



Elément : Bases de Données Cycle Ingénieur/ ENSAF 1ère Année V. 2024



abdelhak.boulaalam@usmba.ac.ma

Database Concepts
©Boulaalam, SMBA University
National School of Applied Sciences Fes
<https://sites.google.com/a/usmba.ac.ma/boulaalam/>



Elément 2 : Partie 3
1- Introduction à SQL
- Outline -


Partie 2: Bases de données

- Partie 1: Introduction et rappel modélisation
- Partie 2: Algèbre Relationnelle - Relational Algebra
- Partie 3:** Le langage SQL (norme SQL)
 - TPs: Implémentation sous Oracle 19c

DB & SQL

2

©Boulaalam/SMBA University



1- Introduction à SQL - Outline -

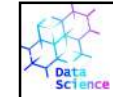
Elément 2: Bases de données

- Introduction à SQL
 - SQL : Introduction et historique
 - SGBDR - RDMS : Architecture
 - Fondements SQL
 - Vue d'ensemble du langage de requête SQL
 - Définition des données SQL
 - Structure de requête de base des requêtes SQL
 - Opérations de base supplémentaires
 - Définir les opérations
 - Valeurs nulles
 - Fonctions d'agrégation
 - Sous-requêtes imbriquées
 - Modification de la base de données

DB & SQL

3

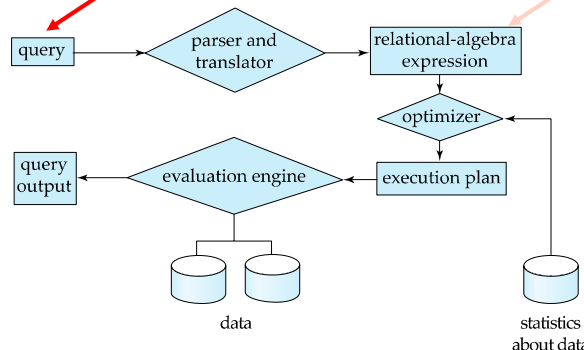
©Boulaalam/SMBA University



Partie 3: SQL query

- Pourquoi étudier l'algèbre relationnelle??
- Comment les SGBDs Traitent les requêtes ??
- SQL !!

Ns Somme ici
Partie 3



Ns Somme ici
Partie 2

DB & SQL

4

©Boulaalam/SMBA University



Introduction à SQL: History



Introduction à SQL History - Introduction to SQL

SQL- Historique

- Modèle Relationnel → proposé par E. F. Codd en 1970.
- D. D. Chamberlin et d'autres du IBM → ont développé le langage appelé SQL:
 - Structured Query Language,
 - Comme sous-langage **de données** pour le modèle relationnel (langage de bas niveau)
- Initialement orthographié SEQUEL:
 - le langage a été décrit dans une série d'articles à partir de **1974**,
 - et il a été utilisé dans un prototype de système relationnel appelé System R, de IBM fin des années **1970**.



Introduction à SQL History - Introduction to SQL

SQL- Historique

- **Ingres**, un autre prototype de système de gestion de bases de données relationnelles, a été développé à l'Université de Californie.
- Le système R a été évalué et affiné sur une période de plusieurs années, et il est devenu la **base du premier système de gestion de base de données relationnelle** d'IBM, SQL / DS, qui a été annoncé en **1981**.
- Le DB2 d'IBM, utilisant également SQL comme langage, a été publié en **1983**.
- Microsoft SQL Server, MySQL, Informix, Sybase, PostgreSQL, Microsoft Access et de nombreux autres systèmes de gestion de bases de données relationnelles ont également **incorporé SQL**.



Introduction à SQL History - Introduction to SQL

SQL- Historique

- **L'ANSI** (American National Standards Institute) et **l'ISO** (Organisation internationale de normalisation) **ont adopté SQL comme langage standard pour les bases de données relationnelles** et publié des spécifications pour le langage SQL à partir de **1986**.
 - Cette norme est généralement appelée **SQL1** ou **SQL-86**
 - Une révision mineure, appelée **SQL-89**, a été publiée trois ans plus tard.

<https://www.ansi.org/>

<https://www.iso.org/home.html>



Introduction à SQL History - Introduction to SQL

SQL- Historique

- Une révision majeure, **SQL2**, (SQL-92) a été adoptée par l'ANSI et l'ISO en **1992**.
- La norme **SQL3** a été développée au fil du temps, avec des parties majeures publiées en **1999, 2003, 2006 et 2008**.
 - Les nouvelles fonctionnalités comprenaient des capacités de gestion de **données orientées objet**, des **déclencheurs**, **nouveaux types** de données, prise en charge de **XML** et autres fonctionnalités.
- La prise en charge des bases de données temporelles a été ajoutée à la **norme en 2011**.
- La dernière version est **SQL:2016**



Introduction à SQL History - Introduction to SQL

SQL- Historique

- La plupart des fournisseurs de SGBD relationnelles ont **ajouté des fonctionnalités à la norme**:
 - Ils utilisent leurs propres extensions du langage, créant une variété de dialectes autour de la norme.
- Étant donné que certaines parties de la norme actuelle sont facultatives, tous les fournisseurs **ne fournissent pas l'ensemble complet des fonctionnalités décrites dans la norme**.
- Cette partie 3 de ce cours se concentrera sur les fonctionnalités **strictement relationnelles** les **plus utilisées** qui sont disponibles dans la plupart des SGBD **relationnels**.



Les 2 grandes parties du SQL: SQL DDL SQL DML



Introduction à SQL SQL Parts

Les Parties SQL

- **SQL DML**
 - Offre la possibilité:
 - **d'interroger** des informations de la base de données,
 - **d'insérer** des tuples,
 - de **supprimer** des tuples,
 - de **modifier** des tuples dans la base de données.



Introduction à SQL SQL Parts

Les Parties SQL

- **SQL DDL** (voir TP1)
 - Inclut des commandes pour **spécifier les contraintes d'intégrité**.
 - Définition de vue:
 - Le DDL inclut des commandes pour définir des vues.
 - Le langage de définition de données SQL DDL permet de spécifier des informations sur les relations (tables), notamment :
 - Le **schéma** de chaque relation.
 - Le **type** de valeurs associées à chaque attribut.
 - Les **contraintes d'intégrité**
 - L'ensemble des **indices** à maintenir pour chaque relation.
 - Informations de **sécurité et d'autorisation** pour chaque relation.
 - La structure de **stockage physique** de chaque relation sur disque.



Introduction à SQL SQL Parts

Les Parties SQL

- **Autres parties SQL**
 - **Contrôle des transactions:**
 - Comprend des commandes permettant de spécifier le début et la fin des transactions.
 - **Embedded SQL et SQL dynamique:**
 - Définissent comment les instructions SQL peuvent être intégrées dans des langages de programmation à usage général.
 - **Autorisation:**
 - Inclut des commandes pour spécifier les droits d'accès aux relations et aux vues.



Introduction à SQL SQL Parts

Note du cours

- Les commandes données dans ce support de cours utilisent généralement la syntaxe **Oracle**.
 - Elles peuvent nécessiter de légères modifications pour s'exécuter sur d'autres SGBD.



SGBDR - RDMS : Architecture Un aperçu



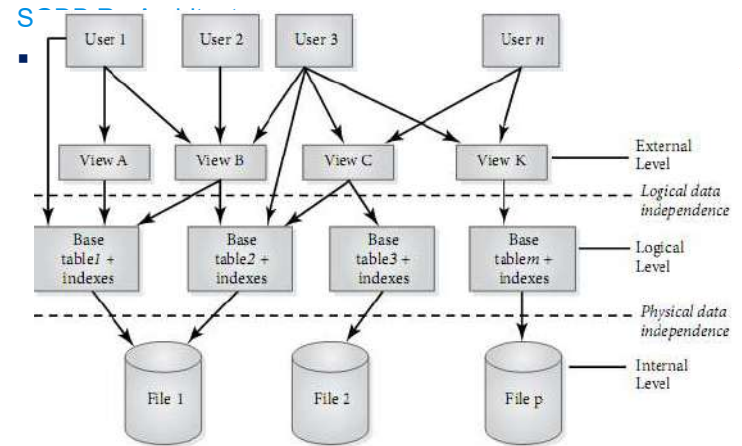
Architecture d'un SGDB R

SGDB R : Architecture

- Ils prennent en charge l'**architecture standard à trois niveaux** pour les BDs.
 - Voir les parties 1 et 2 CM
 - Voir aussi l'élément 1 de ce cours



Architecture d'un SGDB R



Architecture d'un SGDB R

SGDB R : Architecture

- les BDs relationnelles fournissent à la fois l'**indépendance des données logiques et physiques**:
 - Séparer les niveaux externe, logique et interne.
 - Le niveau logique des bases de données relationnelles:
 - se compose de tables de base qui sont physiquement stockées.
 - Ces tables sont créées par l'administrateur de la base de données à l'aide de la commande SQL DDL:
 - **CREATE TABLE** tableNom



Architecture d'un SGDB R

SGDB R : Architecture

- Une table de base peut avoir n'importe quel nombre d'index:
 - soit créé par le système lui-même,
 - soit créé par le DBA à l'aide de la commande SQL DDL **CREATE INDEX...**
- Un index est utilisé pour accélérer la récupération des enregistrements en fonction de la valeur dans une ou plusieurs colonnes.
- Un index répertorie les valeurs qui existent pour les colonnes indexées et donne l'emplacement des enregistrements qui ont ces valeurs.
- La plupart des SGDB relationnelles utilisent des arborescences B ou B+ pour les index.



