

```
-- Quels sont tous les couples possibles (avions, pilotes) entre les en considérant les avions et pilotes de la compagnie 'AF' ?
SELECT * FROM Pilote CROSS JOIN AvionsdeAF WHERE compa = 'AF';
```

> L'ordre de la requête: toujours la construction des relations d'abord (`SELECT`, `JOIN`, etc.) puis `WHERE` s'applique sur la relation finale.

Les jointures, le *cœur* du modèle relationnel

Les jointures **permettent d'extraire les données issues de plusieurs relations** (tables). Le processus de normalisation et d'atomisation des données a pour conséquence d'augmenter le nombre de tables. Ainsi, la majorité des requêtes adressées à une base de données relationnelle utilise des jointures (et c'est une bonne chose !)

> Une jointure met en relation deux tables sur la base d'un *prédicat*, ou *clause de jointure* (comparaison de colonnes). Généralement, cette comparaison fait intervenir une clef étrangère d'une table avec une clef primaire d'une autre table (car le modèle relationnel est fondamentalement basé sur les valeurs). Mais pas toujours ! Voir le cas des *inéquijointures*.

Opération de jointure (`INNER JOIN`, `OUTER JOIN`)

L'opération de jointure est au *cœur* du modèle relationnel. C'est une des opérations de l'algèbre relationnelle.

> Pour mettre trois tables T1, T2 et T3 en jointure, il faut utiliser deux clauses de jointures (une entre T1 et T2 et l'autre entre T2 et T3). Pour n tables, il faut n-1 clauses de jointures. Si vous oubliez une clause de jointure, un produit cartésien restreint est généré.

Comprendre la jointure

Pour bien comprendre l'opération de jointure, il est utile de penser à l'opération de produit cartésien (`CROSS JOIN`) entre deux tables `t1` et `t2` : cette opération *associe à une ligne de* `t1` *chaque ligne de* `t2` (cf exemple reconstruction d'un jeu de cartes). **C'est une jointure sans prédicat** (sans condition).

La jointure (`INNER JOIN` ou `JOIN`) fonctionne de la même manière sauf que **l'association entre une ligne de t1 avec une ligne de t2 ne se fait que si le prédicat est évalué à vrai**, via la clause `ON`.

Une jointure ne peut se faire que sur des colonnes identiques, de même **type** !

Résultat d'une jointure

Comme toute opération d'algèbre relationnelle, **la jointure est une opération de fermeture**, elle renvoie donc **une nouvelle relation**, une nouvelle table.

Sans projection, cette relation est constituée **de tous les attributs (colonnes) de chaque table**.

Exemple de jointure

Reprenons notre association *un-à plusieurs* `Client--*Facture`

```
-- Liste de toutes les factures et du client associé
SELECT * FROM Facture
JOIN Client
ON Facture.id_client = Client.id_client;
-- Résultat
+------------+-----------+-------------+---------------+--------------+------+-----------+----------------+------------+
| id_facture | id_client | num_facture | date_emission | montant_euro | paye | id_client | prenom_client | nom_client |
+------------+-----------+-------------+---------------+--------------+------+-----------+----------------+------------+
|          1 |         1 | F-900-08    | 2022-12-12    |       120.50 |    1 |         1 | John           | Doe        |
|          2 |         1 |  F-500-02   | 2023-01-13    |        90.00 |    1 |         1 | John           | Doe        |
-- etc..
```

On retrouve une table composée des colonnes de `Facture` et de `Client`.

> On remarque qu'on a deux colonnes avec le même nom, ce qui est interdit en algèbre relationnelle. En fait, chaque colonne est dans l'espace de nom de sa table. Par exemple, la première colonne `id_client` est en fait `Facture.id_client` (le `.` est *un opérateur de portée*), la deuxième est `Client.id_client`.

