

Bases de données

Chapitre 4 - SQL

Source : Céline Rouveirol

CM4 du 17/05/2022



Bases de SQL

On distingue plusieurs sortes de commandes SQL parmi lesquelles :

- LDD (langage de définition de données),
- LMD (langage de manipulation des données) c'est-à-dire du LMJ (langage de mise à jour) et du LID (langage d'interrogation des données),
- LCD (langage de contrôle des données).



Sources des transparents

- M.P. Dorville/F. Goasdoué, LRI, Université Paris Sud
- V. Mogbil, LIPN, Université Paris Nord
- J. Ullman <http://infolab.stanford.edu/~ullman/>



Le Langage de Définition de Données

- Ensemble de commandes qui définit une base de données et les objets qui la composent
- la définition d'un objet inclut
 - sa création: CREATE
 - sa modification ALTER
 - sa suppression DROP



CREATE TABLE

La forme générale d'une création de table est :

```
CREATE TABLE nom_de_table (
    déclaration_attribut [...,déclaration_attribut],
    [contrainte_de_table,...,contrainte_de_table]) ;
```

```
déclaration_attribut ::=
    nom_de_colonne type_de_donnees
    [DEFAULT expression]
    [<liste contrainte_de_colonne>]
```



Types et contraintes en SQL

► Types

- BOOLEAN → booléens
- INTEGER, SMALLINT, NUMBER → numériques entiers
- FLOAT, REAL, DECIMAL(n,d), NUMBER(n,d) → numériques réels
- CHAR(n), VARCHAR(n), VARCHAR2(n) → caractères et chaînes
- DATE, TIME, DATETIME → dates

► Contraintes d'intégrité

- PRIMARY KEY → clef primaire
- FOREIGN KEY → clef étrangère
- REFERENCES → contrainte d'inclusion
- CHECK → contrainte générale
- UNIQUE → valeur unique et clefs candidates
- NOT NULL → obligation de valeur
- CONSTRAINT → nom d'une contrainte



CREATE TABLE: contrainte de colonne

```
contrainte_de_colonne ::=
CONSTRAINT nom_de_contrainte
{
    NOT NULL | NULL |
    UNIQUE | PRIMARY KEY |
    CHECK (condition) |
    REFERENCES table_de_reference
    [ (colonne_de_reference) ]
    [ ON [DELETE|UPDATE] [CASCADE|SET NULL]]
```



CREATE TABLE: contrainte de table

```
contrainte_de_table ::=
CONSTRAINT nom_de_contrainte
{
    UNIQUE (liste_de_colonne)
    | PRIMARY KEY (liste_de_colonne)
    | CHECK (condition) |
    FOREIGN KEY (liste_de_col)
    REFERENCES table_de_ref
    [ (liste_de_col.de_ref) ]
    [ ON [DELETE|UPDATE] [CASCADE|SET NULL]]
```



CREATE TABLE

- CREATE TABLE crée une nouvelle table, initialement vide, dans la base de données courante. La table sera la propriété de l'utilisateur lançant la commande.
- Si un nom de schéma est donné (par exemple, CREATE TABLE monschema.matable ...), alors la table est créée dans le schéma spécifié. Sinon, elle est créée dans le schéma actuel. Le nom de la table doit être distinct des noms des autres tables, séquences, index ou vues dans le même schéma.



CREATE TABLE

- Les clauses de contrainte optionnelle spécifient les contraintes que les nouvelles lignes ou les lignes mises à jour doivent satisfaire pour qu'une opération d'insertion ou de mise à jour réussisse. Une contrainte est un objet SQL qui aide à définir l'ensemble de valeurs valides de plusieurs façons.



Create table: exemple

Soit le schéma suivant *Compagnie*(*comp* : chaîne, *nomComp* : chaîne)
Pilote(*brevet* : chaîne, *nom* : chaîne, *age* : entier, *compa* : chaîne)
Pilote[*compa*] \subseteq *Compagnie*(*comp*)

Les commandes de création sont :

```
Create table Compagnie(
  com varchar2(30),
  nomComp varchar2(50) not null,
  primary key (comp));
```

```
Create table Pilote(
  brevet varchar2(20),
  nom varchar2(30) not null,
  age smallint,
  compa varchar2(30),
  primary key (brevet),
  foreign key (compa) references Compagnie (comp),
  constraint agechk check (age>=0));
```



CREATE TABLE: contrainte de clé

- La contrainte de clé primaire PRIMARY KEY spécifie qu'une ou plusieurs colonnes d'une table peuvent contenir seulement des valeurs uniques et non nulles.
- PRIMARY KEY est simplement une combinaison de UNIQUE et NOT NULL, mais identifier un ensemble de colonnes comme clé primaire fournit aussi des métadonnées sur le concept du schéma: une clé primaire implique que d'autres tables peuvent se lier à cet ensemble de colonnes comme un unique identifiant pour les lignes.
- Une seule clé primaire peut être spécifiée pour une table, qu'il s'agisse d'une contrainte de colonne ou de table.



CREATE TABLE, contrainte de clé étrangère

Exemple

Soit la relation *Vend*(*bar*, *biere*, *prix*) et la relation *Biere*(*nom*, *couleur*, *origine*). On peut imposer qu'une bière vendue dans un bar soit une "vraie" bière, i.e., qu'elle soit effectivement décrite dans la relation *Biere*.