Architecture d'un SGDB R

SGDB R : Architecture

- Au niveau physique:
 - les tables de base et leurs index sont représentés dans des fichiers.
- Remarques:
 - La **représentation physique des tables** peut ne pas correspondre exactement à notre notion d'une table de base en tant **qu'objet bidimensionnel composé de lignes et de colonnes** comme une feuille de calcul.
 - Cependant, les lignes de la table correspondent à des enregistrements stockés physiquement, bien que leur ordre et d'autres détails de stockage puissent être différents de notre concept de table.



Architecture d'un SGDB R

SGDB R : Architecture

- Le SGBD, et **non le système d'exploitation**, contrôle la structure interne des fichiers de données et des index.
 - L'utilisateur ne sait généralement pas quels index existent et n'a aucun contrôle sur quel index sera utilisé pour localiser un enregistrement.
- Une fois les tables de base créées, le **DBA** peut **créer des vues pour les utilisateurs**:
 - à l'aide de la commande SQL **CREATE VIEW ...**



Architecture d'un SGDB R

SGDB R : Architecture

- les **vues relationnelles** peuvent être:
 - des fenêtres dans des tables de base ou des tables virtuelles, **non stockées en permanence** mais créées lorsque l'utilisateur **a besoin** d'y accéder.
- Les utilisateurs ignorent que leurs vues ne sont pas physiquement stockées sous forme de table.
- Dans un système relationnel:
 - la vue signifie une seule table, vue par un utilisateur.



Architecture d'un SGDB R

SGDB R : Architecture

- les **vues relationnelles**
 - Ce n'est pas exactement la même chose que notre terme vue externe:
 - ce qui signifie la base de données telle qu'elle apparaît à un utilisateur particulier.
 - Dans notre terminologie:
 - Une vue externe peut être constituée de **plusieurs tables de base et / ou vues**.



Architecture d'un SGDB R

SGDB R : Architecture

- L'une des caractéristiques les plus utiles d'une base de données relationnelle est qu'elle permet une définition de **base de données dynamique**.
- L'**administrateur** de base de données et les **utilisateurs** qu'il autorise à le faire peuvent:
 - Créer de nouvelles tables,
 - Ajouter des colonnes aux anciennes,
 - Créer de nouveaux index,
 - Définir des vues et supprimer n'importe lequel de ces objets à tout moment.
- La flexibilité des bases de données relationnelles encourage les utilisateurs à expérimenter différentes structures et permet de modifier le système pour répondre à leurs besoins changeants.
- Cela permet au DBA de s'assurer que la base de données est un modèle utile de l'entreprise tout au long de son cycle de vie.



SQL DDL - Data Definition Language



SQL DDL – Data Definition Language

SQL DDL (rappel)

- SQL (DDL) : Le langage de **définition de données**
 - Il permet de spécifier des informations sur les relations, notamment:
 - Le schéma de chaque relation.
 - Le type de valeurs associées à chaque attribut.
 - Les contraintes d'intégrité
 - L'ensemble des indices à maintenir pour chaque relation.
 - Informations de sécurité et d'autorisation pour chaque relation.
 - La structure de stockage physique de chaque relation sur le disque.



SQL DDL – Data Definition Language

SQL DDL

- **Types de base (atomique) par le standard SQL**
 - **char(n):**
 - une chaîne de caractères de longueur fixe n.
 - **varchar(n):**
 - une chaîne de caractères de longueur variable avec une longueur maximale n.
 - **int:**
 - un entier (un sous-ensemble fini d'entiers qui dépend de la machine).
 - **smallint:**
 - un petit entier (un sous-ensemble dépendant de la machine de type entier).



SQL DDL – Data Definition Language

SQL DDL

■ Types de base (atomique) par le standard SQL

- **numeric(p, d)** : Un nombre à virgule fixe avec une précision spécifiée par l'utilisateur.
 - Le nombre est composé de p chiffres (plus un signe), et d des p chiffres se trouvent à droite de la virgule décimale.
 - Exemple: **numeric(3,1)**
 - permet de stocker exactement 44,5, mais ni 444,5 ni 0,32 ne peuvent être stockés exactement dans un domaine de ce type.
- **real, double precision**: nombres à virgule flottante et double précision à virgule flottante avec précision dépendante de la machine.
- **float(n)** : nombre à virgule flottante avec une précision d'au moins n chiffres.



SQL DDL – Data Definition Language

SQL DDL

■ CREATE TABLE ... – Syntaxe

```
CREATE TABLE r
(A1 D1,
 A2 D2,
 ...,
 An Dn,
 <integrity-constrainti>,
 ...,
 <integrity-constraintk>);
```

```
CREATE TABLE instructor (
    ID      char(5),
    name    varchar(20),
    dept_name varchar(20),
    salary  numeric(8,2))
```

```
CREATE TABLE department
(dept_name varchar(20),
 building  varchar(15),
 budget   numeric(12,2),
 primary key (dept_name));
```



SQL DDL – Data Definition Language

SQL DDL

■ CREATE TABLE ...

- Utilisée pour créer les tables.
- Utilisé à tout moment pendant le cycle de vie du système:
 - le développeur de la base de données peut commencer avec un petit nombre de tables et les ajouter à mesure que des applications supplémentaires.
- Comme dans le modèle relationnel abstrait, les lignes sont considérées comme **non ordonnées**.
- Cependant, les colonnes **sont ordonnées** de gauche à droite, pour correspondre à l'ordre des définitions de colonnes dans la commande **CREATE TABLE**.



SQL DDL – Data Definition Language

SQL DDL

■ CREATE TABLE ...

- Exemple SQL DDL pour le schéma de base de données university (Voir TP1)



SQL DDL – Data Definition Language

SQL DDL

- **CREATE TABLE ... et contraintes d'intégrité**
 - Types de contraintes d'intégrité:
 - **primary key** (A_1, \dots, A_n)
 - **foreign key** (A_m, \dots, A_n) **references** r
 - **not null**
 - SQL **empêche** toute mise à jour de la base de données qui **viole** une contrainte d'intégrité!!
 - Exemple:

```
CREATE TABLE instructor (  
    ID          char(5),  
    name        varchar(20) NOT NULL,  
    dept_name   varchar(20),  
    salary       numeric(8,2),  
    PRIMARY KEY (ID),  
    FOREIGN KEY (dept_name) REFERENCES department);
```



SQL DDL – Data Definition Language

SQL DDL

- **CREATE TABLE ... et contraintes d'intégrité**
 - Types de contraintes d'intégrité:
 - **PRIMARY KEY** (A_1, A_2, \dots, A_m):
 - les attributs A_1, A_2, \dots, A_m forment la clé primaire de la relation.
 - Rappel: Les attributs de clé primaire doivent être **non nuls** et **uniques** (not null and unique);
 - **FOREIGN KEY** (A_k1, A_k2, \dots, A_kn) **REFERENCES** s
 - les valeurs des attributs (A_k1, A_k2, \dots, A_kn) pour tout tuple dans la relation **doit correspondre** aux valeurs des attributs de clé primaire de certains tuple dans la relation s .



SQL DDL – Data Definition Language

SQL DDL

- **CREATE TABLE ... et contraintes d'intégrité**
 - Exercice/Questions:
 - Dans la table course :
FOREIGN KEY (dept_name) **REFERENCES** department
 - pour chaque tuple de course, le nom de département spécifié dans le tuple **doit exister** dans l'attribut de clé primaire (dept_name) de la relation de département.
 - Q1: Si cette contrainte n'existe pas! Résultat?
 - un cours dans un département inexistant.
 - Q2: Discuter les autres contraintes PK FK de l'exemple.
 - Remarque:
 - Certains SGBD, y compris MySQL, nécessitent une syntaxe alternative:
FOREIGN KEY (dept_name) **REFERENCES** department (**dept_name**)
 - où les attributs référencés dans le référencé sont répertoriés explicitement



SQL DDL – Data Definition Language

SQL DDL

- **CREATE TABLE ... et contraintes d'intégrité**
 - **not null**
 - Cette contrainte pour un attribut spécifie que la valeur nulle n'est pas autorisée pour cet attribut
 - la contrainte exclut la valeur nulle du domaine de cet attribut.



SQL DDL – Data Definition Language

SQL DDL

- **Contraintes d'intégrité et violation**
 - Important:
 - SQL empêche toute mise à jour de la base de données **qui viole** une contrainte d'intégrité.
 - SQL signale une erreur et empêche la mise à jour.
 - Q3: Donner quelques exemples de violation des contraintes d'intégrité pour la BD university.
 - Remarques:
 - Une relation nouvellement créée est **initialement vide**.
 - L'**insertion**, la **mise à jour** et la **suppression** des tuples dans une relation, leur mise à jour et leur suppression sont effectuées par les instructions de manipulation de données du SQL DML: **INSERT, UPDATE ET DELETE**



SQL DDL – Data Definition Language

DROP et DELETE

- Pour supprimer une relation/table r d'une base de données SQL:
 - DROP TABLE r;**
 - supprime toutes les informations sur la relation r de la BD.
 - Elle supprime tous les **tuples** de r et le **schéma** de r.
 - C'est une action plus drastique que **DELETE FROM r;**
 - DELETE FROM r;**
 - Elle conserve la relation r, mais supprime tous les tuples de la table r (vider la table r).