Les différents types de jointure

Il existe différents types de jointure:

- **jointure interne** (`INNER JOIN`): plus courante, symétrique
  - *equijointure*: utilise un prédicat d'égalité
  - *inéquijointure*: utilise un prédicat de non-égalité
  - *autojointure*: cas particulier de l'équijointure. Jointure d'une table avec elle-même
- **jointure externe** (`OUTER JOIN`): plus compliquée, asymétrique: une table est dominante
  - *unilatérale*
  - *bilatérale*

Jointure interne

Jointure interne: *équijointure* (`JOIN` ou `INNER JOIN`)

Une équijointure **utilise l'opérateur d'égalité dans la clause de jointure** (clause `ON`) et compare généralement des clefs primaires avec des clefs étrangères.

```
-- Cf Exercicse du TD Module 5
SELECT * FROM
Pilote
JOIN Compagnie
ON Pilote.compa = Compagnie.comp;
```

Sucre syntaxique: `USING`

La requête précédente peut s'écrire également avec le mot-clef `USING`.

La clause `USING` permet de déclarer la liste des noms de colonnes qui doivent exister dans chaque table *pour réaliser la jointure*. Si les tables `t1` et `t2` contiennent toutes deux la colonne `c1`, la requête réalisera une équijointure sur cette colonne.

```
-- équivalent à la requête précédente: la colonne id_client existe dans les deux tables
-- la jointure est réalisée sur la colonne id_client
SELECT * FROM Facture JOIN Client USING (id_client);
```

Sucre syntaxique: `NATURAL JOIN`

MySQL propose aussi la clause `NATURAL JOIN`. Elle est équivalente à INNER JOIN avec une clause USING qui liste toutes les colonnes de même nom dans chaque table. NATURAL JOIN fera automatiquement la jointure sur les colonnes.

```
-- équivalent à la requête précédente
SELECT * FROM Facture NATURAL JOIN Client;
```

> En pratique, on préférera **ne pas utiliser cet opérateur** qui n'est pas assez *explicite* (demande de connaître la structure des tables pour comprendre la requête de jointure).

Jointure interne: *inéquijointure*

Une équijointure **utilise l'opérateur d'inégalité dans la clause de jointure** (<, >, <=, >=, BETWEEN, LIKE, IN).

À l'inverse des équijointures, la clause n'est pas basée sur l'égalité des clefs primaires et clefs étrangères.

Jointure interne: *Autojointure*

Cas particulier de l'équijointure, l'autojointure réalise une jointure d'une table avec elle-même.

Jointure externe: `OUTER JOIN`

Les jointures externes permettent d'extraire des lignes qui ne répondent pas au critère de jointure.

Lorsque deux tables sont en jointure externe, une table est *dominante*, l'autre est *subordonnée*. Ce sont **les lignes de la table dominante qui seront retournées même si elles ne respectent pas la condition de jointure (prédicat)**.

> Comme les jointures internes, les jointures externes sont souvent utilisées sur les clefs primaires/clefs étrangères.

Il existe deux types de jointures externes:

- jointure *unilatérale*: (dominant>subordonné) `{LEFT|RIGHT} OUTER JOIN`.
- jointure *bilatérale*: (pas de dominant) `FULL OUTER JOIN`*. Permet d'extraire des lignes qui ne répondent pas au critère de jointure des deux côtés de la clause de jointure.

> *Pas implémenté en MySQL ! À réaliser avec UNION. Pas abordé ici.

Jointure externe unilatérale: `{LEFT|RIGHT} OUTER JOIN`

- `LEFT` : retourne toutes les lignes de la table à gauche (de `JOIN`), **même si le prédicat n'est pas vérifié**
- `RIGHT` : idem, mais à droite

```
-- ici Compagnie est la table dominante (à gauche du JOIN)
SELECT * FROM Compagnie
LEFT OUTER JOIN
Pilote
ON compa=comp;
```

Dans cette requête, on extrait toutes les compagnies et leurs pilotes, *même les compagnies qui n'ont pas de pilotes* (prédicat non vérifié).

Fonctions

Le SGBDR met à disposition un grand nombre de fonctions *built-in* (prêtes à l'emploi) pour manipuler les données (interroger, transformer).

