Poly de Cours 1.

SQL

Objectifs du Chapitre: savoir utiliser un SGBD:

- ✎ écrire des requêtes SQL.
- ✎ manipuler (insérer, modifier, effacer) des données
- ✎ créer les tables, avec des contraintes d'intégrité

Prérequis: ce document fait quelques références au modèle relationnel et surtout à l'algèbre relationnelle.
Notation: ⊘ les remarques entre ces deux symboles ne sont pas à apprendre ⊘, ni les remarques marquées d'un bandeau étoilé.

Contents

1 Le langage SQL: introduction

La plupart des exemples utilisés dans ce polycopié utilisent une base de donnée de films dont voici un extrait:

Cine

NOM_CINE	ADRESSE	TELEPHONE
ugc bercy	3 rue...	06043494
...		

Prog

NOM_CINE	TITRE	SALLE	HORAIRE
ugc bercy	Le discours d'un roi	1	20h00
...			

Film

TITRE	MeS	Acteur
Le discours...	T.Hooper	C.Firth
...		

1.1 Présentation du langage SQL

SQL (Structured Query Language) est le langage de programmation le plus utilisé pour interagir avec une base de donnée relationnelle.

⊘ Ce langage a été développé à l'origine par Donald D. Chamberlin et Raymond F. Boyce pour IBM à la fin des années 1970 (projets SEQUEL, System-R), suite au développement du modèle relationnel par Edgar F. Codd. Originellement appelé SEQUEL (pour Structured English QUEry Language, et parce que c'était la suite du langage QUEL) le nom du langage fut changé pour SQL dès 1975 le nom SEQUEL étant déjà déposé par une compagnie anglaise d'aviation. Certains utilisateurs anglophones continuent néanmoins à le prononcer ['si:kwəl] au lieu de [ˌɛsˌkju:ˌɛl]. Le langage s'est rapidement imposé dans les systèmes relationnels au point de devenir le standard de facto dans les SGBD relationnels. Le langage a été normalisé par l'ANSI en 1986 puis standardisé par l'ISO en 1987. Le langage a depuis été enrichi à travers une dizaine de standards successifs. Le standard SQL92 (SQL2) contient déjà l'essentiel des éléments discutés dans ce cours, mais a été enrichi par SQL:99, SQL:2003,... SQL:2019.⊘

Le standard SQL décrit entre autres les points suivants (sur plusieurs dizaines de chapitres et plus d'un millier de pages):

- créer, modifier, supprimer le schéma d'une base de donnée. On parle de *Langage de Définition de Données (DDL)*

  Le standard décrit en particulier les types de données pris en charge (entiers, chaînes de caractères), même si en pratique il reste des variations entre SGBD qui peuvent compliquer la portabilité des applications.

- extraire, insérer, modifier, supprimer des informations (plus précisément des lignes) dans les tables de la base de donnée. On parle de *Langage de Manipulation de Données (DML)*

- définir les droits d'accès des utilisateurs

- gérer les transactions

- gérer les métadonnées (informations sur le schéma)

- gérer les connections, contrôler comment un client/programme peut se connecter au serveur...

La figure 1 illustre une requête SQL simple et le résultat qu'elle renvoie lorsque l'on exécute la requête sur la table `Client` illustrée. Si l'on avait souhaité afficher toutes les colonnes de la table client, on aurait pu écrire :

```
SELECT *
FROM Client;
```

⚠ La plupart des SGBD imposent de terminer une instruction SQL par un point virgule.

Requête:

```
SELECT Nom, Prenom
FROM Client;
```

Table Client

NSS	NOM	PRENOM	ADRESSE
1111	GROZ	Benoit	75016
3333	BIDOIT	Nicole	75014
2222	COHEN	Sarah	75008

Résultat

NOM	PRENOM
GROZ	Benoit
BIDOIT	Nicole
COHEN	Sarah

Figure 1: Une base de données, constituée de 2 relations: Client et Achat.

1.2 Atouts et limitations du langage SQL

Les principaux atouts du langage SQL sont:

- ✔ son statut de standard quasi incontournable (popularité, portabilité) pour interroger des données relationnelles
- ✔ sa simplicité d'utilisation

Un des principaux avantages de SQL est sa standardisation. C'est en grande partie cette standardisation qui lui a permis de devenir la lingua franca dans le domaine des bases de données.

La facilité d'utilisation est au coeur du langage SQL, comme du modèle relationnel et donc des SGBD relationnels. SQL est en particulier un langage *déclaratif*; on y décrit le résultat souhaité en laissant libre le SGBD de calculer la requête de la façon la plus efficace. On oppose ainsi un langage déclaratif comme SQL à un langage *procédural* comme l'algèbre relationnelle qui, elle, décrit le calcul à effectuer par une séquence d'opérations. De ce point de vue, SQL décrit les requêtes d'une façon similaire à une formule logique.

On peut rapprocher le concept de langage déclaratif du concept de langage de haut niveau: on souhaite faciliter la vie de l'utilisateur en lui présentant une interface qui abstrait (cache) les détails d'implémentation à l'utilisateur. On retrouve d'ailleurs aussi cette notion d'abstraction dans l'architecture à 3 niveaux ANSI SPARC.

Les principales limitations du langage SQL sont:

- ✘ style de programmation désuet (langage des années 80)
- ✘ plutôt limité à des traitements simples (le langage n'est pas conçu pour implémenter des algorithmes ou transformation de données complexes)
- ✘ langage limité au modèle relationnel, donc aux données structurées.
- ✘ la plupart des langages de programmation, contrairement à SQL, ne proposent pas nativement des opérations sur des tables vues comme ensembles de lignes. Pour manipuler des données d'un SGBD depuis un langage de programmation orienté objet, on est donc amené à convertir les données relationnelles en objets (ORM) et vice-versa, et à parcourir les lignes une à une (curseurs), ce qui peut être fastidieux. On parle parfois de l'inadéquation d'impédance (🇬🇧 impedance mismatch) relationnelle-objet.
- ✘ sémantique parfois contre-intuitive sur les points de détails (en particulier en présence de NULL).

Par ailleurs, la standardisation est imparfaite: les types de données diffèrent légèrement d'un SGBD à un autre, ce qui complique la portabilité. L'omniprésence de SQL et des SGBD relationnels à partir de la fin des années 80 a longtemps été un frein à l'émergence d'approches radicalement nouvelles: SQL a absorbé au fur et à mesure des extensions successives des outils pour manipuler les données "objet", les données semi-structurées (XML, JSON) etc. Néanmoins, à partir du milieu des années 2000 de nombreux SGBD NoSQL à l'architecture radicalement différente sont apparus qui n'utilisaient pas SQL pour traiter les données. Une partie de ces SGBD (mais pas tous) se sont par la suite rapprochés des SGBD relationnels en terme de langage de requête, de stockage de données ou de fonctionnalités. Enfin, pour ce qui est d'analyser des données, les librairies de python (et tout particulièrement la librairie pandas) permettent d'interroger des données structurées aussi simplement qu'en SQL, tout en étant intégrées à un véritable langage de programmation permettant des manipulations (transformations, calculs) plus complexes. Il peut donc être plus facile et efficace d'analyser des données sous pandas que dans SQL, surtout lorsque l'on manipule un tout petit nombre de relations plutôt qu'une base de donnée complète.

1.3 ⊘Lien entre SQL et la logique⊘

S'il est vrai que SQL décrit les requêtes d'une façon déclarative similaire à une formule logique, la similitude n'est peut être pas apparente pour le néophyte. Il faut en effet faire abstraction de la syntaxe de SQL, qui se veut proche de la description en langue naturelle (anglais). Mais la transformation est quasi immédiate pour les théoriciens, qui convertissent souvent le coeur des requêtes SQL (selection projection jointure) en requêtes de la logique du premier ordre. Cette conversion de requêtes SQL en formules logiques permet de prouver des propriétés en utilisant les résultats connus en Logique, voire d'évaluer des requêtes en utilisant le langage Datalog. Pour rendre apparent le lien avec la logique, considérons la requête suivante :

```
SELECT adresse
FROM Cine C, Prog P
WHERE C.nom_cine = P.nom_cine and P.Horaire = '20h00'
```

Cette requête peut se décrire comme suit: *"sélectionner toutes les adresses a telles qu'il existe des valeurs n,tel,ti,s,h pour lesquelles il y a une ligne (n,a,tel) dans Cine et (n,ti,s,h) dans Prog avec h=20h00."* En Datalog, ceci s'écrit:

```
Résultat(a) :- Cine(n,a,tel), Prog(n,ti,s,h), h=20h00
```

On devine ici une formule logique: la variable *a* est "libre" (non quantifiée), et on a une quantification existentielle sur les variables *n,tel,ti,s,h*. La variable *n* sert à faire la jointure, et la variable *h* à filtrer selon l'horaire.

Pour aller plus loin, on distingue deux formalismes logiques couramment utilisés pour représenter les requêtes relationnelles: le *Domain Relational Calculus* et le *Tuple Relational Calculus*.

1.4 Quelques propriétés de SQL

Un langage conçu pour manipuler des tables: SQL opère essentiellement sur des tables, qu'il s'agisse de créer une table, joindre deux tables, sélectionner certaines lignes d'une table. Les requêtes SQL opèrent ainsi sur des tables vues comme des ensembles de lignes, et ont pour résultat une table (de la même façon que les expressions d'algèbre relationnelle prennent en entrée des relations et ont pour résultat une relation).