SQL DDL – Data Definition Language

ALTER TABLE

- Utiliser pour:
 - **Renommer** une table
 - **Ajouter** des attributs (colonnes) à une table
 - **Supprimer** un attribut
 - **Augmenter** la largeur de la colonne
- Ces opérations sont faites sur une relation existante.
 - Tous les tuples de la relation ont la **valeur null** comme valeur pour le nouveau attribut.
- Exemples:
 - **ALTER TABLE r ADD att D;**
 - Att: nom de l'attribut à ajouter
 - D: le type de l'attribut ajouté.
 - **ALTER TABLE r DROP COLUMN A;**
 - Pour supprimer des attributs d'une relation.
 - **ALTER TABLE emp DROP COLUMN gst;**



SQL DDL – Data Definition Language

ALTER TABLE

- Exemples:
 - Pour modifier les propriétés des colonnes d'une table, en **changeant le type** de données, la **taille**, la **valeur par défaut** ou **les contraintes**, à l'aide de la commande :
 - **ALTER TABLE r MODIFY COLUMN Att newSpecifications;**
 - **newSpecifications** de la colonne utilisent le même libellé que dans une commande CREATE TABLE.
 - Le type de données ne peut être modifié que si toutes les lignes de la table contiennent des valeurs nulles pour la colonne en cours de modification.
 - La taille ou la précision peut toujours être augmentée même si des valeurs non nulles sont stockées,
 - mais elles ne peuvent être diminuées que si aucune ligne existante ne viole les nouvelles spécifications.



SQL DDL – Data Definition Language (fin)

Ressources

- **SQL | DDL, DQL, DML, DCL and TCL Commands**, <https://www.geeksforgeeks.org/sql-ddl-dql-dml-dcl-tcl-commands/>
- SO/IEC 9075-1:2016, Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework), <https://www.iso.org/standard/63555.html>
- **SQL Language Reference**, Oracle® Database SQL Language Reference 19c, E96310-20, January 2023, <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/>
- Cours et tutoriels sur le langage SQL, <https://sql.sh/>



Introduction to SQL: SQL DML

Manipulation de la base de données : SQL DML

- Le langage de requête SQL est **déclaratif**, ou **non-procédural**:
 - ce qui signifie qu'il nous permet de spécifier quelles données doivent être récupérées **sans donner les procédures pour les récupérer**.
 - Il peut être utilisé comme langage interactif pour les requêtes, intégré dans un langage de programmation hôte.
- Les instructions SQL DML sont :
 - SELECT
 - UPDATE
 - INSERT
 - DELETE



Introduction à SQL Introduction to SQL: SQL DML

Manipulation de la base de données : SQL DML

- SQL query
 - La structure de base d'une requête SQL se compose de trois clauses: **SELECT**, **FROM** et **WHERE**.
- Une requête prend en **entrée**:
 - les relations répertoriées dans la clause FROM,
 - les opérateurs comme spécifié dans les clauses WHERE et SELECT,
- Puis **produit** une relation comme résultat.
- **Syntaxe**:

```
SELECT  A1, A2, ..., An
FROM    r1, r2, ..., rm
WHERE   Predicates;
```



Introduction à SQL Introduction to SQL: SQL DML

Manipulation de la base de données : SQL DML

- **Requêtes sur une seule relation**
 - **Requête 1**: "Trouvez les noms de tous les instructeurs."
SELECT name FROM instructor;
 - **Requête 2**: «Trouver les noms de département de tous les instructeurs»
SELECT dept_name FROM instructor;
 - un nom de département peut apparaître plusieurs fois.
 - Dans la définition formelle et mathématique du modèle relationnel, une relation est **un ensemble**.
 - Ainsi, les tuples en double n'apparaîtraient jamais dans les relations.
 - Pour **forcer l'élimination** des doublons: mot-clé distinct
SELECT DISTINCT dept_name FROM instructor;



Introduction à SQL Introduction to SQL: SQL DML

Manipulation de la base de données : SQL DML

Requêtes sur une seule relation

- mot clé **ALL**:
 - pour spécifier explicitement que les **doublons ne sont pas supprimés**:

```
SELECT ALL dept_name FROM instructor;
```

- La clause SELECT peut également contenir des **expressions arithmétiques** impliquant les opérateurs +, -, * et / opérant sur des **constantes** ou des **attributs** de tuples. Par exemple:

- Requête 3:


```
SELECT ID, name, dept_name, salary * 1.1 FROM instructor;
```

 - 10% d'augmentation



Introduction à SQL Introduction to SQL: SQL DML

Manipulation de la base de données : SQL DML

Requêtes sur une seule relation (fin)

- La clause WHERE pour satisfaire un **prédicat spécifié**.

- Requête 4: "Trouvez les noms de tous les instructeurs du département informatique qui ont un salaire supérieur à 70 000 "

```
SELECT      name
FROM        instructor
WHERE       dept_name = 'Comp. Sci.' AND salary > 70000;
```

- SQL permet l'utilisation des connecteurs logiques **AND**, **OR**, et **NOT** dans la clause where.
- Les opérandes des connecteurs logiques peuvent être des expressions impliquant les opérateurs de comparaison <, <=, >, >=, = et <>.
- SQL nous permet d'utiliser les opérateurs de comparaison pour comparer des chaînes et des expressions arithmétiques, ainsi que des types spéciaux, tels que les types DATE.



Introduction à SQL Introduction to SQL: SQL DML

Manipulation de la base de données : SQL DML

Requêtes sur les relations multiples

- Requête 5: "Récupérer les noms de tous les instructeurs, ainsi que leurs noms de département et le nom du bâtiment de département."

- En SQL, pour répondre à des requêtes de ce genre:
 - Répertorier les relations auxquelles il faut accéder dans la clause **FROM** et
 - Spécifier la condition de correspondance dans la clause **WHERE**. (jointure AR)

```
SELECT name, instructor.dept_name, building
FROM   instructor, department
WHERE  instructor.dept_name = department.dept_name;
```

- Notez que l'attribut dept_name existe dans les 2 relations.
- Utiliser **le nom de relation** comme un **préfixe** (dans le nom de l'instructeur.dept_name et department.dept_name) pour indiquer clairement à quel attribut nous nous référons:
 - **enlever l'ambiguïté**



Introduction à SQL Introduction to SQL: SQL DML

Manipulation de la base de données : SQL DML

Requêtes sur les relations multiples

- Requête 6:
 - "Trouver uniquement les noms des instructeurs et les identifiants des cours enseignés pour les instructeurs du département informatique."

```
SELECT name, course_id
FROM   instructor, teaches
WHERE  instructor.ID = teaches.ID
AND    instructor.dept_name = 'Comp. Sci.';
```



Requêtes SQL: structure de base SQL Queries

Requêtes sur les relations multiples

- En général, pour un SGBD, la signification d'une requête SQL peut être comprise comme suit:
 1. Générez un produit cartésien des relations répertoriées dans la clause from.
 2. Appliquez les prédicats spécifiés dans la clause where sur le résultat de l'étape 1.
 3. Pour chaque tuple du résultat de l'étape 2, sortez les attributs (ou résultats d'expressions) spécifiés dans la clause select.



Requêtes SQL: structure de base SQL Queries

Requêtes sur les relations multiples

- Cette séquence d'étapes permet de **clarifier le résultat** d'une requête SQL et non **la manière dont elle doit être exécutée**.
- Une implémentation réelle de SQL n'exécuterait pas la requête de cette manière; il optimiserait plutôt l'évaluation en ne générant (autant que possible) que des éléments du produit cartésien qui satisfont aux prédicats de la clause where.
- Lors de l'écriture de requêtes, vous devez veiller à inclure les conditions de clause where appropriées.
- Si requête 5 sans condition de clause where:
 - elle produira le produit cartésien, qui pourrait être une relation énorme.
 - Pour l'exemple leur produit cartésien a $12 \times 13 = 156$.
 - Pour aggraver les choses, supposons que nous ayons 200 instructeurs. Supposons que chaque instructeur enseigne trois cours, nous avons donc 600 tuples dans la relation teaches.
 - Ensuite, le processus itératif précédent génère $200 * 600 = 120\,000$ tuples dans le résultat.