Fonctions numériques

ABS
ACOS
ATAN
CEIL
COS
DEGREES
EXP
FLOOR
LOG
POW
RADIANS
SQRT
ROUND
SIN
etc.

Fonctions de dates

ADDDATE

ADDTIME

CURDATE, CURRENT_DATE

CURTIME, CURRENT_TIME

DATE

DATEDIFF(date1,date2)

DATE_ADD(date, INTERVAL exp type)

DATE_FORMAT(date, format)

DAY

YEAR

MONTH

LASTDAY

MINUTE

MICROSECOND

WEEKOFYEAR

etc.

Fonctions de groupe

`AVG` : moyenne

`COUNT` : compter le nombre d'enregistrements dans une table

MAX

MIN

`GROUP_CONCAT` : retourne une chaîne de caractère concaténée

`STDDEV` : écart-type

SUM: somme

`VARIANCE` : variance

Fonctions de groupes

Les fonctions de groupes s'appliquent *à la totalité ou à une seule partie* d'une table.

- `AVG` : Moyenne
- `STDEV` : Écart-type (racine de la variance)
- `VARIANCE` : Variance
- `COUNT` : Retourne le nombre d'enregistrements
- `MAX,MIN` : valeurs max et min
- `GROUP_CONCAT` : Composition d'un ensemble de valeurs.

> À l'exception de `COUNT`, toutes les fonctions ignorent les valeurs `NULL` (il faudra utiliser `IFNULL` pour contrer cet effet).

Fonctions de groupes

Voyons ces fonctions de groupe en pratique. Prenons cet exemple avec une relation `Pilote`.

```
CREATE TABLE Pilote(
brevet VARCHAR(6),
nom VARCHAR(16),
nbHVol DECIMAL(7,2),
prime INT(4),
embauche DATE,
typeAvion CHAR(4),
compa VARCHAR(4)
CONSTRAINT pk_pilote PRIMARY KEY (brevet)
);

INSERT INTO Pilote VALUES ('PL-1', 'Gratien Viel', 450, 500, '1965-02-05','A320', 'AF');
INSERT INTO Pilote VALUES ('PL-2', 'Didier Donsez',  0, NULL, '1995-05-13', 'A320',    'AF');
INSERT INTO Pilote VALUES ('PL-3', 'Richard Grin',  1000, NULL, '2001-09-11', 'A320', 'SING');
INSERT INTO Pilote VALUES ('PL-4', 'Placide Fresnais',2450, 500,'2001-09-21', 'A330', 'SING');
INSERT INTO Pilote VALUES ('PL-5', 'Daniel Vielle',    400, 600, '1995-01-16', 'A340', 'AF');
INSERT INTO Pilote VALUES ('PL-6', 'Francoise Tort', NULL, 0, '2000-12-24', 'A340', 'CAST');

Table Pilote;
```

Exemples de fonctions de groupes

```sql
-- Moyenne des heures de vol et des primes des pilotes de la compagnie 'AF'.
SELECT AVG(nbHVol), AVG(prime) FROM Pilote WHERE compa = 'AF';
-- Nombre de pilotes, d'heures de vol et de primes distinctes recensées dans la table.
SELECT COUNT(*), COUNT(nbHVol) , COUNT(prime) FROM Pilote;
-- Nom des pilotes de la compagnie 'AF'.
SELECT compa, GROUP_CONCAT(nom) FROM Pilote GROUP BY compa;
-- Nombre d'heures de vol le plus élevé, date d'embauche la plus récente. Nombre d'heures de vol le moins élevé, date d'embauche la plus ancienne.
SELECT MAX(nbHVol), MAX(embauche) "Date+", MIN(prime) , MIN(embauche) "Date-" FROM Pilote;
-- Écart type des primes, somme des heures de vol, variance des primes des pilotes de la compagnie 'AF'.
SELECT STDDEV(prime) , SUM(nbHVol) , VARIANCE(prime) FROM Pilote WHERE compa = 'AF';
```

Opération d'ordonnancement et de regroupements: `ORDER BY`, `GROUP BY` et `HAVING`