Les chaînes de caractères: Les chaînes de caractères sont délimitées par des guillemets simples en SQL. Ceci sera expliqué plus en détail en section 3.10.1.

La casse en SQL:

Les mots-clefs (`SELECT, FROM...`) sont insensibles à la casse. Il est souvent d'usage de les écrire en majuscule pour distinguer du reste de la requête, mais ce n'est pas une nécessité.

Les noms de tables et de colonnes sont généralement insensibles à la casse, mais peuvent être sensibles sous certains systèmes. C'est une option paramétrable, la valeur défaut dépend du SGBD/de l'OS...

Attention, les chaines de caractères utilisées comme données sont sensibles à la casse : quand on compare la valeur de deux chaînes de caractère (typiquement dans les conditions WHERE ou HAVING d'une requête) la casse est prise en compte: 'aa≠ 'Aa' Ces chaînes de caractères apparaissent typiquement dans les conditions `WHERE, HAVING`. Un nom de table ou de colonne ne s'écrit jamais entre guillemets.[1]

Guide pour adopter un bon style de programmation

Comme pour tout langage, il est nécessaire d'adopter un style cohérent. Dans tout fichier/projet/etc., adoptez une convention et respectez là: par exemple mots clef majuscule, nom tables et colonnes minuscule. Ou tout minuscule. Le choix d'un style cohérent ne se limite pas à la casse: pensez à indenter le code, etc.
Quelques guides de style:
https://about.gitlab.com/handbook/business-ops/data-team/platform/sql-style-guide/
https://docs.telemetry.mozilla.org/concepts/sql_style.html
Il existe des linter pour SQL, avec des résultats plus ou moins discutables. Les requêtes SQL sont généralement courtes, donc utiliser un linter ne se justifiera en tout cas pas dans le cadre de nos cours.

1.5 Différences entre le modèle relationnel et SQL

Pour conclure, on rappelle ici les principales différences entre le modèle relationnel et la manipulation de tables en SQL. Le modèle relationnel étant un modèle purement théorique, les SGBDs utilisent en pratique souvent un vocabulaire légèrement différent, correspondant aux concepts de SQL, cf Figure 2. Parmi les principales différences entre le modèle relationnel et SQL:

[1] sauf cas particulier où l'on utiliserait ce nom comme une donnée. Par exemple, la table `pg_tables` sous postgresql contient le nom de toutes les tables de la base, donc on peut écrire une requête: `select * from pg_tables where tablename = 'film';`

Modèle Relationnel	SQL	
Relation	= Table	client
Attribut	= Colonne	adresse
Domaine	= Type	string
Clefs	= Clefs	nss : clef primaire de Client

Figure 2: Correspondances entre les concepts du modèle relationnel et SQL

- SQL autorise une table à contenir des doublons: c'est rare, car les tables contiennent souvent un identifiant unique, mais cela arrive (et c'est assez fréquent dans les résultats de requêtes).
- Il n'y a pas fondamentalement d'ordre entre les lignes d'une table en SQL, par contre SQL permet de spécifier un ordre dans lequel on souhaite afficher le contenu d'une table.
- En SQL comme dans le modèle relationnel, les relations d'une base de données ont des noms distincts, ainsi que les attributs d'une même relation, mais il est possible dans une requête de renvoyer une table dont plusieurs colonnes porteront le même nom.
- Le concept de clé a été formalisé dans le cadre du modèle relationnel avant d'être implémenté en SQL. Mais SQL permet de définir d'autres contraintes d'intégrité (en particulier pour gérer les NULLs).
- Il n'y a pas de NULL dans le modèle relationnel, et le langage SQL est plus expressif que l'algèbre relationnelle.

2 SQL comme Langage de Définition de Données

L'instruction `CREATE TABLE` permet de créer une table en SQL. On définit lors de la création de table le type des données contenues dans chaque colonne de la table. Une fois les tables créées, on peut insérer des données par la commande `INSERT`.

```
CREATE TABLE Vente (id INT, annee INT, produit VARCHAR(20));
INSERT INTO Vente VALUES(1,2020,'coquillettes');
SELECT * FROM Vente;
```

```
 id | annee |   produit
----+-------+-------------
  1 |  2020 | coquillettes
```

2.1 Les types de données en SQL

SQL définit de nombreux types de données, que l'on retrouve avec quelques variations dans tous les principaux BDs. En particulier:

pour les (chaines de) caractères	- chaînes de caractères de taille fixe, complétées à droite par des espaces: `CHAR(taille)` - chaînes de taille variable: `VARCHAR(taille_max)`
pour les données numériques	- entiers sur 32 bits: `INTEGER` (mais aussi `SMALLINT/BIGINT`...) - nombres en précision fixe (nb chiffres dont nbDecimales après virgule): `NUMERIC(nb, nbDecimales)` - nombres en virgule flottante: `REAL/DOUBLE PRECISION`.
pour les dates et mesures temporelles	- types date/heure: `DATE/TIME/TIMESTAMP`
pour les données binaires	- les Binary Large Objects `BLOB`, qui peuvent typiquement servir à stocker un fichier d'image ou de son.
autres	- ARRAY, ROW, JSON, XML, types définis par l'utilisateur

Le type `TIME` contient au minimum heure + min + s, alors que type `TIMESTAMP` (en français; "estampille") contient date+ time+ fractions de secondes sur 6 chiffres minimum.

Dans la plupart des applications des bases de données (en particulier les applications comptables), on privilégie le type `NUMERIC` aux flottants, pour éviter les erreurs d'arrondi.

Définir des types de données L'utilisateur peut définir ses propres types de données. Entre autres, il est possible de restreindre le domaine d'un type ou d'énumérer explicitement les valeurs possibles avec les instructions suivantes :

```
CREATE DOMAIN centaines AS INT CHECK(VALUE >99 AND VALUE <1000);

CREATE TYPE bug_status AS ENUM ('new', 'open', 'closed');
```

La valeur NULL Quel que soit le type de donnée d'un attribut, SQL permet d'y insérer une valeur spéciale, la valeur NULL. Cette valeur sert à indiquer qu'aucune valeur n'a été renseignée dans la ligne pour l'attribut concerné. Soit que l'attribut est inapplicable pour l'élément correspondant (on peut penser à la date de fin pour un projet non achevé), soit que l'on ne connaisse pas la valeur de l'attribut par manque d'information. La présence de valeurs nulles peut surprendre ou compliquer l'écriture de certaines requêtes, comme nous l'expliquerons plus en détail plus tard.

2.2 Créer et modifier des relations (tables) en SQL, contraintes d'intégrité

L'instruction ci-dessous permet de créer une table, initialement vide. Il faut pour cela définir le schéma de la table: nom et type des différents attributs (colonnes)valeurs. Pour insérer des données (ligne = tuple) dans la table, on utilise des instructions `INSERT`. Il est aussi possible de créer une table à partir du résultat d'une requête:

```
CREATE TABLE Film (
Titre CHAR(20),
MeS CHAR(20),
Acteur VARCHAR(20));
```

```
INSERT INTO Film VALUES('Rear Window','Hitchcock','Stewart');
INSERT INTO Film VALUES('Rear Window','Hitchcock','Kelly');
INSERT INTO Film VALUES('Rear Window','Hitchcock',NULL);
```

```
CREATE TABLE Film2 AS
  SELECT * FROM Film;
```

Il est possible de modifier le schéma d'une table après sa création, comme nous le verrons plus tard, mais ces modifications de schémas sont plutôt rare. On définit donc généralement les contraintes d'intégrité en même temps que l'on crée les tables. SQL permet de définir les contraintes d'intégrité suivantes pour maintenir la cohérence des donnée:

- NOT NULL
- valeurs uniques
- clefs primaires
- clefs étrangères
- condition de valeurs (check)

http://www.postgresql.org/docs/9.0/static/ddl-constraints.html

2.2.1 Contrainte NOT NULL

La contrainte NOT NULL sert à interdire la présence de valeurs NULL (que ce soit par insertion ou modification) pour un attribut de la table, comme illustré en Figure 3. Toute opération conduisant à l'apparition d'une valeur nulle dans cette colonne sera rejetée. La contrainte NOT NULL est souvent utilisée dans les bases de données, pour s'assurer que les données entrées sont complètes. Mais on ne dispose pas toujours de l'information pour certains attributs ce qui oblige parfois à autoriser la présence de NULL.

2.2.2 Contrainte d'unicité `UNIQUE`

Une contrainte d'unicité interdit à deux lignes d'avoir les mêmes valeurs sur les attributs. Avec la contrainte en Figure 4 (gauche), deux lignes peuvent par exemple avoir la même valeur sur `Nom_Cine`, mais elles devront alors être différentes soit sur titre soit sur horaire, et vice versa. La contrainte indique donc que "`Nom_Cine,Titre, Horaire`" est une clé de la relation Prog.

 ⊘ Toute ligne comportant un null sur au moins un de ces champs est ignorée par la contrainte (sauf chez Oracle).[2] ⊘

Il est bon de savoir que les contraintes d'unicités sont vérifiées par le SGBD en maintenant un index sur la valeur. Ajouter une contrainte d'unicité peut donc ralentir les opérations de mise à jour (qui doivent désormais mettre à jour l'index) mais peuvent accélérer certaines requêtes (qui peuvent exploiter l'index pour accéder plus rapidement aux données pertinentes).