Opérations de base supplémentaires

- Un certain nombre d'opérations de base supplémentaires sont prises en charge dans la norme SQL.
- **Opération Rename**
 - Considérez à nouveau la requête 5:

```
select    name, course_id
from      instructor, teaches
where     instructor.ID= teaches.ID;
```

- Le résultat est une table avec les att: name et course_id



Opérations de base supplémentaires

- Un certain nombre d'opérations de base supplémentaires sont prises en charge dans la norme SQL.
- **Opération Rename**
 - Discussion:
 1. deux relations dans la clause from peuvent avoir des attributs avec le même nom, auquel cas un nom d'attribut est dupliqué dans le résultat.
 2. si nous utilisons une expression arithmétique dans la clause select, l'attribut résultant n'a pas de nom.
 3. même si un nom d'attribut peut être dérivé des relations de base comme dans l'exemple, nous pouvons vouloir changer le nom d'attribut dans le résultat.
 - Par conséquent, SQL fournit un moyen de renommer les attributs d'une relation de résultat. Il utilise la clause as:

old-name **AS** new-name



Opérations de base supplémentaires

- Un certain nombre d'opérations de base supplémentaires sont prises en charge dans la norme SQL.
- Opération Rename**
 - as clause peut être utilisée dans les clauses select et from

```
select name as instructor_name, course_id
from instructor, teaches
where instructor.ID= teaches.ID;
```

```
select T.name, S.course_id
from instructor as T, teaches as S
where T.ID= S.ID;
```

→ tous les instructeurs de l'université qui ont enseigné un cours, trouvez leur nom et l'ID du cours de tous les cours qu'ils ont enseignés



Opérations de base supplémentaires

- Un certain nombre d'opérations de base supplémentaires sont prises en charge dans la norme SQL.
- Opération Rename**
 - Une autre raison de renommer une relation est un cas où nous souhaitons comparer des tuples dans la même relation.
 - Il nous faut alors prendre le produit cartésien d'une relation avec lui-même.
 - Requête 6: "Trouvez les noms de tous les instructeurs dont le salaire est supérieur à au moins un instructeur du département de biologie".

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept_name =
'Biologie';
```



Opérations de base supplémentaires

- Un certain nombre d'opérations de base supplémentaires sont prises en charge dans la norme SQL.
- Opération Rename** - Requête 6: discussion
 - nous ne pouvons pas utiliser la notation instructor.salary
 - car il ne serait pas clair quelle référence est destinée.
 - T et S peuvent être considérés comme des copies.
 - plus précisément, ils sont déclarés comme des alias,
 - c'est-à-dire comme des noms alternatifs utilisés pour renommer une relation est appelé un nom de corrélation (correlation name) dans le standard SQL,
 - communément appelé un alias de table, ou une variable de corrélation, ou une variable de tuple (table alias, correlation variable, tuple variable)



Opérations de base supplémentaires

- String Operations**
 - SQL spécifie des chaînes en les mettant entre guillemets simples, par exemple, 'Computer'.
 - Un seul guillemet qui fait partie d'une chaîne peut être spécifié en utilisant deux guillemets simples;
 - Par exemple, la chaîne "It's right" peut être spécifiée par 'It"s right'.
 - Le standard SQL spécifie que l'opération d'égalité sur les chaînes est sensible à la casse;
 - par conséquent, l'expression «'comp. sci.' = 'Comp. Sci.' » Est évalué à faux.
 - Cependant, certains systèmes de base de données, tels que MySQL et SQL Server, ne distinguent pas les majuscules des minuscules lors de la correspondance des chaînes;



Opérations de base supplémentaires

String Operations

- SQL permet également une variété de fonctions sur les chaînes de caractères:
 - la concaténation (en utilisant «||»),
 - l'extraction des sous-chaînes,
 - la recherche de la longueur des chaînes,
 - la conversion des chaînes en majuscules et minuscule,
 - en supprimant les espaces à la fin de la chaîne.
- Il existe des variations sur l'ensemble exact des fonctions de chaîne prises en charge par différents SGBD.
- TAR: Donner et faire une comparaison entre les différentes fonctions pour manipuler les string pour SBD suivants: Oracle, MySQL et PostgreSQL. (par email avant dimanche)



Opérations de base supplémentaires

String Operations

- La correspondance de modèles peut être effectuée sur des chaînes en utilisant l'opérateur **LIKE**
- Deux caractères spéciaux:
 - Pourcentage (%): le caractère% correspond à **n'importe quelle sous-chaîne**.
 - Underscore (_): le caractère correspond à **n'importe quel caractère**.



Opérations de base supplémentaires

String Operations

- Exemples
 - Les modèles sont sensibles à la casse
 - Sauf pour MySQL, ou avec l'opérateur **ILIKE** dans PostgreSQL
 - 'Intro%' correspond à toute chaîne commençant par "Intro".
 - '% Comp%' correspond à toute chaîne contenant "Comp" comme sous-chaîne □ 'Intro. to Computer Science', et 'Computational Biology'
 - '___' correspond à n'importe quelle chaîne de trois caractères exactement.
 - '___%' correspond à toute chaîne d'au moins trois caractères.



Opérations de base supplémentaires

String Operations

- Requête 7: «Trouvez les noms de tous les départements dont le nom du bâtiment inclut la sous-chaîne 'Watson'.

```
select dept_name
from department
where building like '%Watson%';
```

- barre oblique inverse (\) :
- Caractère d'échappement (escape character)
- utilisé juste avant un caractère de modèle spécial pour indiquer que le caractère de modèle spécial doit être traité comme un caractère normal.
- like 'ab\%cd%' escape '\': tout strings commence par "ab%cd".
- like 'ab\\cd%' escape '\\': tout strings commence par "ab\cd".



Opérations de base supplémentaires

■ String Operations

• NOT LIKE

- SQL nous permet de rechercher des incompatibilités au lieu de correspondances en utilisant l'opérateur de comparaison **NOT LIKE**.
- Certaines implémentations fournissent des variantes de l'opération similaire qui ne distinguent pas les minuscules et les majuscules.



Opérations de base supplémentaires

■ Symbole *

- dans la clause Select, Le symbole astérisque «*» peut être utilisé pour désigner «tous les attributs».

```
select instructor.*  
from instructor, teaches  
where instructor.ID= teaches.ID;
```

- tous les attributs de l'instructeur doivent être sélectionnés.

```
select *  
from instructor, teaches  
where instructor.ID= teaches.ID;
```

- indique que tous les attributs de la relation résultante de la clause from sont sélectionnés.



Opérations de base supplémentaires

■ Ordonner l'affichage des tuples

- SQL offre à l'utilisateur un certain contrôle sur l'ordre dans lequel les tuples d'une relation sont affichés.
- clause **ORDER BY**:
 - fait apparaître les tuples dans le résultat d'une requête dans l'ordre trié.
- Requête 8: « Pour lister par ordre alphabétique tous les instructeurs du département de physique »

```
select    name  
from      instructor  
where     dept_name = 'Physics'  
ORDER BY name;
```



Opérations de base supplémentaires

■ Ordonner l'affichage des tuples

- **ORDER BY** par défaut répertorie les éléments dans l'ordre croissant.
- Pour spécifier l'ordre de tri: nous pouvons spécifier:
 - **DESC** pour l'ordre décroissant
 - **ASC** pour l'ordre croissant.
- **ORDER BY** peut être effectuée sur plusieurs attributs.
- Requête 9: « répertorier la relation instructor par ordre décroissant de salaire. Si plusieurs instructeurs ont le même salaire, nous les classons par ordre croissant de nom ».

```
SELECT    *  
FROM      instructor  
ORDER BY  salary desc, name asc;
```




Opérations de base supplémentaires

- Les prédicats dans la clause where - Clause WHERE : predicates
 - BETWEEN**: opérateur de comparaison de la norme SQL.
 - Simplifier les clauses where qui spécifient qu'une valeur doit être inférieure ou égale à une certaine valeur et supérieure ou égale à une autre valeur.
 - Requête 10: trouver les noms des instructeurs dont le salaire se situe entre 90 000 et 100 000

```
select name
from instructor
where salary between 90000 and 100000;
```

```
select name
from instructor
where salary <= 100000
and salary >= 90000;
```

- NOT BETWEEN**: De même, nous pouvons utiliser cet opérateur pour la comparaison.



Opérations de base supplémentaires

- Les prédicats dans la clause where - Clause WHERE : predicates
 - Le standard SQL nous permet d'utiliser la notation (v1, v2,..., vn) pour dénoter un tuple contenant les valeurs v1, v2,..., vn;
 - la notation est appelée constructeur de lignes.
 - Les opérateurs de comparaison peuvent être utilisés sur des tuples, et l'ordre est défini lexicographiquement.
 - Par exemple, (a1, a2) <= (b1, b2) est vrai si a1 <= b1 et a2 <= b2;
 - Exemple requête SQL: (les deux possibilités)

```
select name, course_id
from instructor, teaches
where instructor.ID= teaches.ID and dept_name = 'Biology';
```

```
select name, course_id
from instructor, teaches
where (instructor.ID, dept_name)= (teaches.ID, 'Biology');
```



Opérations de base supplémentaires

- UNION, INTERSECT et EXCEPT**
 - Ces opérations opèrent sur des relations et correspondent aux opérations de l'ensemble mathématique \cup , \cap et $-$.
 - Exemple

L'ensemble de tous les cours enseignés au semestre d'automne 2017=C1

```
select course_id
from section
where semester = 'Fall' and year= 2017;
```

course_id
CS-101
CS-347
PHY-101

L'ensemble de tous les cours enseignés au semestre de printemps 2018 = C2

```
select course_id
from section
where semester = 'Spring' and year= 2018;
```

course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199



Opérations de base supplémentaires

- UNION, INTERSECT et EXCEPT**
 - Trouver l'ensemble de tous les cours enseignés à l'automne 2017 ou au printemps 2018, ou les deux.
 - les parenthèses autour de chaque instruction de sélection sont facultatives → pour faciliter la lecture;
 - certaines SGDB ne permettent pas l'utilisation des parenthèses.