La contrainte qui impose qu'une bière de la relation *Vend* soit décrite dans la relation *Biere* est une contrainte de *clé étrangère* ou *foreign key* en anglais.



CREATE TABLE, contrainte de clé étrangère

Deux manières de préciser une contrainte de clé étrangère: utiliser le mot-clé REFERENCES

- dans la déclaration d'un attribut (pour les clés mono-attribut)
- comme une contrainte de table: FOREIGN KEY (attribute1,...,attributen) REFERENCES <relation> (attribute1',...,attributen')
- attention, les attributs référencés doivent être déclarés comme des PRIMARY KEY ou UNIQUE.



Contrainte de clé étrangère, exemple

```
CREATE TABLE Bière (  
  nom CHAR(20) PRIMARY KEY,  
  couleur CHAR(20),  
  origine CHAR(20));
```

```
CREATE TABLE Vend (  
  bar CHAR(20),  
  biere CHAR(20) REFERENCES Bière(nom),  
  prix REAL );
```



Contrainte de clé étrangère, autre syntaxe

```
CREATE TABLE Bière (  
  nom CHAR(20) PRIMARY KEY,  
  couleur CHAR(20),  
  origine CHAR(20));
```

```
CREATE TABLE Vend (  
  bar CHAR(20),  
  biere CHAR(20),  
  prix REAL,  
  FOREIGN KEY(biere) REFERENCES Bière(nom));
```



Clé étrangère, pourquoi ?

S'il existe une contrainte de clé étrangère d'un ensemble d'attributs d'une relation R à une clé de la relation S , deux violations sont possibles:

- Une insertion ou une maj de R introduit des valeurs qui n'existent pas dans S
- Un effacement ou une maj de S fait que certains nuplets de R "pointent dans le vide"

Exemple

Soit $R = Vend$ et $S = Biere$.

- une insertion ou maj dans $Vend$ qui introduit une bière qui n'existe pas doit être rejetée
- Un effacement ou une maj de $Biere$ qui supprime une bière qui est référencée par des nuplets de $Vend$ peut être traité de 3 manières différentes.



Stratégie de propagation de modification

- Default : rejette la modification
- Cascade : fait les mêmes changements dans $Vend$, i.e. si effacement/maj d'un nplet dans $Biere$, alors effacement/maj des nuplets de $Vend$ référençant cette bière
- Set NULL : les nuplets référençant une bière supprimée dans $Biere$ sont mis à NULL pour l'attribut $biere$ dans $Vend$.

Lors de la déclaration d'une clé étrangère, il est possible de choisir les stratégies SET NULL ou CASCADE indépendamment pour l'effacement ou la maj. Après la déclaration de foreign-key : ON [UPDATE | DELETE] [SET NULL | CASCADE]. La stratégie par défaut est de rejeter l'effacement et la maj.



17/82



18/82

CREATE TABLE: propagation de modification

Exemple

```
CREATE TABLE Vend (
  bar CHAR(20),
  biere CHAR(20),
  prix REAL,
  FOREIGN KEY(biere)
    REFERENCES Biere(nom)
    ON DELETE SET NULL
    ON UPDATE CASCADE
);
```



CREATE TABLE: contrainte de colonne

- ▶ La distinction entre contrainte de colonne et contrainte de table est que celle de colonne ne s'applique qu'à une colonne à la fois.
- ▶ La clause NOT NULL impose de renseigner le champ par une valeur valide.



19/82



20/82

CREATE TABLE: contrainte CHECK

Les contraintes de la forme CHECK (condition) sont des contraintes de domaine. Elles utilisent les opérateurs suivants (cf SELECT):

- A BETWEEN a AND b: équivalent à $A \geq a$ AND $A \leq b$
- A IN (a1,a2,...,an) équivalent à $A=a1$ OR $A=a2$ OR ... OR $A=an$
- A LIKE motif où le motif est une expression telle que le caractère % remplace toute suite (même vide) de caractères et le caractère _ remplace exactement un caractère.

La condition peut utiliser le nom de l'attribut, mais tout autre nom de relation ou d'attribut doit apparaître dans une sous requête



Contrainte de domaine, exemple

```
CREATE TABLE Vend (  
    bar CHAR(20),  
    biere CHAR(20) CHECK ( biere IN  
        (SELECT nom FROM Bière)),  
    prix REAL CHECK ( prix <= 5.00 )  
);
```

Les contraintes de domaine ne sont vérifiées que quand une valeur de la table courante est ajoutée ou mise à jour. CHECK (prix <= 5.00) vérifie chaque nouveau prix et refuse la mise à jour si le prix est supérieur à 5 (euros). Par contre CHECK (biere IN (SELECT nom FROM Bière)) n'est pas vérifiée lors d'une suppression de Biere (par opposition à la contrainte foreign-key).