[2] On peut expliquer ce comportement en interprétant chaque null comme une valeur inconnue: comme la valeur null est inconnue, il est possible qu'elle satisfasse la contrainte et on ne refusera que les opérations dont on est sûr qu'elles violent la contrainte. Une valeur null ne sera donc jamais considérées comme identique à une autre valeur, même si l'autre valeur est null aussi. Pour plus de détails, voir http://troels.arvin.dk/db/rdbms/#constraints-unique

```
CREATE TABLE Film (
Titre CHAR(20) NOT NULL,
Acteur VARCHAR(20),
MeS CHAR(20) NOT NULL DEFAULT 'Kurosawa',
)
```

Film

TITRE	ACTEUR	MeS
Les petits mouchoirs	Cotillard	Canet
La plage	Canet	
Mon Idole	Canet	Canet
Bienvenus chez les ch'tis	Boon	Boon

↪ Instance impossible car la contrainte sur MeS n'est pas respectée.

Figure 3: Une instance impossible en présence de contrainte not null

```
CREATE TABLE Prog (
Nom_Cine CHAR(20),
Titre CHAR(20),
Salle INT,
Horaire TIME,
UNIQUE (Nom_Cine,Titre, Horaire))
```

```
CREATE TABLE Cine (
Nom_Cine CHAR(20),
Adresse  VARCHAR(60),
Telephone CHAR(8),
PRIMARY KEY(Nom_Cine, Adresse))
```

Figure 4: Les contraintes de clés

2.2.3 Contrainte de clé primaire PRIMARY KEY

On ne peut définir qu'une clé primaire par table, même si cette clé peut comporter plusieurs attributs. En définissant la clé primaire, on définit le "moyen d'accès" privilégié à la table[3]. Une clef primaire vérifie que les attributs qui la composent sont tous non null, et que deux lignes ne sont jamais identiques sur l'ensemble de la clé. La contrainte en Figure 4 (droite) impose donc que `(Nom_Cine, Adresse)` soit UNIQUE, et que les deux attributs soient NOT NULL.

2.2.4 Déclaration des contraintes en ligne ou hors ligne

Comme illustré en Figure 5, une contrainte de clef sur une colonne peut être déclarée en ligne, ou hors de ligne. Déclarer la contrainte hors de ligne est nécessaire lorsque la contrainte met en jeu plusieurs attributs, et cela permet de choisir le nom qui lui est attribué (à défaut de quoi Postgresql baptisera la contrainte avec un nom par défaut, mais pour une instruction ALTER TABLE le choix d'un nom de contrainte sera obligatoire). Choisir le nom de la contrainte peut parfois aider à repérer plus facilement dans le message d'erreur quelle contrainte n'est pas satisfaite, et peut faciliter les modifications de la contrainte. Les contraintes d'unicité ou de domaine peuvent de manière similaire être déclarées en ligne ou pas.

2.2.5 Contraintes de domaine

On peut aussi définir des contraintes restreignant le domaine d'un attribut. Ce type de contrainte évalue une condition et retourne une erreur si le résultat est `Faux`. La condition peut porter sur plusieurs colonnes de la table, faire appel à une requête SQL sur une autre table, etc.:

[3] certains SGBDs comme SQL Server ou MySQL InnoDB vont en quelque sorte stocker les lignes sur le disque par ordre croissant en fonction de la la clé primaire: on comprend donc qu'il ne soit pas possible de trier la table sur plusieurs critères à la fois

```
CREATE TABLE Cine (
Nom_Cine CHAR(20) PRIMARY KEY,
Adresse  VARCHAR(60),
Telephone CHAR(8))
```

```
CREATE TABLE Cine (
Nom_Cine CHAR(20),
Adresse  VARCHAR(60),
Telephone CHAR(8),
PRIMARY KEY(Nom_Cine))
```

```
CREATE TABLE Cine (
Nom_Cine CHAR(20),
Adresse  VARCHAR(60),
Telephone CHAR(8),
CONSTRAINT pk_nom_cine PRIMARY KEY(Nom_Cine))
```

Figure 5: Trois manières équivalentes de déclarer les contraintes

```
CREATE TABLE Prog (
Nom_Cine CHAR(20),
Titre CHAR(20),
Salle INT NOT NULL CHECK (1 <= salle AND salle < 10),
Horaire TIME)
```

```
CREATE TABLE Prog (
...,
CHECK ('France'= SELECT pays FROM Cine WHERE Prog.Nom_Cine=Cine.nom))
```

⊘En ce cas, la contrainte est vérifiée à chaque modification de `Prog`, mais pas de `Cine`.⊘

2.2.6 Contraintes de clés étrangères

La clef étrangère doit être clé primaire (ou unique) dans la table référencée.

```
CREATE TABLE Prog (
Nom_Cine CHAR(20) REFERENCES Cine(Nom_Cine),
Titre CHAR(20),
Salle INT,
Horaire TIME,
CONSTRAINT fk_titre FOREIGN KEY (Titre) REFERENCES Film(Titre))
```

La contrainte vérifie ici que chaque titre de film au programme apparait dans une des lignes de la table `Film`. Un peu comme un "pointeur", chaque nom dans la table programme fait donc bien référence à une ligne de laa table cinéma.

⚠ La contrainte est vérifiée à chaque modification de `Prog` ET chaque modification de `Film`.

Prog

NOM_CINE	TITRE	SALLE	HORAIRE
ugc bercy	Le discours d'un roi	1	20h00
ugc bercy	Le discours d'un roi	3	20h00
ugc bercy	Bienvenus chez les ch'tis	2	22h00
ugc bercy	Inception	1	14h00
le champo	Le discours d'un roi	1	18h00
le campo	Inception	1	20h00

Film

TITRE	ACTEUR	MeS
Inception	Nolan	Page
Les petits mouchoirs	Cotillard	Canet
Le discours d'un roi	Hooper	Firth
Mon Idole	Canet	Canet
Bienvenus chez les ch'tis	Boon	Boon

2.2.7 Récapitulatif des contraintes d'intégrité

Considérons les contraintes :

```
-- dans Cine:
Telephone CHAR(8) NOT NULL
CONSTRAINT pk_nc PRIMARY KEY (Nom_Cine))
-- dans Prog:
CONSTRAINT fk_cine FOREIGN KEY (Nom_Cine) REFERENCES Cine(Nom_Cine)
UNIQUE (Nom_Cine,Titre, Horaire)
```

Les tables ci-dessous violent des contraintes (donc l'instance ci-dessous est impossible en présence des contraintes ci-dessus) :

Cine

NOM_CINE	ADRESSE	TELEPHONE
ugc bercy	3 rue...	06043494
le champo	17 av...	01049059
ugc bercy	18 bd...	

Prog

NOM_CINE	TITRE	SALLE	HORAIRE
ugc bercy	Le discours d'un roi	1	20h00
ugc bercy	Le discours d'un roi	3	20h00
ugc bercy	Le discours d'un roi	2	22h00
ugc bercy	Inception	1	14h00
le champo	Le discours d'un roi	1	18h00
le campo	Inception	1	20h00

N.B. les symboles (✎,★,...) n'ont rien de standard: je les utilise juste ici comme légende des violations

2.3 Modifier le schéma d'une base de donnée

Outre la création de nouvelles tables, on peut modifier le schéma d'une base de donnée en utilisant les instructions `ALTER TABLE, DROP TABLE`

```
DROP TABLE Prog
-- Erreur si d'autres objets dépendent de la table
```

En présence de clefs étrangères, respecter l'ordre pour détruire les tables, ou bien utiliser:

```
DROP TABLE Prog CASCADE
-- Détruit aussi les objets faisant référence à Prog
```

```
ALTER TABLE Prog ADD COLUMN jour: DATE;

ALTER TABLE Prog DROP COLUMN jour: DATE;

ALTER TABLE Prog ADD CONSTRAINT unicite_salles UNIQUE(nom_cine,salle,horaire);
```

3 SQL comme Langage de Manipulation de Données

Les principales opérations de SQL pour manipuler les données sont:

- SELECT : pour interroger
- INSERT : pour insérer des tuples dans une table
- UPDATE : pour modifier des données dans une table
- DELETE : pour supprimer des tuples dans une table

On peut considérer que SQL, comme l'algèbre relationnelle, manipule les données par lignes: pour ces deux langages, l'unité d'information est essentiellement la ligne. Donc chacune de ces opérations SQL manipule de façon à peu près indépendantes les différentes lignes d'une table.

3.1 Insérer des données: INSERT

```
INSERT INTO Prog (Nom_Cine, Titre, Salle, Horaire)
VALUES ('ugc', 'Dersou Ouzala', 1, '20h00');
-- ne pas oublier les ' ' pour les chaînes de caractères

INSERT INTO Prog (Nom_Cine, Horaire, Titre, Salle)
VALUES ('ugc', '10h00', 'Dersou Ouzala', NULL);
```

Préciser les noms de colonnes n'est pas nécessaire dans le premier exemple, mais l'est dans le second et permet en général:

- d'insérer la valeur par défaut (NULL sauf mention explicite d'un autre DEFAULT) pour les colonnes non spécifiées.
- de donner les valeurs dans le désordre.