```
(select course_id from section where semester = 'Fall' and
year= 2017)
union
(select course_id from section where semester = 'Spring' and
year= 2018);
```

- L'opération d'union élimine automatiquement les doublons (contrairement à la clause select). CS-101 et CS-319 n'apparaît qu'une seule fois dans le résultat

c1 union c2 →

course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101



Opérations de base supplémentaires

■ UNION, INTERSECT et EXCEPT

- Pour trouver l'ensemble de tous les cours enseignés à l'automne 2017 et au printemps 2018

```
(select course_id from section where semester = 'Fall'
and year= 2017)
intersect
(select course_id from section where semester = 'Spring'
and year= 2018);
```

- Résultat: ne contient qu'un seul tuple avec CS-101.

- L'opération d'intersection **élimine automatiquement les doublons.**

c1 **intersect** c2 →

course_id
CS-101

NB. MySQL n'implémente pas l'opération d'intersection



Opérations de base supplémentaires

■ UNION, INTERSECT et EXCEPT

- INTERSECT ALL**: Si nous voulons conserver tous les doublons

```
(select course_id from section where semester = 'Fall'
and year= 2017)
Intersect all
(select course_id from section where semester = 'Spring'
and year= 2018);
```

- Le nombre de tuples en double qui apparaissent dans le résultat est égal au nombre minimum de doublons dans c1 et c2.
- Par exemple, si quatre sections de ECE-101 ont été enseignées au semestre d'automne 2017 et deux sections de ECE-101 ont été enseignées au semestre de printemps 2018, alors il y aurait deux tuples avec ECE-101 dans le résultat.



Opérations de base supplémentaires

■ UNION, INTERSECT et EXCEPT

- Pour trouver tous les cours enseignés au semestre d'automne 2017 mais pas au semestre de printemps 2018

```
(select course_id from section where semester = 'Fall'
and year= 2017)
except
(select course_id from section where semester = 'Spring'
and year= 2018);
```

- except sort tous les tuples de sa première entrée qui ne se produisent pas dans la deuxième entrée. Elle élimine automatiquement les doublons dans les entrées avant d'effectuer la différence définie.

c1 **except** c2 →

course_id
CS-347
PHY-101

- except all : conserver les doublons,



Opérations de base supplémentaires

■ Null Values

- Les valeurs nulles posent des problèmes particuliers dans les opérations relationnelles:
 - notamment les opérations arithmétiques, les opérations de comparaison et les opérations d'ensemble.
- Le résultat d'une expression arithmétique (+, -, * ou /) est null si l'une des valeurs d'entrée est nulle → r.A + 5 et que r.A est null pour un tuple particulier, le résultat de l'expression doit également être nul.
- Les comparaisons: «1 < null».
 - Il serait faux de dire que cela est vrai car nous ne savons pas ce que représente la valeur nulle
 - SQL traite donc comme inconnu **-unknown-** le résultat de toute comparaison impliquant une valeur nulle



Opérations de base supplémentaires

Null Values

- Comme le prédicat dans une clause where peut impliquer des opérations booléennes AND, OR, et NOT sur les résultats de comparaisons, les définitions des opérations booléennes sont étendues pour traiter la valeur **unknown**.
 - and
 - true and unknown = unknown
 - false and unknown = false
 - unknown and unknown = unknown.
 - or
 - true or unknown = true
 - false or unknown = unknown
 - unknown or unknown = unknown.
 - not
 - not unknown = unknown.



Opérations de base supplémentaires

Null Values

- SQL utilise le mot clé spécial **null** dans un prédicat pour tester une valeur nulle.
- Exemple: Trouver tous les instructeurs qui apparaissent dans la relation d'instructeur avec des valeurs nulles pour le salaire.

```
select name
from instructor
where salary IS NULL;
```



Opérations de base supplémentaires

Null Values

- Le prédicat **IS NOT NULL** réussit si la valeur à laquelle il est appliqué est not null.
- SQL nous permet de tester si le résultat d'une comparaison est **inconnu**, plutôt que vrai ou faux, en utilisant les clauses **IS UNKNOWN** et **IS NOT UNKNOWN**.

```
select name
from instructor
where salary > 10000 is unknown;
```



Fonctions d'agrégation

Aggregate Functions

- Les fonctions d'agrégation: prennent une **collection** (un ensemble ou plusieurs ensembles) de **valeurs** en entrée et retournent une **valeur unique**.
- Le standard SQL propose 5 fonctions d'agrégation intégrées:
 - Moyenne: avg
 - Minimum: min
 - Maximum: max
 - Somme: sum
 - Count: count
- Remarque:
 - L'entrée de sum et avg doit être une collection de nombres.



Fonctions d'agrégation

- Aggregate Functions - Agrégation de base
- Requête 12: «Trouvez le salaire moyen des instructeurs du département d'informatique»

```
select avg(salary)
from instructor
where dept_name = 'Comp. Sci.';
```

- Résultat: une relation avec un seul attribut contenant un seul tuple avec une valeur numérique correspondant au salaire moyen des instructeurs du département d'informatique.
 - Remarque: Le SGBD peut donner un nom par défaut à l'attribut généré par agrégation, composé du texte de l'expression;
- MAIS nous pouvons donner un nom significatif en utilisant la clause **AS**

```
select avg(salary) as avg_salary
from instructor
where dept_name = 'Comp. Sci.';
```



Fonctions d'agrégation

- Aggregate Functions - Agrégation de base

- Les salaires au département d'informatique sont de 75 000, 65 000 et 92 000
 - Le salaire moyen est de $232\ 000 / 3 = 77\ 333,33$
 - La conservation des doublons est importante pour calculer une moyenne.
- Supposons que le département d'informatique ajoute un quatrième instructeur dont le salaire se situe à 75 000.
 - Si les doublons étaient éliminés, nous obtiendrions la mauvaise réponse ($232\ 000 / 4 = 58\ 000$) plutôt que la bonne réponse de 76 750.



Fonctions d'agrégation

- Aggregate Functions - Agrégation de base

- **DISTINCT**
- Il y a des cas où nous devons éliminer les doublons avant de calculer une fonction d'agrégation
 - nous utilisons le mot-clé **DISTINCT** dans l'expression d'agrégation.
- Requête 13: «Trouver le nombre total d'instructeurs qui enseignent un cours au semestre de printemps 2018».

```
select count(distinct ID)
from teaches
where semester = 'Spring' and year = 2018;
```

- En raison du mot clé distinct avant l'**ID**, même si un instructeur enseigne plusieurs cours, elle n'est comptée qu'une seule fois dans le résultat.



Fonctions d'agrégation

- Aggregate Functions - Agrégation de base

- **COUNT (*)**
 - Pour compter le nombre de tuples dans une relation.
 - Exemple: Trouver le nombre de tuples dans la relation course.

```
select count (*)
from course;
```

- Remarques:
 - SQL ne permet pas l'utilisation de distinct avec count (*).
 - Il est légal d'utiliser distinct avec max et min, même si le résultat ne change pas.
 - Nous pouvons utiliser le mot clé **all** au lieu de distinct pour spécifier la rétention en double, mais comme tout est la valeur par défaut, il n'est pas nécessaire de le faire.



Fonctions d'agrégation

Aggregate Functions - Agrégation avec regroupement

GROUP BY

- Dans des cas nous voudrions appliquer la fonction d'agrégation non seulement à un seul ensemble de tuples, mais aussi à un groupe d'ensembles de tuples;
- En SQL en utilisant la clause group by.
- L'attribut ou les attributs donnés dans la clause group by sont utilisés pour former des groupes.
- Les tuples avec la même valeur sur tous les attributs de la clause group by sont placés dans un groupe.
- Requête 14: «Trouvez le salaire moyen dans chaque département».