`ORDER BY {col_name|exp|position} [ASC|DESC]`

Dans une table, **il n'existe aucune notion d'ordre des lignes !**

Utilisable qu'une fois en fin de requête, la clause `ORDER BY` permet de *trier* les résultats par un attribut (colonne), dans l'ordre ascendant (du plus petit au plus grand, alphabétique, etc.) ou descendant

```
-- Retourner les résultats par ordre alphabétique
SELECT * FROM Pilote CROSS JOIN AvionsdeAF WHERE compa = 'AF' ORDER BY nom ASC;
```

> Pour en apprendre plus, voir la clause SELECT de la documentation officielle.

Opération d'ordonnancement et de regroupements: `ORDER BY`, `GROUP BY` et `HAVING`

Regroupement: `GROUP BY` et `HAVING`

Le groupement de lignes dans une requête se programme avec la clause `GROUP BY`.

```
SELECT col1[, col2...], fonction1Groupe(...)[,fonction2Groupe(...)...]
FROM nomTable [ WHERE condition ]
GROUP BY {col1 | expr1 | position1} [,{col2... }]
[ HAVING condition ]
[ORDER BY {col1 | expr1 | position1} [ASC | DESC] [,{col1 ... }] ];
```

La clause `WHERE` de la requête permet d'exclure des lignes pour chaque groupement, ou de rejeter des groupements entiers. Elle s'applique donc à la totalité de la table.

La clause `GROUP BY` liste les colonnes du groupement. La clause `HAVING` permet de poser des conditions sur chaque groupement.

> HAVING est comme le WHERE d'un regroupement (restriction)

La clause `ORDER BY` permet de trier le résultat.

Les colonnes présentes dans le `SELECT` doivent apparaître dans le `GROUP BY`. **Seules des fonctions ou expressions peuvent exister en plus** dans le `SELECT`.

Utilisées avec `GROUP BY`, **les fonctions de groupes s'appliquent désormais à chaque regroupement**

Voyons ça en pratique:

```
-- Moyenne des heures de vol et des primes pour chaque compagnie.
SELECT compa, AVG(nbHVol), AVG(prime) FROM Pilote GROUP BY(compa) ;
-- Nombre de pilotes (et ceux qui ont de l'expérience du vol) par compagnie.
SELECT compa, COUNT(*), COUNT(nbHVol) FROM Pilote GROUP BY(compa) ;
-- Nombre d'heures de vol le plus élevé, date d'embauche la plus récente pour chaque compagnie.
SELECT compa, MAX(nbHVol), MAX(embauche) "Date+" FROM Pilote GROUP BY(compa) ;
-- Nombre de pilotes qualifiés par type d'appareil et par compagnie.
SELECT compa, COUNT(brevet) FROM Pilote
GROUP BY(compa)
HAVING COUNT(brevet)>=2 ;
```

Utiliser les index

> Les index sont très importants !

Les SGBDR créée systématiquement un index chaque fois que l'on pose une clef primaire (`PRIMARY KEY`) ou une contrainte d'unicité (`UNIQUE`) sur une table.

En revanche, il n'y a pas d'index créé automatiquement par le SGBDR derrière une `FOREIGN KEY` (clef étrangère). À vous de le faire si vous réalisez des jointures sur cette colonne.

Un mot sur les jointures et les performances

Les SGBDR sont optimisés pour faire des jointures.

- faites des jointures !
- **créer des index là où c'est nécessaire** (en fonction de la logique métier), notamment sur les colonnes de clef étrangères impliquées dans une jointure
- laissez le SGBD faire les optimisations pour vous. **Concentrez-vous sur vos relations, vos index et vos requêtes bien formulées !**

Références utiles

- Subqueries (documentation officielle), l'usage des sous-requêtes dans MySQL
- Common Table Expressions (CTE), permettent d'aliaser des requêtes pour les utiliser comme sous requête. S'écrit avec la clause WITH. **Les CTE sont à privilégier par rapport aux sous-requêtes**: plus lisibles, peuvent être réutilisées.