21/82

22/82

Contrainte au niveau nuplet

- CHECK (<condition>) peut apparaître comme élément d'un schéma de relation.
- La condition peut référencer n'importe lequel des attributs de la relation
- Tout autre attribut ou relation doit apparaître dans une sous-requête
- Ces contraintes sont vérifiées lors de l'ajout (INSERT), de l'insertion ou de la mise à jour (UPDATE) de nuplets



Contrainte au niveau nuplet, exemple

```
CREATE TABLE Vend (  
    bar CHAR(20),  
    biere CHAR(20),  
    prix REAL,  
    CHECK (bar = 'Joe''s Bar' OR  
        prix <= 5.00)  
);
```



23/82

24/82

Modification de la structure d'une table

Il faut préciser la nature de la modification. Plusieurs syntaxes :

- ALTER TABLE nom_table ADD champ type | CONSTRAINT nom ctr;
- ALTER TABLE nom_table MODIFY champ type[options] ;
- ALTER TABLE nom_table DROP champ | CONSTRAINT nom ;

C'est pour cela qu'il est bon de nommer les contraintes !!!

Exemples

```
ALTER TABLE Pilote ADD indice integer;
ALTER TABLE Pilote DROP CONSTRAINT agechk;      ALTER
TABLE product_information
MODIFY (min_price DEFAULT 10);
```



ALTER TABLE (1)

ALTER TABLE change la définition d'une table existante. Il y a plusieurs variantes:

- ADD nom_de_colonne domaine et ADD (liste_de_colonnes) ajoutent une nouvelle colonne à la table en utilisant la même syntaxe que CREATE TABLE. Une nouvelle colonne ne peut avoir la contrainte NOT NULL que si la table est vide.
- ADD contrainte_de_table ajoute une nouvelle contrainte à une table en utilisant la même syntaxe que CREATE TABLE.



ALTER TABLE (2)

- DROP COLUMN nom_de_colonne supprime une colonne d'une table. Les indexes et les contraintes de table référençant cette colonne sont automatiquement supprimés. Il faut utiliser l'option CASCADE si certains objets hors de la table dépendent de cette colonne (clés étrangères,...).
- DEFAULT expression ne modifie pas les lignes déjà présentes dans la table. Les valeurs par défaut ne s'appliquent qu'aux prochaines commandes INSERT.



ALTER TABLE (3)

- RENAME TO change le nom d'une table ou le nom d'une colonne de la table. Elle est sans effet sur les données stockées.
- DROP CONSTRAINT supprime des contraintes d'une table. Plusieurs contraintes peuvent avoir le même nom, toutes ces contraintes sont supprimées.
- ▶ ALTER TABLE effectue une copie temporaire de la table originale. Les modifications sont faites sur cette copie, puis la table originale est effacée, et enfin la copie est renommée pour remplacer l'originale. Cette méthode permet de rediriger toutes les commandes automatiquement vers la nouvelle table sans pertes. Durant l'exécution de ALTER TABLE, la table originale est lisible par d'autres clients. Les modifications et insertions sont reportées jusqu'à ce que la nouvelle table soit prête.



Suppression d'une table

`DROP TABLE nom_table`: efface toutes les données et le schéma d'une table

Exemple

```
DROP TABLE Pilote;
```



29/82

Dictionnaire des Données

- ▶ Une des parties les plus importantes d'une BD
- ▶ Un ensemble de tables qui stockent toute l'information de la base de données.
- ▶ Un dictionnaire des données contient:
 - la définition de tous les schémas d'objets de la base de données (tables, vues, indexes, fonctions, triggers, ...)
 - l'espace mémoire alloué et utilisé par ces objets
 - les valeurs par défaut pour les attributs
 - toute Information sur les contraintes d'intégrité
 - le nom des utilisateurs, le rôle et les droits de chaque utilisateur
 - ...
- ▶ Le dictionnaire est structuré en *tables* et *vues*
- ▶ Toutes ces tables et vues sont stockées dans le tablespace SYSTEM de cette BD
- ▶ Ces tables sont en read only et peuvent uniquement être requêtées (`SELECT ...`)



31/82

DROP TABLE

`DROP TABLE nom_de_table [, ...] [AS (clause_select)] [CASCADE]`;

- ▶ `DROP TABLE` supprime des tables de la base de données. Seul son propriétaire peut détruire une table.
- ▶ `DROP TABLE` supprime tout index, règle, déclencheur et contrainte existant sur la table cible. Néanmoins, pour supprimer une table qui est référencée par une contrainte de clé étrangère d'une autre table, `CASCADE` doit être spécifié (`CASCADE` supprimera la contrainte de clé étrangère, pas l'autre table elle-même)



30/82

Organisation du dictionnaire

- ▶ Vues avec préfixe `USER_`
 - Décrit l'environnement de l'utilisateur dans la BD, informations à propos du schéma des objets créés par l'utilisateur, les droits accordés par l'utilisateur, ...
 - Ne décrit que ce qui est pertinent pour l'utilisateur
 - `SELECT object_name, object_type FROM USER_OBJECTS`;
- ▶ Vues avec préfixe `ALL_`
 - Vues qui décrivent la perspective de l'utilisateur sur tout le SGBD. Ces vues donnent les informations sur les objets auxquels l'utilisateur a accès en dehors des objets dont l'utilisateur est propriétaire.
 - `SELECT owner, object_name, object_type FROM ALL_TABLES`; spécifie toutes les tables auxquelles un utilisateur accède et leurs propriétaires
- ▶ Les vues avec le préfixe `DBA` ne peuvent être requêtées que par l'administrateur oracle.