```
select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name;
```



Fonctions d'agrégation

Aggregate Functions - Agrégation avec regroupement

GROUP BY

- Requête 14: «Trouvez le salaire moyen dans chaque département»:
 - Etape 1 : regroupés les tuples par l'attribut dept_name: Fig gauche
 - Etape 2: L'agrégat spécifié est calculé pour chaque groupe: Fig droite

ID	name	dept_name	salary
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

dept_name	avg_salary
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000



Fonctions d'agrégation

Aggregate Functions - Agrégation avec regroupement

GROUP BY

- Requête 15: «Trouvez le nombre d'instituteurs dans chaque département qui enseignent un cours au semestre de printemps 2018»

```
SELECT dept_name, COUNT (DISTINCT ID)
AS instr_count
FROM instructor, teaches
WHERE instructor.ID= teaches.ID
and semester = 'Spring'
and year = 2018
GROUP BY dept_name;
```

dept_name	instr_count
Comp. Sci.	3
Finance	1
History	1
Music	1



Fonctions d'agrégation

Aggregate Functions - Agrégation avec regroupement

GROUP BY

- Règles:
- Lorsqu'une requête SQL utilise le regroupement, il est important:
 - de s'assurer que les seuls attributs qui apparaissent dans l'instruction select sans être agrégés sont ceux qui sont présents dans la clause group by
 - Autrement tout attribut qui n'est pas présent dans la clause group by ne peut apparaître dans la clause select qu'en tant qu'argument d'une fonction d'agrégation, sinon la requête est traitée comme erronée.
- Exemple:
 - requête suivante est erronée car l'ID n'apparaît pas dans la clause group by et pourtant il apparaît dans la clause select sans être agrégé:

```
select dept_name, ID, avg (salary)
from instructor
group by dept_name;
```



Fonctions d'agrégation

Aggregate Functions - Agrégation avec regroupement

La clause **HAVING**

- Parfois, il est utile d'indiquer une condition qui s'applique aux **groupes** plutôt qu'aux tuples.
- Par exemple:
 - nous pourrions nous intéresser uniquement aux départements où le salaire moyen des instructeurs est supérieur à 42 000.
 - Cette condition ne s'applique pas à un seul tuple; il s'applique plutôt à chaque groupe construit par la clause **GROUP BY**.



Fonctions d'agrégation

Aggregate Functions - Agrégation avec regroupement

La clause **HAVING**

- Pour exprimer une telle requête, nous utilisons la clause having de SQL.
- SQL applique des prédicats dans la clause having après la formation des groupes, de sorte que les fonctions d'agrégation peuvent être utilisées dans la clause having.
- Nous exprimons cette requête en SQL comme suit:

```
SELECT dept_name, avg (salary) as avg_salary
FROM instructor
GROUP BY dept_name
having avg (salary) > 42000;
```

dept_name	avg_salary
Physics	91000
Elec. Eng.	80000
Finance	85000
Comp. Sci.	77333
Biology	72000
History	61000



Fonctions d'agrégation

Aggregate Functions - Agrégation avec regroupement

La clause **HAVING**

- Comme pour select, tout attribut présent dans la clause having sans être agrégé doit apparaître dans la clause group by
 - sinon la requête est erronée.
- La signification d'une requête contenant des clauses d'agrégation, group by ou having est définie par la séquence d'opérations suivante:
 - Comme pour les requêtes sans agrégation, la clause from est d'abord évaluée pour obtenir une relation.
 - Si une clause where est présente, le prédicat de la clause where est appliqué à la relation de résultat de la clause from.
 - Les tuples satisfaisant le prédicat where sont ensuite placés en groupes par la clause group by si elle est présente. Si la clause group by est absente, l'ensemble entier de tuples satisfaisant le prédicat where est traité comme étant dans un groupe
 - La clause having, si elle est présente, est appliquée à chaque groupe; les groupes qui ne satisfont pas au prédicat de la clause having sont supprimés.
 - La clause select utilise les groupes restants pour générer des tuples du résultat de la requête, en appliquant les fonctions d'agrégation pour obtenir un tuple de résultat unique pour chaque groupe.



Fonctions d'agrégation

Aggregate Functions - Agrégation avec regroupement

La clause **HAVING**

- Exemple utilisation à la fois d'une clause having et d'une clause where dans la même requête:
 - Requête 16: «Pour chaque section de cours offerte en 2017, trouvez le nombre total moyen de crédits (tot_cred) de tous les étudiants inscrits dans la section, si la section compte au moins 2 étudiants.

```
SELECT course_id, semester, year, sec_id, avg
(tot_cred)
FROM student, takes
WHERE student.ID= takes.ID and year = 2017
GROUP BY course_id, semester, year, sec_id
HAVING count (ID) >= 2;
```




Fonctions d'agrégation

- **Aggregate Functions - Agrégation avec les valeurs Null et Boolean**
 - Les valeurs nulles, lorsqu'elles existent, compliquent le traitement des opérateurs d'agrégats.
 - Par exemple:
 - supposons que certains tuples dans la relation instructors ont une valeur nulle pour le salaire.
 - La requête suivante:

```
select sum (salary)
from instructor;
```

- Dans cette requête, Plutôt que de dire que la somme globale est elle-même nulle, la norme SQL indique que l'opérateur de SUM doit **ignorer les valeurs nulles dans son entrée**.
- En général, les fonctions d'agrégation traitent les valeurs null selon la règle suivante:
 - Toutes les fonctions d'agrégation, à l'exception de count (*), ignorent les valeurs null dans leur collection d'entrée.



Fonctions d'agrégation

- **Aggregate Functions - Agrégation avec les valeurs Null et Boolean**
 - Un type de données booléen pouvant prendre des valeurs true, false et unknown a été introduit dans SQL: 1999 (SQL3)
 - Les fonctions d'agrégation, **some** et **every**, peuvent être appliquées à une collection de valeurs booléennes et calculer la disjonction (ou) et la conjonction (et), respectivement, des valeurs.



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

- SQL fournit un mécanisme d'imbrication des sous-requêtes.
- Une sous-requête est une expression de SELECT-FROM-WHERE imbriquée dans une autre requête.
- Une utilisation courante des sous-requêtes consiste:
 - à effectuer des **tests d'appartenance** à un ensemble,
 - à effectuer des **comparaisons d'ensembles**,
 - à déterminer **la cardinalité d'un ensemble** en imbriquant des sous-requêtes dans la clause WHERE.



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Définir/Tester l'adhésion

- SQL permet de tester les tuples pour l'appartenance à une relation.
 - Les tests de connectivité **IN** pour l'appartenance à un ensemble, où l'ensemble est une collection de valeurs produites par une clause select.
 - L'absence de tests conjonctifs pour l'absence d'appartenance à un ensemble.
 - Exemple - Requête 17:
 - «Trouvez tous les cours enseignés dans les semestres d'automne 2017 et de printemps 2018».
 - Première solution (croiser 2 ensembles)
 - Une autre alternative: trouver tous les cours enseignés à l'automne 2017 et qui sont également membres de l'ensemble des cours enseignés au printemps 2018.
- 1- Nous commençons par trouver tous les cours enseignés au printemps 2018:

```
(select course_id
from section
where semester = 'Spring' and year= 2018)
```



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Définir/Tester l'adhésion

- Exemple - Requête 17:
 - «Trouvez tous les cours enseignés dans les semestres d'automne 2017 et de printemps 2018».
- 2- Ensuite trouver les cours qui ont été enseignés à l'automne 2017 et qui apparaissent dans l'ensemble des cours obtenus dans la sous-requête 1.

```
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Fall' and year= 2017 and
      Course_id IN (select course_id
                    from section
                    where semester = 'Spring' and year=
                      2018);
```



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Définir/Tester l'adhésion

- Notez que nous devons utiliser **DISTINCT**:
 - car l'opération d'intersection supprime les doublons par défaut.
- Cet exemple montre qu'il est possible d'écrire la même requête de plusieurs façons en SQL.
- Cette flexibilité est bénéfique, car elle permet à un utilisateur de penser à la requête de la manière qui semble la plus naturelle.



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Définir/Tester l'adhésion

- Nous utilisons la construction **NOT IN** d'une manière similaire à la construction **IN**.
- Par exemple:
 - pour trouver tous les cours enseignés au semestre d'automne 2017 mais pas au semestre de printemps 2018, que nous avons exprimé plus tôt en utilisant l'opération **except**, nous pouvons écrire:

```
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Fall' and year= 2017 and
      Course_id NOT IN (select course_id
                       from section
                       where semester = 'Spring' and year=
                         2018);
```



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Définir/Tester l'adhésion

- Les opérateurs **IN** et **NOT IN** peuvent également être utilisés sur les ensembles énumérés.
- Exemple: sélectionner les noms des instructeurs dont les noms ne sont ni «Mozart» ni «Einstein»

```
select distinct name
from instructor
where name not in ('Mozart', 'Einstein');
```

- Requête 18: «Trouvez le nombre total d'étudiants (distincts) qui ont suivi des sections de cours enseignées par l'instructeur avec l'ID 110011»

```
select count (distinct ID)
from takes
where (course_id, sec_id, semester, year) in (select
course_id, sec_id, semester, year
from teaches
where teaches.ID= '10101');
```




Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Comparaison d'ensemble

- À titre d'exemple de la capacité d'une sous-requête imbriquée à **comparer des ensembles**
- Requête 19- bis: «Trouvez les noms de tous les instructeurs dont le salaire est supérieur à au moins un instructeur du département de biologie»

```
SELECT name
FROM instructor
WHERE salary > SOME (select salary
                     from instructor
                     where dept_name = 'Biology');
```

SQL propose cependant un style alternatif pour écrire la requête précédente. L'expression «**supérieur à au moins un**» est représentée en SQL par **> some**. → proche à notre formulation de la requête en langue naturelle.



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Comparaison d'ensemble

- Requête 19- bis – suite: la sous requête génère **l'ensemble** de toutes les valeurs salaire de tous les instructeurs du département de biologie.
- La comparaison **> SOME** : vraie si la valeur salaire du tuple est supérieure **à au moins un membre de l'ensemble de toutes les valeurs** salariales pour les instructeurs en biologie.
- SQL permet aussi:
 - < some,
 - <= some,
 - >= some, = some, et <> some.
 - = some ⇔ in,
 - <> some ne sont pas les mêmes que not in.



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Comparaison d'ensemble

- Requête 19- bis 2 –Trouvons le nom de tous les instructeurs dont le salaire est supérieur à celui de chaque instructeur du département de biologie.
- La construction **>ALL** correspond à la phrase «**supérieur à tous**».