⚠ L'insertion peut être refusée si elle contredit une clé (primaire ou étrangère). Si aucune clé primaire ou unique n'est définie sur la table il est possible de répéter plusieurs fois une même insertion, et la table ainsi obtenue contiendra plusieurs lignes identiques. Il est rare que l'on souhaite ce genre de comportement, ce qui est une des raisons pour lesquelles on définit généralement une clé primaire sur chaque table.

De nombreux SGBD proposent des commandes mixtes Update-insert

```
INSERT INTO produit(id,prix) VALUES (1,10.5)
  ON CONFLICT (id) DO UPDATE SET prix = 10.5; -- PostgreSQL
...
  ON DUPLICATE KEY UPDATE prix=10.5; -- MariaDB
```

3.2 Supprimer des données: DELETE

Comme pour les insertions, une suppression va concerner une (ou plusieurs) ligne complète. La suppression peut aussi échouer en si une clé étrangère fait référence à la ligne que l'on souhaite supprimer. En revanche, lorsque la clé étrangère a été définie avec l'option`ON DELETE CASCADE`, la suppression d'une ligne entrainera en cascade la suppression des lignes qui en dépendent.

```sql
DELETE FROM Prog WHERE Heure >= '20h00';

DELETE FROM Prog
WHERE Titre in (SELECT Titre FROM Film WHERE LOWER(Acteur) Like 'Jean-%');

-- Comportements possible en cas de Foreign Key: échec ou cascade.
```

Pour vider complètement une table, TRUNCATE est plus rapide.

```sql
TRUNCATE Prog;
```

Truncate ne parcourt pas les données et se contente de redéfinir la table comme "vide". C'est donc une opération quasi instantannée. Mais truncate ne parcourt pas la table, donc à réserver aux utilisateurs avertis! De plus on ne peut effectuer l'opération `TRUNCATE t` en présence de clé étrangère référençant `t` puisque `TRUNCATE` est une instruction DDL et ne vérifie donc pas si les lignes de `t` sont référencées par des lignes d'autres tables.
(pas d'entrée dans le journal⇒ perturbe MVCC, ne déclenche pas les triggers).

3.3 Modifier (mettre à jour) des données: UPDATE

```sql
UPDATE Prog
SET Horaire = '21h00'
WHERE Nom_Cine LIKE 'ugc%' AND Titre= 'Dersou Ouzala';
/* remplace l'horaire par '21h00' dans les lignes de la table
programme pour lesquelles le titre est 'D...' et le nom_cine commence
par 'ugc' */

-- Autres expressions possibles:
UPDATE Prog SET Horaire = Horaire+1; -- les séances seront retardées
UPDATE Produits SET PrixTTC = PrixHT * TVA;
```

Prog avant

NOM_CINE	TITRE	SALLE	HORAIRE
ugc bercy	Dersou Ouzala	1	20h00
ugc bercy	Dersou Ouzala	2	22h00
ugc bercy	Kagemusha	1	14h00
le champo	Dersou Ouzala	1	18h00
le champo	Kagemusha	1	20h00

Prog après mise à jour

NOM_CINE	TITRE	SALLE	HORAIRE
ugc bercy	Dersou Ouzala	1	21h00
ugc bercy	Dersou Ouzala	2	21h00
ugc bercy	Kagemusha	1	14h00
le champo	Dersou Ouzala	1	18h00
le champo	Kagemusha	1	20h00

3.4 Le coeur des requêtes SQL: SELECT FROM WHERE

Nous donnes ci-dessous pour certaines requêtes SQL l'expression équivalente de l'algèbre relationnelle. Bien sûr l'équivalence n'est pas tout à fait rigoureuse puisque les modèles diffèrent (doublons, etc).

Pour identifier la meilleure façon d'évaluer une requête, l'optimiseur de requêtes va estimer le coût des différentes expressions d'algèbre relationnelle permettant de calculer la requête, et sélectionnera ainsi le plan le plus efficace avant de compiler la requête. Pour estimer le coût d'un plan de requête, l'optimiseur se base sur des statistiques lui permettant d'estimer le coût de chaque stratégie (nombre de lignes dans la table, indexes disponibles...).

3.4.1 Requêtes SQL sur une seule table

Nous avons déjà vu comment formuler une requête affichant l'ensemble de la table, ou certaines colonnes de la table:

```sql
SELECT * FROM FILM;
```

```sql
SELECT titre, acteur FROM FILM;
```

Mais on souhaite généralement ajouter des conditions aux requêtes pour restreindre les lignes sélectionnées. Pour effectuer cette sélection, on utilise la clause `WHERE`, optionnelle, qui garde les tuples évaluant condition à "`Vrai`" et élimine les autres.

- La clause FROM contient les relations utiles à la requête: c'est en fait le produit cartésien de ces relations
- La clause SELECT contient les attributs (ou expressions) à garder dans le résultat
- La clause WHERE contient les conditions précisant quelles lignes garder

⚠ Les doublons ne sont pas éliminés dans le résultat.

Ce type de requête permet d'exprimer les sélections et projections de l'algèbre relationnelle, à ceci près que SQL n'élimine pas les doublons car il opère sur des multiensembles (et non des ensembles) de lignes.

```
SELECT * FROM FILM WHERE Acteur='Lonsdale'
```
$\leftrightarrow \sigma_{Acteur='Lonsdale'}(FILM)$

```
SELECT Titre
FROM FILM
WHERE Acteur='Lonsdale' OR Acteur='Astaire'
```
$\leftrightarrow \pi_{titre}(\sigma_{Acteur='Lonsdale' \vee Acteur='Astaire'}(FILM))$

Sémantique formelle (cas mono-relation):

SELECT $A_1, \ldots, A_k$
FROM R $\leftrightarrow$ $\pi_{A_1,\ldots,A_k}\sigma_C(R)$
WHERE C

Illustration de requête SQL: cas mono-relation

Prog

```
NOM_CINE   TITRE                SALLE HORAIRE
---------- -------------------- ----- -------
ugc bercy  Le discours d'un roi     1   20h00
ugc bercy  Le discours d'un roi     2   22h00
ugc bercy  Inception                1   14h00
le champo  Le discours d'un roi     1   18h00
le champo   Inception               1   20h00
```

```
SELECT Nom_cine, Titre
FROM Prog
WHERE Nom_cine = 'ugc bercy'
  AND HORAIRE < 23h00
  AND HORAIRE > 10h00
```

```
NOM_CINE   TITRE
---------- --------------------
ugc bercy  Le discours d'un roi
ugc bercy  Le discours d'un roi
ugc bercy  Inception
```

Les clauses SELECT et FROM, elles, ne sont pas optionnelles en général. Sauf dans le cas particulier où l'on souhaite sélectionner une expression constante. En ce cas, certains SGBD comme Postgres permettent de se passer de clause FROM tandis que d'autres imposent l'usage d'une table quasi fictive: DUAL

```
SELECT 2+3 -- Postgres, MariaDB
SELECT 2+4 FROM DUAL -- Oracle, MariaDB
```

3.4.2 Requêtes SQL sur plusieurs tables

Dans les bases de données, les informations sont souvent réparties sur plusieurs tables (en particulier du fait de la normalisation). SQL permet bien sûr d'établir des correspondances entre lignes de plusieurs tables à l'aide de jointures (ou de produits cartésiens). Très souvent la jointure porte sur une paire clé primaire/clé étrangère, même si ce n'est pas toujours le cas (en particulier on joint rarement sur une clé dans le cas d'une auto-jointure: jointure de deux copies d'une même table).

On rappelle que le produit cartésien est une opération particulièrement coûteuse, mais est rarement utilisé en pratique. Les jointures ont dans le pire des cas le même coût qu'un produit cartésien (quadratique pour deux tables, exponentiel dans le nombre de tables à joindre), mais sont moins souvent beaucoup plus efficaces en pratique. Une des principales tâches de l'optimiseur de requête est de choisir la meilleure manière d'évaluer les jointures (ordre de calcul des jointures, algorithmes utilisé...).

Une requête qui semble ne jamais terminer dans les TPs que nous ferons, c'est souvent une requête dans laquelle les conditions de jointures ont été oubliées, ce qui transforme les jointures en produit cartésien.

```
SELECT Nom_Cine, Film.Titre, Horaire
FROM Film, Prog
WHERE Film.Titre = Prog.Titre
  AND Acteur = 'M.Freeman'
```
$\leftrightarrow \pi_{Nom\_Cine,Film.titre,Horaire}(\sigma_{Acteur='M.Freeman'}(Film \bowtie Prog))$

Sémantique formelle (cas multi-relation):

SELECT $A_1, \ldots, A_k$ FROM $R_1, \ldots, R_p$ WHERE C $\leftrightarrow$ $\pi^*_{A_1,\ldots,A_k}(\sigma_C(R_1^* \times \cdots \times R_p^*))$

-projection multiensembliste
-tous attributs distincts

Cette syntaxe ne distingue pas les conditions de jointures des autres conditions dans la clause `WHERE`. Nous verrons plus tard une autre syntaxe faisant la distinction. Mais la syntaxe ci-dessus est tout à fait "légitime" car l'optimiseur de requête est capable de repérer dans la clause `WHERE` les conditions mettant e oeuvre deux tables, et va directement appliquer un algorithme de calcul de jointure. L'utilisateur n'a donc pas à s'inquiéter en utilisant cette syntaxe puisque le SGBD ne va donc pas commencer par calculer la requête sous forme d'un produit cartésien suivi de sélections.