32/82

Quelques tables ou vues du dictionnaire

- ▶ DICTIONARY (DICT): vue du dictionnaire
- ▶ USER.TABLES: tables et vues créées par l'utilisateur
- ▶ USER.CATALOG: tables et vues sur lequel l'utilisateur a des droits, sauf celles du dictionnaires des données
- ▶ USER.TAB_COLUMNS: attributs des colonnes des tables ou vues créées par l'utilisateur
- ▶ USER.INDEXES: index créés par l'utilisateur
- ▶ ...



Exemples d'utilisation

Par exemple, recherche d'information sur les contraintes

- ▶ On recherche les noms des tables et vues contenant le mot 'CONSTRAINTS': `Select table_name from dict where table_name like '%CONSTRAINTS%'`
- ▶ On affiche le nom des colonnes de la table USER_CONSTRAINTS: `describe USER_CONSTRAINTS;`
- ▶ On cherche les informations que l'on veut: `select constraint_name, constraint_type, table_name from users_constraints`
- ▶ `select * from user_tables;`
- ▶ `select table_name from user_tables;`
- ▶ `select * from user_indexes where table_name = 'TOTO';`



33/82



34/82

Le langage de manipulation de données : LMD

- Ensemble de commandes qui permet la consultation et la mise à jour des objets créés par le langage de définition de données
- la mise à jour inclut
 - l'insertion de nouvelles données
 - la modification de données existantes
 - la suppression de données existantes

Exemple

- `SELECT <liste champ(s)> FROM <liste nom_table(s)> [WHERE condition(s)] [options];`
- `INSERT INTO <nom_table> [(<liste champ(s)>)] VALUES (<liste valeurs>);`
- `UPDATE <nom_table> SET <champ> = <expression> [WHERE <liste condition(s)>];`
- `DELETE FROM <nom_table> [WHERE <liste condition(s)>];`



Exemples INSERT - UPDATE - DELETE

- ▶ Ajout d'un tuple
`INSERT INTO PRODUIT
VALUES (400, Nouveau produit, 78.90);`
- ▶ Mise à jour d'un attribut
`UPDATE CLIENT SET Nom='Dudule' WHERE NumCli = 3;`
- ▶ Suppression de tuples
`DELETE FROM CLIENT WHERE Ville = 'Lyon';`



35/82



36/82

Insertion dans une table

Pour insérer un nuplet dans une relation:

```
INSERT INTO <relation>
VALUES ( <list of values> );
```

Exemple

On veut ajouter à la table de schéma Apprécie(Client, Bière) le fait que Jean aime la 1664.

```
INSERT INTO Apprécie
VALUES('Jean', '1664');
```



37/82

Insérer plusieurs tuples à la fois

- ▶ Il est possible d'ajouter à une table le résultat d'une sous-requête:

```
INSERT INTO <relation>
( <subquery> );
```

Exemple

Si on suppose que la table Fréquente(Client, Bar) stocke la liste des clients d'un bar, on peut créer une relation ConnaissancesJean(Nom) qui stocke tous les clients qui fréquentent les mêmes bars que Jean.

```
INSERT INTO ConnaissancesJean
(SELECT d2.Client
 FROM Fréquente f1, Fréquente f2
 WHERE f1.Client = 'Jean' AND
       f2.Client <> 'Jean' AND
       f1.bar = f2.bar
);
```



39/82

Spécifier des attributs dans un INSERT

On peut ajouter à la relation une liste de noms d'attributs, si:

- ▶ on a oublié l'ordre des attributs de la relation
- ▶ certains attributs ont des valeurs manquantes (le système complètera les valeurs manquantes à NULL ou à une valeur par défaut spécifiée)

Exemple

On veut ajouter à la table de schéma Apprécie(Client, Bière) le fait que Jean aime la 1664.

```
INSERT INTO Apprécie (Bière, Client)
VALUES('1664', 'Jean');
```



38/82

Mise à jour des données

- ▶ UPDATE : mise à jour des données d'une table
- ▶ La modification est effectuée sur toutes les lignes de la table ou sur les seules lignes qui vérifient une condition de sélection

```
UPDATE <nom_table>
SET <colonne1> = <expression1> [... , <colonneN> =
<expressionN> ]
[WHERE <liste conditions> ]
```

```
UPDATE <nom_table>
SET (colonne1,..., colonnen) =
SELECT ...
[WHERE conditions]
```

- ▶ DELETE: suppression des lignes d'une table
- ```
DELETE FROM <nom_table> [WHERE <liste conditions>]
```



40/82

## Extraction de données: commande SELECT

- ▶ extrait les données d'une ou plusieurs tables, selon certains critères, et range le résultat de la sélection dans une nouvelle table
- ▶ une requête SQL s'exprime au moyen de la commande SELECT de la façon suivante:

SELECT <cible> : ce que l'on veut  
 FROM <liste table(s)>: dans quelles tables  
 [WHERE <liste condition(s)>]: selon des critères d'extraction, éventuellement



## Select-From-Where: Exemple

```
SELECT nom FROM Bière
WHERE origine = 'Belge';
```

```
SELECT nom FROM Bière
WHERE origine = 'Belge' AND couleur = 'ambrée';
```



41/82



42/82

## Syntaxe commande SELECT

Voici une version simplifiée de SELECT. Nous verrons après une série d'exemples une version plus détaillée.

```
SELECT [ALL|DISTINCT] nom_de_colonne [,liste]
FROM nom_de_table [alias] [,liste]
[WHERE
{ condition |
[NOT] EXISTS (clause_select) |
nom_de_colonne IN (clause_select) |
nom_de_colonne operateur [ALL|ANY] (clause_select) }
[GROUP BY nom_de_colonne [,liste_nom_de_colonne]]
[HAVING condition]
[ORDER BY nom_de_colonne [ASC|DESC] [,liste]]
```

On considère l'algèbre relationnelle étendue au cas multi-ensembliste. On précise DISTINCT dans le champs SELECT pour retrouver un resultat ensembliste. ALL est la valeur par défaut.