```
select name
from instructor
where salary > all (select salary
                  from instructor
                  where dept_name = 'Biology');
```

SQL permet également des comparaisons **<all, <= all, >= all, = all et <> all**. **<> all** est identique à **not in**, alors que **= all** n'est pas identique à **in**.



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Comparaison d'ensemble

- Requête 20: "Trouvez les départements qui ont le salaire moyen le plus élevé."
- 1. Nous commençons par écrire une requête pour trouver tous les salaires moyens,
- 2. Puis l'imbriquer en tant que sous-requête d'une requête plus large qui trouve les départements pour lesquels le salaire moyen est supérieur ou égal à tous les salaires moyens

```
select dept_name
from instructor
group by dept_name
having avg (salary) >= ALL (select avg (salary)
                          from instructor
                          group by dept_name);
```



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Test des relations vides

- SQL inclut une fonctionnalité pour tester si une sous-requête a des tuples dans son résultat.
- La construction **EXISTS** renvoie la valeur true si la sous-requête d'argument n'est pas vide.
- Requête 21: «Trouver tous les cours enseignés au semestre d'automne 2017 et au semestre de printemps 2018»

```
select course_id
from section as S
where semester = 'Fall' and year= 2017 and
      EXISTS (select *
              from section as T
              where semester = 'Spring' and year= 2018 and
                    S.course_id= T.course_id);
```



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Test des relations vides

- La requête illustre également une fonctionnalité de SQL où un **nom de corrélation** provenant d'une requête externe, peut être utilisé dans une sous-requête de la clause where.
- Une sous-requête qui utilise un nom de corrélation provenant d'une requête externe est appelée une **sous-requête corrélée**.
 - une règle de portée s'applique aux noms de corrélation.
 - Dans une sous-requête, selon la règle, il est légal d'utiliser uniquement des noms de corrélation définis dans la sous-requête elle-même ou dans toute requête qui contient la sous-requête.
 - Si un nom de corrélation est défini à la fois localement dans une sous-requête et globalement dans une requête conteneur, la définition locale s'applique.
 - Cette règle est analogue aux règles de portée habituelles utilisées pour les variables dans les langages de programmation.



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Test des relations vides

- Nous pouvons tester la non-existence de tuples dans une sous-requête en utilisant la construction **NOT EXISTS**.
- Nous pouvons utiliser la construction NOT EXISTS pour simuler l'opération de confinement d'ensemble (c'est-à-dire le surensemble):
- nous pouvons écrire «la relation A contient la relation B» comme "not exists (B except A)." . (Bien qu'il ne fasse pas partie des normes SQL actuelles, l'opérateur contains était présent dans certains premiers systèmes relationnels.)
- Pour illustrer l'opérateur not exists:
 - «Trouver tous les étudiants qui ont suivi tous les cours proposés dans le département de biologie»



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Test des relations vides

- Pour illustrer l'opérateur not exists:
 - «Trouver tous les étudiants qui ont suivi tous les cours proposés dans le département de biologie»

```
select S.ID, S.name
from student as S
where not exists ((select course_id
                  from course
                  where dept_name = 'Biology')
                 except
                 (select T.course id
                  from takes as T
                  where S.ID = T.ID));
```



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Test des relations vides

- Requête SQL pour «trouver le nombre total d'étudiants (distincts) qui ont suivi les sections de cours enseignées par l'instructeur avec l'ID 110011».
- Une autre façon d'écrire la requête, en utilisant la construction **EXISTS**, est la suivante:

```
select count (distinct ID)
from takes
where exists (select course_id, sec_id, semester, year
              from teaches
              where teaches.ID= '10101'
                 and takes.course_id = teaches.course_id
                 and takes.sec_id = teaches.sec_id
                 and takes.semester = teaches.semester
                 and takes.year = teaches.year
              );
```



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Test pour l'absence de tuples en double

- SQL inclut une fonction booléenne pour tester si une sous-requête a des tuples en double dans son résultat.
- La construction unique renvoie la valeur true si la sous-requête d'argument ne contient pas de tuples en double.
 - Requête 22: «Trouver tous les cours qui ont été proposés au plus une fois en 2017»

```
select T.course_id
from course as T
where unique (select R.course_id
              from section as R
              where T.course_id= R.course_id and
                 R.year = 2017);
```



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Sous-requêtes dans la clause From

- SQL permet d'utiliser une expression de sous-requête dans la clause from.
- Le concept clé appliqué ici est que toute expression SELECT-FROM-WHERE renvoie une relation comme résultat et, par conséquent, peut être insérée dans une autre select-from-where.
 - Requête 23«Trouvez le salaire moyen des instructeurs des départements où le salaire moyen est supérieur à 42 000 ».

```
select dept_name, avg_salary
from (select dept_name, avg (salary) as avg_salary
      from instructor
      group by dept_name)
where avg_salary > 42000;
```



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Sous-requêtes dans la clause From

- Synthèse requête 22
 - La sous-requête génère une relation composée des noms de tous les départements et des salaires moyens des instructeurs correspondants
 - Les attributs du résultat de la sous-requête peuvent être utilisés dans la requête externe.
 - Notez que nous n'avons **pas besoin** d'utiliser la clause **HAVING**
 - car la sous-requête de la clause FROM calcule le salaire moyen, et le prédicat qui figurait précédemment dans la clause HAVING **se trouve désormais dans la clause where de la requête externe.**



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Sous-requêtes dans la clause From

- Nous pouvons donner un nom à la relation de résultat de la sous-requête et renommer les attributs, en utilisant la clause **AS**.

```
select dept_name, avg_salary
from (select dept_name, avg (salary)
      from instructor
      group by dept_name)
      as dept_avg (dept_name, avg_salary)
where avg_salary > 42000;
```

- Les sous-requêtes imbriquées dans la clause **FROM** sont prises en charge par la plupart mais pas toutes les implémentations SQL.
 - Notez que certaines implémentations SQL, notamment MySQL et PostgreSQL, nécessitent que chaque relation de sous-requête dans la clause from reçoive un nom, même si le nom n'est jamais référencé;
 - Oracle permet de donner un nom à une relation de résultat de sous-requête (avec le mot-clé **AS** omis) mais ne permet pas de renommer les attributs de la relation.



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Sous-requêtes dans la clause From

- Une solution de contournement simple pour cela consiste à renommer l'attribut dans la clause select de la sous-requête; dans la requête la clause select de la sous-requête serait remplacée par

```
SELECT dept_name, avg(salary) AS avg_salary
```

- Et remplacer "**AS** dept_avg" par

```
"AS dept_avg"
```

- Exemple, **Requête 23**: nous souhaitons trouver le maximum dans tous les départements du total des salaires de tous les instructeurs dans chaque département.
- La clause **HAVING** ne nous aide pas dans cette tâche, mais nous pouvons écrire cette requête facilement en utilisant une sous-requête dans la clause from, comme suit:

```
select max (tot_salary)
from (select dept_name, sum(salary)
      from instructor
      group by dept_name) as dept_total (dept_name, tot_salary);
```



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

la clause with

- La clause **WITH** permet de définir une relation temporaire dont la définition n'est disponible que pour la requête dans laquelle la clause with se produit.
 - Exemple, **Requête 24**: trouver les départements avec le budget maximum.

```
WITH max_budget (value) AS
  (select max(budget)
   from department)
select budget
from department, max_budget
where department.budget = max_budget.value;
```



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

la clause with

- Requête 24: Synthèse

- La clause **WITH** dans la requête définit le budget max de la **relation temporaire** contenant les résultats de la sous-requête définissant la relation.
- La **relation** est disponible pour une utilisation **uniquement dans les parties ultérieures de la même requête**.
- La clause **WITH**, introduite dans **SQL: 1999**, est prise en charge par de nombreux systèmes de base de données, mais pas tous.
- Nous aurions pu écrire la requête précédente en utilisant une sous-requête imbriquée dans la clause **FROM** ou la clause **WHERE**.
- Cependant, l'utilisation de sous-requêtes imbriquées aurait rendu la requête plus difficile à lire et à comprendre.
- Le claus **WITH** rend la logique de requête plus claire;
 - il permet également d'utiliser cette relation temporaire à plusieurs endroits d'une requête.



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

la clause with

- Par exemple, Requête 25:
 - Supposons que nous voulons trouver tous les départements où le salaire total est supérieur à la moyenne du salaire total de tous les départements.
- Nous pouvons écrire la requête en utilisant la clause **WITH** comme suit.