Le SGBD, lorsqu'il veut évaluer une requête, la transforme en un plan d'exécution qui correspond essentiellement à l'algèbre relationnelle avec une sémantique de multi-ensemble. Du point de vue sémantique, la clause de la requête qui va être évaluée en premier est donc le produit cartésien du FROM en premier puis la sélection du WHERE puis enfin la projection du SELECT.
La vérité est en fait plus compliquée: au moment de l'exécution de la requête le SGBD cherchera si il y a d'autres expressions de l'algèbre relationnelle équivalentes qui peuvent être plus efficace. Mais cet "ordre" fictif des clauses est un bon outil pour comprendre la construction des requêtes.

Prog

```
NOM_CINE   TITRE                SALLE HORAIRE
---------- -------------------- ----- -------
ugc bercy  Le discours d'un roi     1   20h00
ugc bercy  Le discours d'un roi     2   22h00
ugc bercy  Inception                1   14h00
le champo  Le discours d'un roi     1   18h00
le champo   Inception               1   20h00
```

Cine

```
NOM_CINE   ADRESSE TELEPHONE
---------- ------- ---------
ugc bercy  3 rue... 06043494
le champo  17 av... 01049059
nef chava 18 bd... 04387953
```

```
SELECT Telephone, Horaire
FROM Prog, Cine
WHERE Prog.Nom_cine = Cine.Nom_cine
  AND Titre = 'Inception'
```

```
TELEPHONE   HORAIRE
----------  ---------
06043494    14h00
01049059    20h00
```

3.4.3 Préfixer les attributs du nom de leur table

Il arrive que plusieurs tables possèdent des attributs de même nom. La requête SQL doit alors préfixer ces attributs par le nom de la table d'où ils sont issus pour lever l'ambiguïté, comme dans la requête précédente ou celle ci-dessous. En réalité, le SGBD ajoute automatiquement le nom de table en préfixe pour chaque attribut mais l'inférence impossible si l'attribut apparait dans plusieurs tables: c'est pourquoi l'utilisateur doit impérativement spécifier la table dans ce cas. En théorie, il est même assez judicieux de préfixer tous les attributs par le nom de la table correspondante. [4].

```
SELECT Nom_Cine, Film.Titre, Horaire
FROM Film, Prog
WHERE Film.Titre = Prog.Titre
  AND Acteur = 'M.Freeman'
```

[4] Cela aide à maintenir la validité de la requête lorsque le schéma évolue suite à une commande de type ALTER TABLE qui pourrait créer des attributs homonymes

3.4.4 Renommage de tables: `table t_alias`

SQL permet de renommer (ou plus exactement d'attribuer un alias à) des tables. Ceci permet entre autres de distinger plusieurs copies d'une même table, ou de donner un nom à une table résultant d'un calcul intermédiaire (sous requête). Renommer une table est en particulier nécessaire lorsque la même table apparaît plus d'une fois parmi les tables de la clause FROM: il est alors (en général) nécessaire à la fois de renommer les tables, et de préfixer les attributs de ces tables par l'alias de table correspondant.

Les films avec leur MeS et leurs acteurs dans lesquels joue M-F Pisier ?
$\pi_{\text{Titre}}(\sigma_{\text{Actrice=M-F.Pisier}}(\textbf{film})) \bowtie \textbf{film}$

```
SELECT F2.Titre , F2.MeS, F2.Acteur
FROM  FILM  F1, FILM  F2
WHERE F1.Titre = F2.Titre  AND
  F1.Acteur = 'M-F. Pisier'
```

Interprétations de F1 et F2:

- F1 et F2 sont des copies *virtuelles* de **film**
- F1 et F2 sont des variables utilisées pour désigner n'importe quel couple de n-uplets de **film**

⚠ Certains SGBDs comme Postgres et MariaDB autorisent la syntaxe : table AS alias. Mais il vaut mieut éviter le mot-clé AS pour les alias de table: Oracle, par exemple, ne permet pas d'utiliser ce mot clé pour définir un alias de table.

3.4.5 Renommage de colonnes: `expression [AS] alias`

SQL permet d'attribuer un nom de colonne à une expression avant de renvoyer le résultat d'une requête SQL. Ceci sert à

- nommer une colonne résultant d'un calcul,
- renommer une colonne de façon plus explicite
- permettre l'union (ou autre opération) entre deux tables en rendant les schémas compatibles...

Toutes les personnes ayant participées au tournage du film "Marion":

```
SELECT Acteur AS Personne FROM Film WHERE Titre = 'Marion'
UNION
SELECT MeS AS Personne FROM Film WHERE Titre = 'Marion'
```

Le mot clé AS est facultatif.

3.4.6 Elimination des doublons: `SELECT DISTINCT`

Pour éliminer les doublons, on ajoute distinct après le mot-cléf SELECT.

Liste des films projetés dans chaque cinéma:

```
SELECT Nom_Cine, Titre
FROM Prog
```

```
NOM_CINE   TITRE
---------- --------------------
ugc bercy  Le discours d'un roi
ugc bercy  Le discours d'un roi
ugc bercy  Le discours d'un roi
ugc bercy  Inception
le champo  Le discours d'un roi
le champo  Inception
```

```
SELECT DISTINCT Nom_Cine, Titre
FROM Prog
```

```
NOM_CINE   TITRE
---------- --------------------
ugc bercy  Le discours d'un roi
ugc bercy  Inception
le champo  Le discours d'un roi
le champo  Inception
```

3.4.7 Expressions

Dans la clause `SELECT` ou dans les conditions, SQL ne se limite pas à l'usage de simples noms de colonnes: on peut utiliser des expressions telles que:

- une constante

- un attribut
- *expr1* * *expr2* (ou +,-,/)
- une opération sur les chaînes de caractères
  - ↪ `ch1 || ch2` : concaténation
  - ↪ `LOWER(ch)` : met en minuscules
  - ↪ `UPPER(ch)` : met en majuscules
  - ↪ `SUBSTR(ch,i,j)` : extrait la sous chaîne de longueur j débutant à l'indice i
- `CASE WHEN` condition `THEN` expr `ELSE` expr `END`
- ...

```
SELECT 3*montant FROM Ventes;
SELECT SUBSTR('ABCDEFG',3,4) FROM DUAL;  -- (oracle): 'CDEF'
SELECT substring('Thomas' from 2 for 3); -- (postgresql): 'hom'
```

Les conditions booléennes, elles, ont la forme suivante. On les retrouve typiquement dans la clause `WHERE` mais on peut aussi les retrouver dans un CASE, par exemple.

- *attr* `IS NULL`
- *expr* `op` *expr* `op`$\in (=, <, <=, >=, <>)$
- *expr* `BETWEEN` *val1* `AND` *val2*
- *expr* `IN` *(val1,val2,...)*
- *expr* `LIKE` *string-pattern*
  - ↪ `%` : n'importe quelle chaîne de caractère
  - ↪ `_` : n'importe quel caractère
- *condition1* `AND` *condition2*, *condition1* `OR` *condition2*, `NOT` *condition1*...

Films

TITRE	ACTEUR
Rien à Declarer	Boon
La plage	Canet
Bienvenus chez les ch'tis	Boon

```
SELECT Titre
FROM Films
WHERE LOWER(Titre) LIKE '%bien%'
  OR Titre LIKE '%declarer%'
--LOWER: convertir en minuscule
```

```
TITRE
---------------------------
Bienvenus chez les ch'tis
```

⚠ On rappelle que les comparaisons de chaines de caractères sont sensibles à la casse.

3.4.8 Requêtes en présence de NULL

`NULL` représente une valeur inconnue.

Une expression booléenne peut prendre 3 valeurs en SQL: `Vrai`, `Faux`, ou `Inconnu`.

2=2: `Vrai`

2=NULL: `Inconnu`

NULL= NULL: `Inconnu`

SQL utilise donc *logique à 3 valeurs* pour évaluer conditions:

NOT `Inconnu`= `Inconnu`

OR retourne `Vrai` si un des arguments est `Vrai`, `Faux` si les deux arguments sont `Faux`, `Inconnu` sinon

AND retourne `Vrai` si les deux arguments sont `Vrai`...

Pour tester si un attribut est défini, on utilise `attr IS NULL/IS NOT NULL`.

⚠ La clause `WHERE` ne sélectionne que les lignes pour lesquelles la condition s'évalue à `Vrai`. Les lignes renvoyant `Inconnu` ou `Faux` sont donc ignorées. Les conditions dans les CASE ou clause HAVING ont le même comportement: `Inconnu` n'est pas considéré comme vrai.