## Exemples Opérateur DISTINCT

- ▶ Nombre total de commandes
  - ▶ SELECT COUNT(\*) FROM COMMANDE;
  - ▶ Nombre de clients ayant passé commande
  - ▶ SELECT COUNT( DISTINCT NumCli) FROM COMMANDE;
- | COMMANDE | NumCli   | Date | Quantite |
|----------|----------|------|----------|
| 1        | 22/09/05 | 1    |          |
| 3        | 22/09/05 | 5    |          |
| 3        | 22/09/05 | 2    |          |

COUNT(NumCli) donne pour résultat : 3  
 COUNT(DISTINCT NumCli) donne pour résultat : 2



43/82



44/82

## Expressions dans les clauses SELECT

- ▶ Une expression bien formée peut apparaître comme un élément de la clause SELECT.

## Exemple

En utilisant la table Vend(Bar, Bière, Prix):

```
SELECT Bar, Bière, Prix * 1.28 AS PrixDollar
FROM Vend;
```

- ▶ D'autres opérations arithmétiques et fonctions mathématiques:

```
SELECT salaire + prime FROM finances;
SELECT age-annee FROM employe;
SELECT base*taux FROM finances;
SELECT sortie/quantite FROM finances;
utilisation de abs(valeur), ceil(valeur), cos(valeur), ...
```



## Exemples Fonctions d'agrégat

- ▶ Elles opèrent sur un ensemble de valeurs.
  - AVG() : moyenne des valeurs
  - SUM() : somme des valeurs
  - MIN(), MAX() : valeur minimum, valeur maximum
  - COUNT() : nombre de valeurs
- ▶ Moyenne des prix des produits
- ▶ SELECT AVG(PrixUni) FROM PRODUIT;



## Éliminer les duplicats dans un groupement

- ▶ Utiliser DISTINCT dans un groupement
- ▶ Le nombre de prix différents pour la 1664
- ▶ SELECT COUNT(DISTINCT prix) FROM Vend WHERE bière = '1664';
- ▶ Les nuplets qui ont la valeur NULL pour l'attribut d'agrégat ne contribuent jamais au groupement (sum, average, count, min ou max)
- ▶ SELECT count(\*) FROM Vend WHERE bière = '1664'; compte le nombre de bars qui vendent de la 1664
- ▶ SELECT count(prix) FROM Vend WHERE bière = '1664' compte le nombre de bars qui vendent de la 1664 à un prix connu



## La clause FROM

- spécifie le nom de la ou des tables dans lesquelles les données seront sélectionnées
- lorsque plusieurs tables figurent dans cette clause, le système effectue une opération de produit cartésien ou de jointure entre les tables
  - jointure: qualification sur les attributs de jointure dans la clause WHERE
  - produit cartésien: pas de qualification
- les lignes des tables sélectionnées par les opérations de jointure sont celles qui vérifient les conditions entre les attributs de jointure (comparaison entre les valeurs des champs clés des différentes tables) plus les critères d'extraction.



## La clause Where

- ▶ spécifie les critères ou conditions d'extraction portant sur les valeurs des champs sélectionnés
- ▶ les conditions peuvent être combinées à l'aide des opérateurs AND et OR
- ▶ les critères de recherche s'expriment par
  - des comparaisons entre valeurs de champs et/ou constantes
  - des recherches par intervalle (BETWEEN)
  - l'appartenance ou non à un ensemble (IN)
  - la recherche par motif - pattern matching - (LIKE)
  - des tests de nullité (comparaison avec la valeur NULL : IS NULL)



## Exemple

En utilisant les relations Apprécie(Client, Bière) et Fréquente(Client, Bar), trouver l'ensemble des bières qui sont appréciées par au moins une personne qui fréquente le Joe's Bar.

```
SELECT Bière
FROM Apprécie, Fréquente
WHERE Bar = 'Joe's Bar' AND
Fréquente.Client = Apprécie.Client;
```



49/82

50/82

## Syntaxe de la commande SELECT

- \*: sélectionne tous les champs
- *liste champs*: sélectionne les champs de *liste tables*
- *expressions*: combinaison d'attributs et/ou d'opérateurs et/ou de fonctions dont l'évaluation correspond à une valeur
- *agrégats*: fonctions de calcul sur des groupes de lignes sélectionnées
  - count(expression)
  - avg(expression)
  - min(expression)
  - max(expression)
  - sum(expression)



## Lien AR - SQL

Soit  $(\mathcal{R}, att, dom)$  un schéma de BD tel que  $R_1 \cup \dots \cup R_n \subseteq \mathcal{R}$  et  $\{A_1, \dots, A_m\} \subseteq att(\cup_i R_i)$  pour  $i \in \{1, \dots, n\}$ .