```
WITH dept_total (dept_name, value) AS
  (select dept_name, sum(salary)
   from instructor
   group by dept_name),
dept_total_avg(value) AS
  (select avg(value)
   from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value > dept_total_avg.value;
```



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Sous-requêtes scalaires

- SQL permet aux sous-requêtes de se produire partout où une expression **renvoyant une valeur** est autorisée, à condition que la sous-requête ne renvoie **qu'un seul tuple** contenant **un seul attribut**;
 - Ces sous-requêtes sont appelées **sous-requêtes scalaires**.
- Par exemple, Requête 26:
 - Répertorie tous les départements ainsi que le **nombre** d'instructeurs dans chaque département:

```
SELECT dept_name, (select count(*)
                  from instructor
                  where department.dept_name =
                        instructor.dept_name)
                  AS num_instructors
FROM department;
```



Sous-requêtes imbriquées

Sous-requêtes imbriquées - Nested Subqueries

Sous-requêtes scalaires

- La sous-requête dans cet exemple (R26) est garantie de ne renvoyer qu'une seule valeur car elle a un agrégat **COUNT (*)** sans **GROUP BY**.
- L'exemple illustre également l'utilisation des **variables de corrélation**, c'est-à-dire les attributs des relations dans la clause **FROM** de la requête externe, comme **department.dept_name**.
- Des **sous-requêtes scalaires** peuvent se produire dans les clauses **SELECT, WHERE ET HAVING**.



Sous-requêtes imbriquées - FIN

Sous-requêtes imbriquées - Nested Subqueries

Sous-requêtes scalaires

- Les sous-requêtes scalaires peuvent également être définies sans agrégats.
 - Il n'est pas toujours possible de déterminer au moment de la compilation si une sous-requête peut renvoyer plus d'un tuple dans son résultat; si le résultat a plus d'un tuple lorsque la sous-requête est exécutée, une erreur d'exécution se produit.
- Notez que techniquement, le type d'un résultat de sous-requête scalaire est toujours une relation, même s'il contient un seul tuple.
 - Cependant, lorsqu'une sous-requête scalaire est utilisée dans une expression où une valeur est attendue, SQL extrait implicitement la valeur de l'attribut unique du tuple unique dans la relation et renvoie cette valeur.



Modification de la BD

Modification de la BD

DELETE

- Après les base SQL pour extraire les d'informations de la base de données.
- Comment ajouter, supprimer ou modifier des informations avec SQL.

DELETE

- Une requête **DELETE** est exprimée de la même manière qu'une requête normale.
- Remarque:
 - Nous ne pouvons supprimer que des tuples entiers;
 - Nous ne pouvons pas supprimer des valeurs uniquement sur des attributs particuliers.
- La norme SQL exprime une suppression par:

```
DELETE FROM r
WHERE P;
```



Modification de la BD

Modification de la BD

DELETE

- P : un prédicat et r: une relation (table).
- Fonctionnement:
 - DELETE trouve d'abord tous les tuples t dans r pour lesquels P (t) est vrai,
 - puis les supprime de r.
- La clause WHERE peut être omise:
 - auquel cas **tous les tuples de r sont supprimés**.
- Notez qu'une DELETE fonctionne sur **une seule** relation.
- Le prédicat dans la clause where peut être aussi complexe qu'une clause where d'une commande select.
- À l'autre extrême, la clause WHERE peut être vide.



Modification de la BD

Modification de la BD

DELETE

- Exemples:
 - DELETE FROM instructor;
 - supprime tous les tuples de la relation.
 - La relation existe toujours, mais elle est vide.
 - DELETE FROM instructor WHERE dept_name = 'Finance';
 - Supprime tous les tuples du service finance.
 - DELETE FROM instructor WHERE salary BETWEEN 13000 and 15000;
 - ... ?
 - DELETE FROM instructor WHERE dept_name IN (SELECT dept_name FROM department WHERE building = 'Watson');
 - ... ?
 - DELETE FROM instructor WHERE salary < (SELECT **AVG**(salary) FROM instructor);
 - ... ?



Modification de la BD

Modification de la BD

Insertion – INSERT INTO

- Pour insérer des données dans une relation:
 1. nous spécifions un tuple à insérer ou
 2. écrivons une requête dont le résultat est un ensemble de tuples à insérer.
- Les valeurs d'attribut pour les tuples insérés doivent être membres du domaine de l'attribut correspondant (**contrainte de domaine**).
- De même, les tuples insérés doivent avoir **le nombre correct** d'attributs.
- Exemple:
 - Supposons que nous souhaitons insérer le fait qu'il existe un cours CS-437 dans le département d'informatique avec le titre "Database Systems" et quatre heures de crédit

```
INSERT INTO course
VALUES ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```



Modification de la BD

Modification de la BD

Insertion – INSERT INTO

- Dans cet exemple, les valeurs sont spécifiées dans l'ordre dans lequel les attributs correspondants sont répertoriés dans le schéma de relation.
- Pour le bénéfice des utilisateurs qui ne se souviennent peut-être pas de l'ordre des attributs, **SQL permet de spécifier les attributs dans le cadre de l'instruction d'insertion.**
- Par exemple:
 - les instructions d'insertion SQL suivantes sont identiques en fonction à la précédente:

```
INSERT INTO course(course_id, title, dept_name, credits)
VALUES ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

```
INSERT INTO course(title, course_id, credits, dept_name)
VALUES ('Database Systems', 'CS-437', 4, 'Comp. Sci.');
```



Modification de la BD

Modification de la BD

Insertion – INSERT INTO

- Plus généralement, nous pourrions vouloir insérer des tuples **sur la base du résultat d'une requête.**
- Exemple:
 - Supposons que nous voulons faire de chaque étudiant du département de musique qui a gagné plus de 144 heures de crédit un instructeur du département de musique avec un salaire de 18 000

```
INSERT INTO instructor
select ID, name, dept_name, 18000
from student
where dept_name='Music' and tot_cred > 144;
```



Modification de la BD

Modification de la BD

Mise à jour – UPDATE

- Dans certaines situations, nous pouvons souhaiter modifier une valeur dans un tuple sans changer toutes les valeurs dans le tuple.
- Comme nous pourrions l'insérer et la supprimer, nous pouvons choisir les tuples à mettre à jour en utilisant une requête.
- Exemple:
 - Supposons que des augmentations de salaire annuelles soient effectuées et que les salaires de tous les instructeurs soient augmentés de 5%:

```
UPDATE instructor
SET salary = salary * 1.05;
```

- Si une augmentation de salaire doit être versée uniquement aux instructeurs dont le salaire est inférieur à 70 000 →

```
UPDATE instructor
SET salary = salary * 1.05
WHERE salary < 70000;
```



Modification de la BD

Modification de la BD

Mise à jour – UPDATE

- En général, la clause where de l'instruction update peut contenir toute construction légale dans la clause where de l'instruction select (y compris les sélections imbriquées).
- Comme pour l'insertion et la suppression, une sélection dans une instruction de update peut référencer la relation en cours de mise à jour.
- SQL teste d'abord tous les tuples de la relation pour voir s'ils doivent être mis à jour, puis effectue les mises à jour par la suite.
- Exemple:
 - Requête «Donner une augmentation de salaire de 5% aux instructeurs dont le salaire est inférieur à la moyenne»

```
UPDATE instructor
SET salary = salary * 1.05
WHERE salary < (select AVG(salary)
from instructor);
```




Modification de la BD

Modification de la BD

Mise à jour – UPDATE

■ Exemple:

- Requête « Supposons maintenant que tous les instructeurs dont le salaire est supérieur à 100 000 reçoivent une augmentation de 3%, tandis que tous les autres reçoivent une augmentation de 5% »

```
UPDATE instructor
SET salary = salary *1.03
WHERE salary > 100000;

UPDATE instructor
SET salary = salary *1.05
WHERE salary <= 100000;
```

Note: l'ordre des deux instructions de mise à jour est important → Si nous modifions l'ordre des deux relevés, un instructeur avec un salaire d'un peu moins de 100000 recevrait une augmentation de plus de 8%.



Modification de la BD

Modification de la BD

Mise à jour – UPDATE

■ Solution:

- SQL fournit une construction **CASE** que nous pouvons utiliser pour effectuer les deux mises à jour avec une seule instruction de mise à jour, en évitant le problème avec l'ordre des mises à jour.

```
update instructor
set salary = case
    when salary <= 100000 then salary *1.05
    else salary *1.03
end
```

syntaxe générale

```
case
    when pred1 then result1
    when pred2 then result2
    ...
    when predn then resultn
    else result0
end
```



Modification de la BD

Modification de la BD

Mise à jour – UPDATE

- Les **sous-requêtes scalaires** sont utiles dans les instructions update SQL, où elles peuvent être utilisées dans la clause **SET**.
- Exemple: une mise à jour où nous définissons l'attribut tot_cred de chaque tuple étudiant sur la somme des crédits des cours réussis par l'étudiant.
- Nous supposons qu'un cours est réussi si l'étudiant a une note qui n'est ni «F» ni nulle.
 - Pour spécifier cette mise à jour, nous devons utiliser une **sous-requête** dans la clause **SET**:

```
UPDATE student
SET tot_cred = (
    select SUM(credits)
    from takes, course
    where student.ID= takes.ID and
    takes.course_id = course.course_id and
    takes.grade <> 'F' and
    takes.grade is not null);
```



Modification de la BD

Modification de la BD

Mise à jour – UPDATE

- Dans le cas où un étudiant n'a pas réussi un cours, l'instruction précédente définirait la valeur d'attribut tot_cred sur null.
 - Pour définir la valeur à 0 à la place, nous pourrions utiliser une autre instruction de mise à jour pour remplacer les valeurs nulles par 0;
 - une meilleure alternative consiste à remplacer la clause «select sum (credits)» dans la sous-requête par la clause select suivante en utilisant une expression de cas:

```
select case
    when sum(credits) is not null then sum(credits)
    else 0
end
```




FIN