3.4.9 Quelques requêtes SQL pas très intuitives (pièges)

Que renvoient les requêtes suivantes ?

```
SELECT * FROM t WHERE (name <> 'Jean') OR name='Jean'
```

Réponse:

```
echo                    "Q2V0dGUgcmVxdWV0ZSBzw6lsZWN0aW9ubmUgbGVzIGxpZ25lcyBwb3VyIGxlc"\
"3F1ZWxsZXMgSmVhbiBuJ2VzdCBwYXMgbnVsbC4gSWwgc2VyYWl0IGJpZW4gcHLDq"\
"WbDqXJhYmxlIGRlIHLDqcOpY3JpcmUgbGEgcmVxdcOqdGUgZW4gOiBTRUxFQ1QgKi"\
"BGUk9NIHQgV0hFUkUgbmFtZSBJUyBOT1QgTlVMTAo=" | base64 --decode
```

```
SELECT * FROM t WHERE name = name
```

```
echo "Y29tbWUgY2ktZGVzc3VzCg==" | base64 --decode
```

```
SELECT * FROM t WHERE name = NULL
```

```
echo "TmUgcmVudm9pZSBhdWN1bmUgbGlnbmUK" | base64 --decode
```

Autre comportement contre-intuitif: sur certaines instances un peu particulières, la requête suivante ne renvoie pas $R.A \cap (S.A \cup T.A)$. Lorsqu'une des tables S ou T est vide, le produit cartésien est vide, donc la requête SQL ne renvoie aucune ligne. Alors que $S.A \cup T.A$ et $R.A \cap (S.A \cup T.A)$ ne sont pas nécessairement vides.

```
SELECT DISTINCT R.A
FROM R,S,T
WHERE R.A=S.A OR R.A=T.A
```

3.4.10 Trier le résultat d'une requête: `ORDER BY`

Une relation dans un SGBD relationnel est un multi-ensemble, il n'y a pas d'ordre entre les lignes. L' ordre d'affichage du résultat d'une requête SQL est donc arbitraire et peu potentiellement changer d'une exécution à l'autre (même si c'est rare). Il est en fait souvent déterminé par le plan de requête choisi par l'optimiseur: les lignes étant souvent affichées dans l'ordre où elles sont calculées. En revanche, SQL permet à l'utilisateur de choisir l'ordre dans lequel le résultat lui est affiché avec l'instruction `ORDER BY`.

Liste des films projetés dans chaque cinéma triés par titre:

```
SELECT Nom_Cine, Titre
FROM Prog
ORDER BY Titre
```

```
NOM_CINE   TITRE
---------- --------------------
ugc bercy  Inception
le champo  Inception
ugc bercy  Le discours d'un roi
ugc bercy  Le discours d'un roi
ugc bercy  Le discours d'un roi
le champo  Le discours d'un roi
```

```
SELECT Nom_Cine, Titre
FROM Prog
ORDER BY Titre DESC, Nom_Cine ASC
```

```
NOM_CINE   TITRE
---------- --------------------
le champo  Le discours d'un roi
ugc bercy  Le discours d'un roi
ugc bercy  Le discours d'un roi
ugc bercy  Le discours d'un roi
le champo  Inception
ugc bercy  Inception
```

- Le tri se fait par défaut par ordre croissant (🇬🇧 `ASCending`). Le suffixe DESC permet d'obtenir un ordre décroissant (🇬🇧 `DESCending`).
- l'ordre naturel de l'attribut dépend de son type (entier, chaîne de caractères...), et... de la collation. Selon le SGBD – voire même, pour un SGBD configuré de façon identique, selon l'OS sur lequel se trouve le serveur – l'ordre des chaînes de caractères peut varier. Par exemple, le `A` majuscule peut se trouver avant ou après le `b` minuscule.

- on peut trier sur plusieurs colonnes par ordre lexicographique: ORDER BY a DESC, b ASC va trier par a, puis départagera les ex-aequos sur a en les triant sur b.

3.5 Opérations ensemblistes: UNION, INTERSECT, EXCEPT/MINUS

SQL permet exactement les mêmes opérations ensemblistes que l'algèbre relationnelles: union, intersection, différence avec les opérateurs `UNION, INTERSECT, EXCEPT` (respectivement). Pour la différence, certains SGBD utilisent `MINUS` (Oracle, MariaDB, MySQL) et d'autres `EXCEPT` (PostgreSQL, SQL Server). Ces opérations ensemblistes éliminent par défaut les doublons, mais l'ajout du mot-clé `ALL` leur donne une sémantique multiensembliste. Les relations en jeu dans l'opération doivent dans tous les cas avoir même schéma.

Les personnes ayant travaillé sur le film 'Marion':

```
SELECT Acteur AS Personne FROM Film WHERE Titre = 'Marion'
UNION
SELECT MeS AS Personne FROM Film WHERE Titre = 'Marion'
```

$$\leftrightarrow \rho_{MeS \to Personne}(\pi_{MeS}(\sigma_{Titre='Marion'}(Film))) \\ \cup \rho_{Acteur \to Personne}(\pi_{Acteur}(\sigma_{Titre='Marion'}(Film)))$$

Les personnes ayant travaillé sur le film 'Marion', comptées 1 fois par rôle:

```
SELECT Acteur AS Personne FROM Film WHERE Titre = 'Marion'
UNION ALL
SELECT MeS AS Personne FROM Film WHERE Titre = 'Marion'
```

Les titres des films à l'affiche dans lesquels a joué M-F Pisier:

```
SELECT Titre FROM Film WHERE Acteur = 'M-F. Pisier'
  INTERSECT
SELECT Titre FROM Prog
ORDER BY Titre
```

Les titres des films qui ne sont pas à l'affiche:

```
SELECT Titre FROM Film
  EXCEPT -- MINUS sous Oracle ou MariaDB
SELECT Titre FROM Prog
```

Soit t un tuple présent $n_1 \geq 0$ fois dans le résultat de la requête Q_1 et n_2 fois dans Q_2. Q_1 UNION ALL Q_2 renvoie $n_1 + n_2$ copies de t, Q_1 INTERSECT ALL Q_2 en renvoie $\min(n_1, n_2)$ et Q_1 EXCEPT ALL Q_2 en renvoie $n_1 - n_2$ si $n_1 \geq n_2$ (et 0 sinon).

3.6 ⊘Sous-requêtes⊘

Je n'en parlerai pas pour ne pas inciter à les utiliser. Sachez qu'elles existent et peuvent être utiles dans certains cas. Mais on considère généralement qu'il ne faut pas en abuser car elles sont souvent difficiles à optimiser pour le SGBD, et difficiles à comprendre pour l'utilisateur.

3.7 Syntaxe spécifique pour les jointures

Rappel: jusqu'à présent nous avons utilisé les syntaces suivantes pour les produits cartésiens et jointures:

```
-- produit cartésien:
SELECT R.a,S.b FROM R,S
```

```
-- jointure sur l'attribut c
SELECT R.a,S.b FROM R,S WHERE R.c=S.c
```

On parle parfois de jointure pour des conditions portant sur plusieurs tables même lorsqu'il ne s'agit pas d'équijointures (c'est à dire que la comparaison n'est pas une condition d'égalité). On peut même dans ce cas utiliser la syntaxe ci-dessus. Comme nous l'avons déjà indiqué, la syntaxe ci-dessus ne pose pas de problème d'efficacité lors de l'évaluation des requêtes, grâce à l'optimiseur de requêtes. En revanche, certains auteurs considèrent que pour des requêtes en "production" (c'est à dire lorsque l'on programme des requêtes pour une application en entreprise), il est préférable d'utiliser une syntaxe plus lisible qui distingue les jointures des autres conditions de sélection. La doc postgres[5] indique clairement que c'est une question de choix personnel.

```
-- jointure (interne=habituelle)
SELECT F1.Titre, F2.Acteur
FROM FILM F1
JOIN FILM F2 ON F1.Titre = F2.Titre
WHERE F1.Acteur = 'M-F. Pisier'
```

```
-- jointure naturelle
SELECT Nom_Cine,Film.Titre, Horaire
FROM Film
NATURAL JOIN Prog
WHERE Acteur = 'M.Freeman'
```

```
-- précise les attributs de jointure
SELECT attr_a, t2.attr2,...
FROM t1
INNER JOIN t2
USING(attr_a,attr_b,...,attr_c)
```

La jointure naturelle est généralement à proscrire car on ne voit pas les attributs sur lesquels se fait la jointure, et la requête n'est pas stable en cas de modification du schéma de la base de donnée. On préférera donc utiliser un INNER

[5] https://www.postgresql.org/docs/current/queries-table-expressions.html

JOIN avec USING qui précise sur quels attributs on effectue la jointure naturelle: cela revient donc à restreindre la jointure naturelle aux attributs listés. Noter que lorsque l'on utilise la jointure naturelle ou USING, on ne précise plus l'origine des attributs de jointure car les 2 colonnes sont "fusionnées". Le "inner" dans INNER JOIN est à comprendre par opposition à l'OUTER JOIN que nous verrons ensuite. Les mot-clés INNER et OUTER sont facultatifs, car la jointure sera interne par défaut, et externe si l'on spécifie LEFT/RIGHT/FULL JOIN.

La syntaxe peut bien sûr être utilisée pour plusieurs jointures avec plusieurs tables:

```
SELECT ...
FROM t1
  JOIN t2 ON t1.x = t2.y AND t1.u = t2.v
  JOIN t3 on t1.a = t3.b AND t2.a = t3.c
```

Par défaut (en l'absence de parenthèses, qui, ici, ne changeraient rien au résultat de toute façon), les jointures se font dans l'ordre (de gauche à droite). Noter aussi que les mots clés INNER, OUTER sont facultatifs.