L'expression algébrique suivante

$$\pi[A_1, \dots, A_m](\sigma[p](R_1 \times \dots \times R_n))$$

où  $p$  est un prédicat sur  $R_1 \times \dots \times R_n$

est équivalente à la requête

```
SELECT A_1, ..., A_m
FROM R_1, ..., R_n
WHERE p;
```



51/82

52/82

## Exemples traduction AR - SQL

- ▶ Projection:  $\pi[\text{Nom}, \text{Prenom}](\text{CLIENT})$ , i.e. Noms et Prénoms des clients, uniquement
- ▶ `SELECT Nom, Prenom FROM CLIENT;`
- ▶ Sélection :  $\sigma[\text{Ville} = \text{'Lyon'}](\text{CLIENT})$ , i.e. Clients qui habitent à Lyon
- ▶ `SELECT * FROM CLIENT WHERE Ville = 'Lyon';`



## Exemples traduction AR - SQL

- ▶  $\sigma[\text{Quantite} \geq 3](\text{COMMANDE})$ , i.e. Commandes en quantité au moins égale à 3
- ▶ `SELECT * FROM COMMANDE WHERE Quantite >= 3;`
- ▶  $\sigma[50 \leq \text{PrixUni} \leq 100](\text{PRODUIT})$ , i.e. Produits dont le prix est compris entre 50 et 100 euros
- ▶ `SELECT * FROM PRODUIT WHERE PrixUni BETWEEN 50 AND 100;`
- ▶  $\sigma[\text{Quantite IS NULL}](\text{COMMANDE})$ , i.e. Commandes en quantité indéterminée
- ▶ `SELECT * FROM COMMANDE WHERE Quantite IS NULL;`



53/82



54/82

## Variable nuplet

- ▶ Parfois, on a besoin de deux copies de la même table pour faire une requête
- ▶ Il est possible de distinguer les deux copies en faisant suivre le nom de la même relation par un nom de variable nuplet, dans la clause FROM.
- ▶ Il est toujours possible d'introduire des variables nuplet pour rendre la requête plus lisible.

## Exemple

Sélectionner les paires de bières différentes: `SELECT b1.nom, b2.nom FROM Biere b1, Biere b2 WHERE b1.origine = b2.origine AND b1.nom < b2.nom;`

Jointure naturelle et  $\Theta$ -jointure

- ▶  $R \bowtie S$ , sachant que  $R$  et  $S$  partagent les attributs  $A$  s'exprime par `SELECT * FROM R, S WHERE R.A = S.A;`
- ▶  $R \bowtie_{\Theta} S = \sigma[\Theta](R \times S)$  où  $\Theta$  est un prédicat sur  $R \times S$  se traduit par `SELECT * FROM R, S WHERE  $\Theta$ ;`



55/82



56/82



## Exemples Jointures

- ▶ On a  $CLIENT \bowtie COMMANDE = \sigma[CLIENT.NumCli=COMMANDE.NumCli](CLIENT \times COMMANDE)$
- ▶  $(\sigma[CLIENT.NumCli=COMMANDE.NumCli](CLIENT \times COMMANDE))$ , i.e. liste des commandes avec le nom des clients
- ▶ 

```
SELECT Nom, Date, Quantite
FROM CLIENT, COMMANDE
WHERE CLIENT.NumCli = COMMANDE.NumCli;
```
- ▶ 

```
SELECT Client.nomClient, Client2.nomClient
FROM Client, Client AS Client2
WHERE Client.noTelephone = Client2.noTelephone
and Client.nomClient <> Client2.nomClient; , i.e., les
Clients qui ont le même numéro de téléphone
```



## Exemples Theta-Jointures

- ▶  $R \text{ JOIN } S \text{ ON } \langle \text{condition} \rangle$
- ▶ On veut calculer la jointure de  $R(a, b)$  et  $S(b, c)$ : 

```
select * FROM R JOIN S Frequents ON R.b = S.b;
```
- ▶ Résultat de schéma  $(a, b, b, c)$  avec  $R.b = S.b$ .
- ▶ 

```
select * from R JOIN S using (b);
```



## Exemples Jointures

- ▶  $\pi[NumCli, Nom, Date, Quantite](CLIENT \bowtie COMMANDE)$
- ▶ Idem avec le numéro de client en plus et le résultat trié
- ▶ 

```
SELECT C1.NumCli, Nom, Date, Quantite
FROM CLIENT C1, COMMANDE C2
WHERE C1.NumCli = C2.NumCli
ORDER BY Nom;
```
- ▶ Utilisation d'alias (C1 et C2) pour alléger l'écriture + tri par nom.



## Renommer des attributs

Si on veut que le résultat ait des noms d'attributs différents, on peut utiliser AS <nouveau nom> pour renommer un attribut.

## Exemple

```
SELECT nom FROM Bière as NomBière
WHERE origine = 'Belge';
```



## Syntaxe clause SELECT : Requêtes multi-tables

Union, intersection, différence

- ▶ les opérateurs de l'algèbre relationnelle (INTERSECT, UNION, MINUS) s'appliquent sur des requêtes dont les résultats ont même arité (même nombre de colonnes et mêmes types d'attributs)
- ▶ exemple: rechercher les noms et numéros de téléphone des Employés qui sont aussi des Clients
- ▶ 

```
SELECT nomClient AS nomPersonne, noTelephone FROM Client
 INTERSECT
 SELECT nomEmploye AS nomPersonne, noTelephone FROM Employe;
```



## Requêtes imbriquées

- ▶ Les conditions exprimées dans la clause WHERE peuvent être des conditions sur des relations (et non plus sur des valeurs scalaires).
- ▶ En général ces conditions consistent en l'existence d'au moins un tuple dans la relation testée ou en l'appartenance d'un tuple particulier à la relation.
- ▶ La requête de la clause WHERE établit dynamiquement un critère de recherche pour l'interrogation principale



61/82

62/82

## Requêtes imbriquées, exemple

- ▶ Quels sont les employés qui occupent la même fonction que 'MERCIER' ?
- ▶ 

```
SELECT nomEmploye, fonction FROM Employe
 WHERE fonction = (select fonction FROM Employe
 WHERE nomEmploye= 'MERCIER') ;
```
- ▶ Rechercher les employés qui ont un salaire supérieur à la moyenne des salaires de tous les employés en précisant le service où ils travaillent
- ▶ 

```
SELECT nomService, nomEmploye, salaire
 FROM Employe as E, Service as S
 WHERE E.service = S.service AND salaire >
 (select AVG(salaire) FROM Employe);
```



## Requêtes imbriquées

- ▶ Les conditions que l'on peut exprimer sur une relation  $R$  construite avec une requête imbriquée s'utilisent avec les opérateurs suivants:
  - EXISTS  $R$ ; renvoie VRAI si  $R$  n'est pas vide, FAUX sinon
  - $t$  IN  $R$  où  $t$  est un tuple dont le type est celui de  $R$ ; renvoie VRAI si  $t$  appartient à  $R$ , FAUX sinon
  - $v$  cmp ANY  $R$  où  $cmp$  est un opérateur de comparaison; renvoie VRAI si la comparaison avec au moins un des tuples de la relation  $R$  est vraie, FAUX sinon
  - $v$  cmp ALL  $R$  où  $cmp$  est un opérateur de comparaison; renvoie VRAI si la comparaison avec tous les tuples de la relation  $R$  est vraie, FAUX sinon



63/82

64/82

## Exemples prédicats EXISTS et NOT EXISTS

- ▶ Clients qui ont passé au moins une commande
- ▶ 

```
SELECT * FROM CLIENT C1
WHERE EXISTS (SELECT *
FROM COMMANDE C2
WHERE C1.NumCli = C2.NumCli);
```
- ▶ Clients qui n'ont passé aucune commande
- ▶ 

```
SELECT * FROM CLIENT C1
WHERE NOT EXISTS (SELECT *
FROM COMMANDE C2
WHERE C1.NumCli = C2.NumCli);
```



65/82

## Exemples de jointure exprimée avec le prédicat IN

- ▶ Nom des clients qui ont commandé le 23/09/2006
- ▶ 

```
SELECT Nom FROM CLIENT
WHERE NumCli
IN (SELECT NumCli FROM COMMANDE
WHERE Date = '23-09-2005');
```



67/82

## Exemples prédicats ALL / ANY

- ▶ Numéros des clients qui ont commandé au moins un produit en quantité supérieure à chacune [à au moins une] des quantités commandées par le client numéro 1
- ▶ 

```
SELECT DISTINCT NumCli FROM COMMANDE
WHERE QUANTITE > ALL [ANY] (SELECT QUANTITE
FROM COMMANDE
WHERE NumCli = 1);
```



66/82

## Syntaxe clause SELECT

- ▶ La clause ORDER BY trie le résultat d'une requête selon l'ordre des champs spécifiés dans la clause  

```
SELECT * FROM CLIENT
ORDER BY nom [DESC];
```
- ▶ La clause GROUP BY
  - regroupe les enregistrements sélectionnés
  - il est possible de spécifier une ou plusieurs conditions sur les enregistrements regroupés au moyen de la clause HAVING



68/82

## Groupement

- ▶ Une instruction SELECT-FROM-WHERE peut être suivie par un GROUP BY et une liste d'attributs.
- ▶ La relation résultat du SELECT-FROM-WHERE est groupée selon les valeurs de tous les attributs et l'opérateur d'agrégation n'est appliqué que dans chaque groupe
- ▶ A partir de Vend(bar, bière, prix) et de Fréquente(client, bar)
- ▶ 

```
SELECT client, AVG(prix)
FROM Fréquente, Vend
WHERE bière = '1664' AND
 Fréquente.bar = Vend.bar
GROUP BY Client;
```



## Groupement

- ▶ Si un opérateur d'agrégation est utilisé, alors chaque élément du SELECT doit être soit agrégé, soit un attribut de la liste GROUP BY.
- ▶ 

```
SELECT bar, MIN(prix)
FROM Vend
WHERE bière = '1664';
```

 est une requête illégale



69/82



70/82

## HAVING

- ▶ HAVING <condition> peut suivre une clause GROUP BY
- ▶ La condition s'applique sur chaque groupe, les groupes qui ne satisfont pas la condition sont éliminés.
- ▶ Trouver le prix moyen des bières qui sont soit servies dans au moins 3 bars, soit qui sont ambrées.
- ▶ 

```
SELECT bière, AVG(prix) FROM Vend
GROUP BY bière
HAVING COUNT(bar) >= 3 OR
 bière IN (SELECT nom
 FROM Bière
 WHERE couleur = 'ambrée');
```