3.7.1 Jointures externes: OUTER JOIN

Contrairement aux syntaxes ci-dessus, OUTER JOIN n'est pas un "sucre syntaxique", mais permet d'effectuer une opération qu'il serait difficile d'exécuter sans cette instruction; la jointure externe. Un LEFT OUTER JOIN va afficher le résultat de la jointure interne auquel il ajoute toutes les lignes de la table gauche "éliminées" de la jointure car n'ayant pas de ligne correspondante à droite. Ces lignes de la table de gauche sont alors complétées par des nulls dans la partie droite pour respecter le schéma du résultat.

```
SELECT attr1, attr2,...
FROM nom_table1 LEFT OUTER JOIN nom_table2
USING(attr_a,attr_b,...,attr_c)
```

Roles

TITRE	ID_ACT	PERSONNAGE
It's a Wonderful Life	35	George Bailey
It's a Wonderful Life		Mary Bailey
It's a Wonderful Life	36	Clarence Odbody
Rear Window	35	LB Jeffries
The Shawshank Redemption	40	red

Acteur

ID	NOM	PRENOM
35	Stewart	James
40	Freeman	Morgan
50	Serrault	Michel

```
SELECT titre, personnage, nom
FROM roles LEFT OUTER JOIN acteurs ON id_act=id;
```

TITRE	PERSONNAGE	NOM
Rear Window	LB Jeffries	Stewart
It's a Wonderful Life	George Bailey	Stewart
The Shawshank Redemption	red	Freeman
It's a Wonderful Life	Mary Bailey	
It's a Wonderful Life	Clarence Odbody	

On définit de façon symmétrique les RIGHT OUTER JOIN, et on étend naturellement les jointures externes aux FULL OUTER JOIN:

RIGHT OUTER JOIN: affiche les lignes de la table droite sans correspondance et les complète par des null
FULL OUTER JOIN: affiche les lignes des tables gauche ou droite sans correspondance et les complète par des null

Roles

TITRE	ID_ACT	PERSONNAGE
It's a Wonderful Life	35	George Bailey
It's a Wonderful Life		Mary Bailey
It's a Wonderful Life	36	Clarence Odbody
Rear Window	35	LB Jeffries
The Shawshank Redemption	40	red

Acteur

ID	NOM	PRENOM
35	Stewart	James
40	Freeman	Morgan
50	Serrault	Michel

```
SELECT titre, personnage, nom
FROM roles FULL OUTER JOIN acteurs ON id_act=id;
```

TITRE	PERSONNAGE	NOM	
It's a Wonderful Life	George Bailey	Stewart	in both
Rear Window	LB Jeffries	Stewart	in both
The Shawshank Redemption	red	Freeman	in both
It's a Wonderful Life	Mary Bailey		in LEFT OUTER JOIN
It's a Wonderful Life	Clarence Odbody		in LEFT OUTER JOIN
		Serrault	in RIGHT OUTER JOIN

3.8 Les requêtes d'agrégation: GROUP BY

Les bases de données peuvent contenir des millions de lignes. Pour extraire des informations des tables, les analystes vont souvent chercher à résumer les informations contenues dans ces tables. Dans une feuille de calcul Excel, on retrouve fréquemment le calcul du total d'une colonne. On effectue le même type d'opération en SQL avec les fonctions d'agrégations. Les principales fonctions d'agrégation sont `COUNT()`, `SUM()`, `MAX()`, `MIN()`, `AVG()`.

L'idée est de résumer un ensemble de ligne en une seule. Les requêtes d'agrégation permettent de partitionner la table en différents groupes de lignes, chaque groupe étant ensuite résumé par une seule ligne (ex: pour chaque cinéma, son nombre de salle et le nombre de films qui y sont programmés). Un cas particulier est le cas où l'on va définir un unique groupe contenant l'ensemble de la table, qui sera alors résumée en une seule ligne (ex: nombre total de films dans la base, et de salles de cinéma dans la base).

Dans tous les cas, le ou les différents groupes de lignes forment une partition de la table : chaque ligne de la table (la table obtenue après les sélections `WHERE`) est attribuée à un groupe (un seul groupe), et aucun groupe n'est vide.

3.8.1 Partitionner une table en groupes, la clause GROUP BY

Toute agrégation est calculée sur une table, donc sur le résultat d'une requête de type `SELECT * FROM ... [WHERE ...]`. Après avoir donc calculé les jointures et conditions de sélections, le SGBD dispose d'une unique table qu'il va alors partitionner en différents groupes de lignes qui vont ensuite être chacun résumés par une seule ligne.[6]

Dans la requête d'agrégation ci-dessous, les lignes de la table film ayant la même valeur sur la colonne `acteur` vont former un groupe. On va résumer chaque groupe en une ligne contenant le nom de l'acteur, et son nombre de films (le nombre de ligne du groupe).

Film

TITRE	ACTEUR
Rien à Declarer	Boon
Brice de Nice	Cornillac
La vie de Chantier	Boon
Ensemble c'est tout	Canet
Mon Idole	Canet
The dark knight rises	Cotillard
Les petits mouchoirs	Cotillard
La plage	Canet
Inception	Cotillard
Bienvenus chez les ch'tis	Boon

```
SELECT acteur, COUNT(*)
FROM film
GROUP BY acteur
```

ACTEUR	COUNT(*)
Canet	3
Cotillard	3
Boon	3
Cornillac	1

De façon plus générale, la clause `GROUP BY` contient une liste d'expressions $e_1, ...e_n$ (généralement des attributs mais on peut utiliser des expressions plus complexes). Un groupe est alors formé des tuples partageant la même valeur sur tous les attributs de la liste.

[6] En pratique l'évaluation de la requête par le SGBD ne se fait pas ainsi par étape, puisque le SGBD va généralement calculer les groupes au fur et à mesure sans attendre d'avoir calculé le résultat des jointures et sélections. Mais il est plus simple pour comprendre SQL de considérer que l'on procède ainsi par étape.

Prog

```
NOM_CINE   TITRE           SALLE HORAIRE
---------- --------------- ----- -------
ugc bercy  Les évadés          1   21h00
ugc bercy  Les évadés          3   20h00
ugc bercy  La vie est belle    2   10h00
ugc bercy  Le parrain          1   14h00
le champo  La vie est belle    1   18h00
ugc bercy  La vie est belle    1   20h00
```

```
SELECT nom_cine, titre, COUNT(*) NB,
  MIN(Horaire) Tot, MAX(Horaire) Tard,
FROM Prog
GROUP BY nom_cine, titre
ORDER BY NB DESC
```

```
NOM_CINE   TITRE           NB    TOT   TARD
---------- --------------- -- ------- ------
ugc bercy  Les évadés       2  20h00  21h00
ugc bercy  La vie est belle 2  10h00  22h00
ugc bercy  Le parrain       1  14h00  14h00
le champo  La vie est belle 1  18h00  18h00
```

Il faut faire attention lorsque l'on utilise l'agrégat `COUNT`:

- `COUNT(*)` retourne le nombre de lignes dans le groupe.
- `COUNT(a)` retourne le nombre de lignes non nulles sur l'expression a dans le groupe.
- `COUNT(DISTINCT a)` retourne le nombre de valeurs *distinctes* sur l'expression a dans le groupe.

Roles

```
TITRE               ID_ACT PERSONNAGE
------------------- ------ -------------------
It's a Wonderful Life  35 George Bailey
It's a Wonderful Life  36 Mary Bailey
Noblesse Oblige        10 Ethelred D'Ascoyne
Noblesse Oblige        10 Duc de Chalfont
Noblesse Oblige        10 Lord Horatio D'Ascoyne
Noblesse Oblige        11 Edith d'Ascoyne
Noblesse Oblige        12  Sibella Holland
```

Acteur

```
 ID NOM      PRENOM
--- -------- --------
 35 Stewart  James
 36 Donna    Reed
 10 Alec     Guinness
 11 Valerie  Hobson
 12
```

```
SELECT titre,
  count(*) Nb,
  count(id_act) Nb_id,
  count(distinct id_act) Nb_Distinct,
  count(nom) Nb_nom
FROM roles, acteur
WHERE id_act=id
GROUP BY titre;
```

```
         titre         | nb | nb_id | nb_distinct | nb_nom
-----------------------+----+-------+-------------+--------
 It's a Wonderful Life |  2 |     2 |           2 |      2
 Noblesse Oblige       |  5 |     5 |           3 |      4
```

```
SELECT count(*) Nb,
  count(id_act) nb_id,
  count(distinct id_act) nb_distinct,
  max(titre) dernier
FROM roles, acteur
WHERE id_act=id
/* agrégat  mais pas de "group by"
  => groupe implicite sur l'ensemble de la table */
```

```
 nb | nb_id | nb_distinct |     dernier
----+-------+-------------+-----------------
  7 |     7 |           5 | Noblesse Oblige
```

Il est de la même façon possible de calculer `SUM(DISTINCT a)`, `AVG(DISTINCT a)` au lieu de `SUM(a)`, `AVG(a)` si l'on souhaite éliminer les doublons avant de calculer une somme ou moyenne, mais cet usage est plus rare.

3.8.2 Agréger l'ensemble de la table

Lorsqu'une requête SQL contient une fonction d'agrégation mais pas de clause GROUP BY, le SGBD va former un seul groupe contenant l'ensemble de la table, qui va donc être résumée en une seule ligne, comme illustré sur la figure précédente. Le comportement est donc le même que si l'on avait groupé les lignes sur une valeur constante (exemple: `GROUP BY ()`) sauf que l'usage est justement de ne pas utiliser de group by dans ce cas.