## Conditions sur la clause HAVING

- ▶ Ces conditions peuvent référencer des relations ou des variables nuplets quelconques mentionnées dans la clause FROM
- ▶ Elles peuvent référencer des attributs de ces relations, tant que ces attributs ont un sens dans le groupe, i.e., qu'ils sont un attribut agrégé ou de groupement



71/82



72/82

## Exemples groupement

- ▶ Trouver le prix moyen de la bière 1664:
- ▶ `SELECT AVG(prix) FROM Vend  
WHERE bière = '1664';`
- ▶ Quantité moyenne commandée pour les produits faisant l'objet de plus de 3 commandes
- ▶ `SELECT NumProd, AVG(Quantite) FROM Commande  
GROUP BY NumProd  
HAVING COUNT(*)>3;`
- ▶ Attention : La clause HAVING ne s'utilise qu'avec GROUP BY.



## Exemples Groupement

- ▶ Quantité totale commandée par chaque client
- ▶ `SELECT NumCli, SUM(Quantite) FROM COMMANDE  
GROUP BY NumCli;`
- ▶ Nombre de produits différents commandés...
- ▶ `SELECT NumCli, COUNT(DISTINCT NumProd) FROM COMMANDE  
GROUP BY NumCli;`



73/82



74/82

## La division en SQL

- ▶ Les films qui sont projetés dans tous les cinémas, à partir des relations Projection(NumFilm,NumCinéma) et Cinéma(NumCinéma,Adresse,NuméroTél)
- ▶ `SELECT NumFilm  
FROM (SELECT DISTINCT NumCinema,NumFilm FROM  
Projection) AS proj  
GROUP BY NumFilm  
HAVING count(NumCinema)=(SELECT count(*) FROM  
Cinema) ;`



## La division en SQL

- ▶ `SELECT DISTINCT NumFilm FROM Projection AS LesFilms  
WHERE NOT EXISTS  
(SELECT DISTINCT NumCinema FROM Cinema  
MINUS (SELECT DISTINCT NumCinema FROM Projection  
WHERE NumFilm=LesFilms.NumFilm)) ;`
- ▶ `SELECT DISTINCT NumFilm FROM Projection AS LesFilms  
WHERE NOT EXISTS  
(SELECT NumCinema FROM cinema AS lesCinemas  
WHERE NOT EXISTS  
(SELECT * FROM Projection  
WHERE NumFilm=LesFilms.NumFilm  
AND NumCinema=LesCinemas.NumCinema)) ;`



75/82



76/82

## Vue

- ▶ Une *vue* est une table virtuelle, i.e., une relation définie en termes d'autres tables et vues.
- ▶ Une vue (par défaut) n'est pas stockée dans la base de données (par opposition à une table de base).
- ▶ Un utilisateur peut consulter ou modifier une vue (selon ses droits) comme si c'était une table réelle



## Création et suppression d'une vue

- ▶ Une vue est définie par : `CREATE VIEW <nomvue> AS SELECT ... [WITH CHECK OPTION];`
- ▶ Le select peut contenir toutes les clauses d'un SELECT sauf ORDER by
- ▶ PeutBoire(client, bière) est une vue qui "contient" toutes les paires (client, bière) tel que le client fréquente au moins un bar qui sert la bière en question:  

```
CREATE VIEW PeutBoire AS
SELECT Client, Bière
FROM Fréquente, Vend
WHERE Fréquenté.bar = Vend.bar;
```
- ▶ Suppression d'une vue par `DROP VIEW <nomvue>`



77/82

78/82

## Utilisation d'une vue dans un SELECT

- ▶ On peut requêter une vue comme si c'était une table de base. Permet de donner l'accès à un nombre limité d'attributs pour une table de base.
- ▶ `SELECT beer FROM PeutBoire WHERE drinker = 'Sally';`
- ▶ Que se passe-t-il quand on utilise une vue dans une requête : le SGBD interprète la requête comme si la vue était une table de base. Chaque vue agit comme une macro: elle est remplacée dans la requête évaluée par son équivalent.



## Mises à jour à travers une vue

On peut effectuer des delete/update/insert à travers une vue aux conditions suivantes:

- ▶ une seule table
- ▶ pas de group by
- ▶ pas de distinct
- ▶ pas de fonction de groupe
- ▶ les colonnes modifiées sont les colonnes réelles de la table (update)  
: `update PeutBoire set Client = 'Sal' where Client = 'Sally'`
- ▶ toutes les colonnes not null de la table sur laquelle la vue est posée sont présentes dans la vue (insert)
- ▶ si la vue a été créée avec with CHECK OPTION, toute modification au travers de la vue ne peut concerner que des attributs/nuplets de la vue elle même.



79/82

80/82



## Utilité des vues

- ▶ Les vues permettent de dissocier
  - la façon dont un utilisateur voit les données
  - le découpage en tables
- ▶ On peut par exemple, remplacer une table par deux tables sans modifier le schéma de la BD
- ▶ Les vues permettent de simplifier la consultation de la BD, en “précodant” des selects complexes



81/82



## Utilité des vues

Les vues participent à la protection des données :

- ▶ on peut donner accès à une vue sans donner accès à la table sous-jacente (accès à certaines lignes ou à certaines colonnes de cette table)
- ▶ les modifications peuvent également être restreintes avec la clause `with CHECK OPTION`



82/82