3.8.3 La clause HAVING

HAVING permet de ne garder que les groupes satisfaisant une condition:

```
TITRE                     ACTEUR      D
------------------------- --------- ----
Rien à Declarer           Boon       108
Brice de Nice             Cornillac   98
Ensemble c'est tout       Canet      107
Le Dernier Vol            Canet       94
Joyeux Noël               Canet      116
Bienvenus chez les ch'tis Boon       106
```

```
SELECT acteur, COUNT(*) NB,  AVG(D) DMoy
FROM film
GROUP BY acteur
HAVING COUNT(*) >= 2
```

```
ACTEUR      NB  DMOY
---------- ---  ------
Canet        3  105.67
Boon         2  107
```

3.8.4 Expressions autorisées dans la clause SELECT en présence d'agrégat

Une requête invalide:

```
-- Q1
SELECT a, b, sum(c)
FROM t
GROUP BY a
```

Une requête valide:

```
-- Q2
SELECT a, sum(c)
FROM t
GROUP BY a, b
```

Autre requête valide:

```
-- Q3
SELECT a, sum(a)
FROM t
GROUP BY a
```

Lorsque les contraintes du schéma imposent une dépendance fonctionnelle $a \to b$, certains SGBD (ex: PostgreSQL) peuvent tolérer la requête Q1 : ce comportement est autorisé dans une partie optionnelle du standard SQL. Mais ce n'est pas portable, donc il est bien préférable d'inclure l'attribut b dans le groupe. À l'inverse, la requête Q2 est parfaitement valide : rien n'impose d'afficher tous les attributs utilisés pour le groupement dans le résultat de la requête même. On souhaite souvent afficher les valeurs utilisées pour regrouper les données, mais ce n'est pas nécessaire. De la même façon, il est rare d'utiliser dans un agrégat la valeur utilisée pour grouper les données, mais c'est autorisé.

3.8.5 ⊘Sémantique des Agrégats⊘

Qu'est-ce qui caractérise un agrégat et les distingue parmi les fonctions plus générales ? Un agrégat est une fonction qui prend en entrée un ensemble de valeur et renvoie une (une seule) valeur. À quelques détails près, chacun des agrégats que nous utilisons couramment (SUM,COUNT,MAX,...)peut être vu comme un opérateur commutatif et associatif. L'agrégat f initialise un accumulateur a à une valeur par défaut (0 pour la somme,$-\infty$ pour le maximum), puis met à jour la valeur de l'accumulateur en la remplaçant par $f(a, v)$ pour chaque valeur v de l'ensemble à agréger. La moyenne AVG(e) n'est pas associative mais peut-être calculée comme SUM(e)/COUNT(e).

Certaines constructions plus récentes de SQL (fonctions analytique) permettent de trier les lignes à l'intérieur d'un groupe avant de calculer des fonctions qui ne sont alors pas nécessairement commutatives. Donc il existe des agrégats qui ne sont pas commutatifs (ex : ARRAY_AGG sous PostgreSQL, LISTAGG sous Oracle).

3.8.6 ⊘Agrégats en présence de NULLs⊘

Beaucoup d'agrégats comme SUM(e),MAX(e)... ignorent les valeurs nulls : ils calculent respectivement la somme ou le maximum des valeurs non nulles dans l'ensemble.

Pourquoi c1 et c2 n'ont pas toujours la même valeur ci-dessous ?

```
SELECT SUM(a)+SUM(b) c1,
  SUM(a+b) c2
FROM t
```

Réponse:

```
echo "TG9yc3F1ZSBhIGVzdCBudWwgZXQgcGFzIGIsIG91IHZpY2UgdmVyc2EsIGErYiB2YXV"\
"0IG51bGwgZG9uYyBsYSBsaWduZSBlc3QgaWdub3LDqWUgZGFucyBjMiwgbWFpcyBwYXMgZGFucyBjMQo="\
| base64 --decode
```

3.9 Comment rédiger une requête SQL

On rédige une requête SQL en s'appuyant sur le schéma, qu'il faut donc avoir bien compris. La plupart des requêtes s'écrivent sans opération ensembliste et sans sous-requêtes. Je vous suggère alors de suivre l'ordre "intuitif" de traitement des clauses, mais c'est plus une suggestion de raisonnement qu'autre chose: à vous de décider l'ordre qui vous convient le mieux pour rédiger la requête:

1. Commencer par identifier les éléments/tables dont on a besoin pour obtenir une ligne du résultat.
2. Réfléchir ensuite
   (a) si l'on va regrouper des lignes par agrégation
   (b) si l'on va combiner plusieurs requêtes dans une opération ensembliste
   (c) si l'on a besoin de joindre plusieurs lignes d'une même table (autojointures)
3. Ecrire les conditions de jointure, puis les autres conditions de sélection (clause WHERE)
4. Ecrire les expressions de la clause SELECT
5. Ecrire les critères de tri.

La liste suivante rappelle l'ordre d'évaluation intuitif des clauses. Cet ordre explique aussi assez bien que l'on ne puisse réutiliser les alias de la clause SELECT dans la clause GROUP BY, entre autres exemples.

```
FROM
WHERE
GROUP BY
HAVING
SELECT
ORDER BY
```

3.10 Quelques outils techniques

3.10.1 Guillemets: simple ou double

Il est nécessaire d'utiliser des guillemets simples en SQL pour délimiter les chaînes de caractères (et pour échapper un caractère): les guillemets doubles servent à définir des identificateurs (noms de colonnes ou de tables) particuliers. Les identificateur entre guillemets permettent de définir des noms:

- sensible à la casse
- utilisant des espaces
- utilisant des mots clés réservés par le SGBD

```
SELECT a FROM t; -- la colonne nommée a
SELECT 'a'; -- constant string 'a'
CREATE TABLE "ma table" ("select" int); -- identificateur entre guillemets
SELECT "select" from "ma table"; -- faisable mais à éviter.
```

Dans le cadre de ce cours nous éviterons tout simplement d'utiliser les guillemets doubles. Je ne recommande d'ailleurs pas trop leur utilisation, et il est maladroit de mélanger des identificateurs entre guillemets avec d'autres.[7]

Dans une requête SQL, le symbole d'échappement est ':

```
SELECT Titre FROM Film WHERE Acteur = 'O''Brien'; -- cherche O'Brien
```

Mais les applications utilisent plus souvent une requête préparée avec des paramètres.

Pour plus d'informations (ex: comment échapper des caractères): https://docs.postgresql.fr/current/sql-syntax.html#sql-syntax-lexical

⚠ Deux guillemets simples ne sont donc pas équivalents à un guillemet double!

3.10.2 ⊘Les schémas sous postgresql⊘

Un cluster de BD postgresql contient une ou plusieurs BD. Chaque BD comporte un ou plusieurs schémas, qui est essentiellement un espace de noms. Le schéma par défaut est le schéma `public`.

```
CREATE SCHEMA mon_schema;
CREATE TABLE mon_schema.t(id int) -- voire: base.schema.table
SELECT a FROM mon_schema.t;

CREATE SCHEMA mon_schema2;
CREATE TABLE mon_schema2.t(id int) -- valide car pas dans le même schéma
-- DROP SCHEMA mon_schema CASCADE; -- pour effacer un schéma non-vide
```

En général on évite les noms qualifiés. Le schéma utilisé pour une création de table sera le premier dans le chemin de parcours. Le schéma utilisé pour une requête sur la table t sera le premier schéma dans le chemin de parcours dans lequel on trouve une table t:

```
SET search_path TO mon_schema, mon_schema2, public;
SHOW search_path; -- SELECT * FROM t utilisera la table dans mon_schema
```

```
 search_path
--------------
mon_schema, mon_schema2, public;
```

https://docs.postgresql.fr/current/ddl-schemas.html

[7] Un exemple de souci que cela peut causer: les identifiants qui ne sont pas entre guillements sont par défaut convertis en minuscules sous postgresql, en majuscule sous oracle.

3.10.3 Les séquences

Les SGBD proposent des outils pour générer automatiquement des valeurs successives pour la clé primaire: les séquences et les types auto-incrémentés. On peut avoir 0,1 voire plusieurs séquence par table (ca peut donc faire beaucoup de séquences dans une base). Une séquence est essentiellement un compteur que l'on va pouvoir incrémenter pour générer une suite de nombres. On s'en sert principalement pour générer des identifiants.

```
CREATE SEQUENCE serie INCREMENT BY 100 START 101;
SELECT nextval('serie'); -- 101
SELECT currval('serie'); -- le résultat du dernier nextval: ici, 101
```

On retrouve des outils similaires sous Oracle, MariaDB, MySQL, SQL Server...

```
CREATE TABLE t (
    a SERIAL,
    b string
);
-- revient à
CREATE SEQUENCE t_a_seq;
CREATE TABLE t (
    a NOT NULL DEFAULT nextval('t_a_seq'),
    b string
);
```

3.10.4 ⊘Comportements spécifiques à certains SGBDs (non exhaustif)⊘

Je déconseille fortement d'utiliser ces particularités:

- PostgreSQL:

```
SELECT country
FROM cities
GROUP BY city; -- valid if city is PK. Or more generally FD city -> country
-- Optional extensions of standard
```

- MySQL, MariaDB (InnoDB engine):

```
cities(id, city, country)
... FOREIGN KEY REFERENCES (cities.country)
-- FK referencing a non unique column!
-- Non standard
```

- PostgreSQL, MySQL, MariaDB

```
SELECT a AS bbb
FROM t
GROUP BY bbb;
```

https://www.postgresql.org/docs/current/features.html