

Le langage SQL

Structured Query Language



Semestre 2 – 2016/2017

B. EL HATIMI



SQL – *Structured Query language*

- Le langage SQL est un langage de définition et de manipulation des bases de données.
- En tant que langage de manipulation des données, sa syntaxe relève des langages prédicatifs (comme le calcul de tuples et le calcul de domaines).
- Dans son implémentation, le langage SQL est basé sur l'algèbre relationnelle (opérateurs algébriques).

B.EL HATIMI - SQL



Deux langages en un

- SQL est :
 - Un langage de description de données LDD (*Data Definition Language*)
 - Un langage de manipulation de données LMD (*Data Manipulation Language*)

B.EL HATIMI - SQL



Un peu d'histoire ...

- 1974 : IBM lance une réflexion autour d'un langage de données relationnel
- 1982 : IBM82 Première version SQL commercialisée
- 1989 : SQL1 ou SQL-89 (norme ISO89)
- 1992 : SQL2 ou SQL-92 (norme ISO92)
- SQL3 (évolution vers l'objet)

B.EL HATIMI - SQL



SQL : Les versions

- SQL1 [ISO89] : 110 pages.
- SQL2 [ISO92]: 580 pages.
- SQL3 : >2000 pages.

B.EL HATIMI - SQL



Pourquoi SQL ?

- SQL est la seule norme (norme ANSI et ISO) existante pour les LDD/LMD relationnels.
- Tous les SGBDR (soit > 90% du marché des bases de données) utilisent une version de SQL comme LMD/LDD.

B.EL HATIMI - SQL



Compatibilité des SGDBR avec la norme SQL

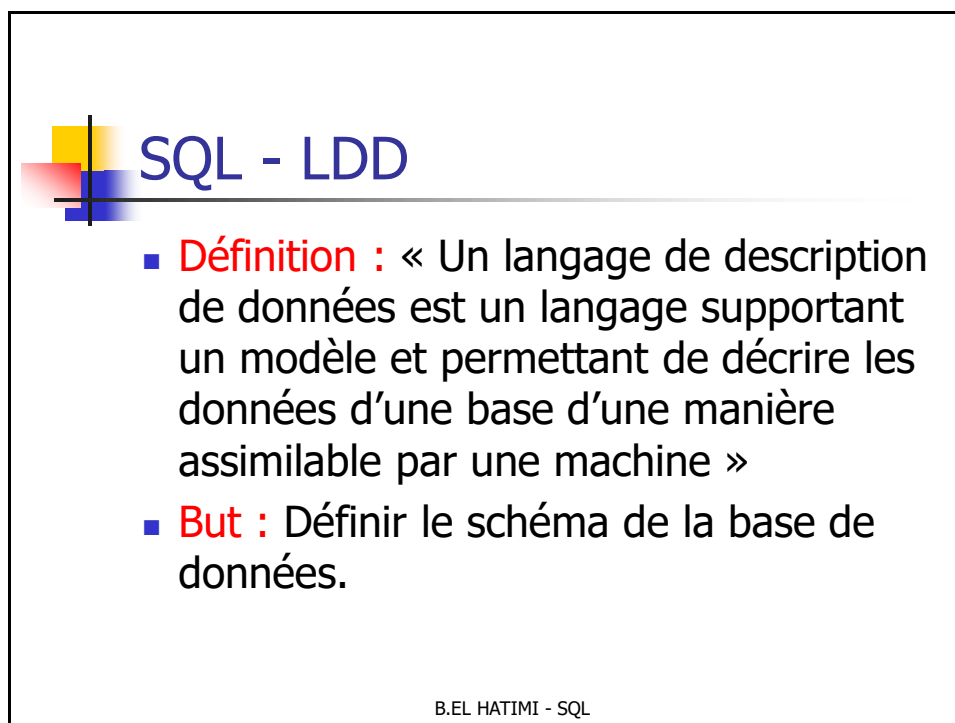
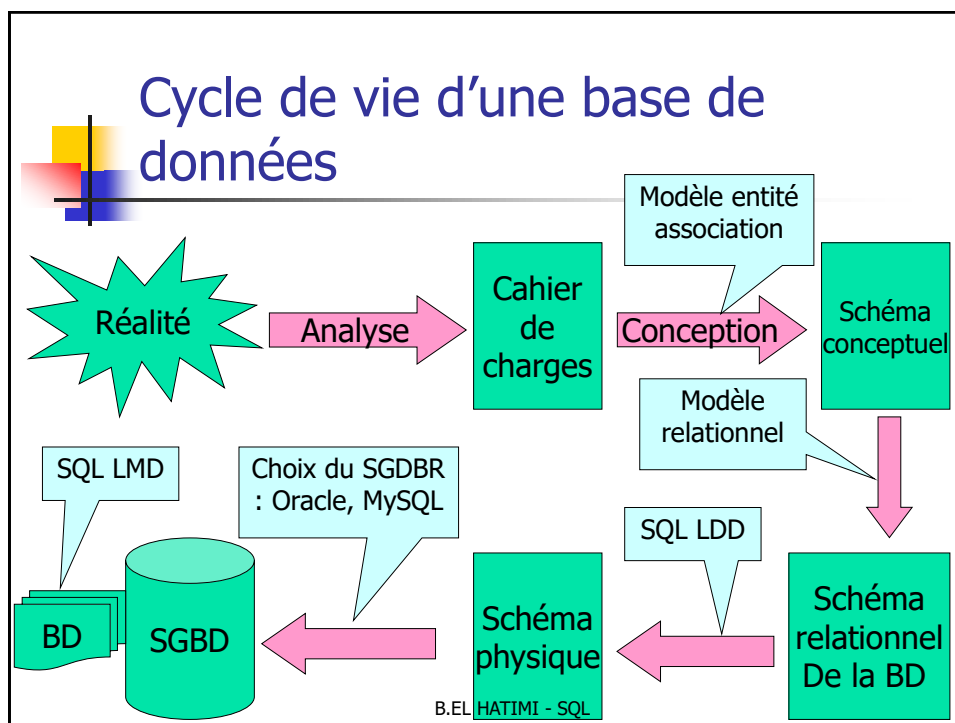
- Chaque éditeur de SGDBR a sa propre version de la syntaxe SQL → Parfois des incompatibilités entre eux.
- Dans une optique d'interopérabilité il faut vérifier systématiquement la compatibilité de la syntaxe de votre script avec la norme SQL.
- Exemples à partir de la documentation PostgreSQL :
 1. There is no REINDEX command in the SQL standard.
 2. This CREATE TYPE command is a PostgreSQL extension. There is a CREATE TYPE statement in the SQL standard that is rather different in detail.
 3. The CREATE TABLE command conforms to the SQL standard, with exceptions listed below.
 4. The command CREATE DOMAIN conforms to the SQL standard.

B.EL HATIMI - SQL



Langage SQL

Partie 1 : SQL – Langage de description de données






Création du schéma

- SQL permet de créer une base de données composées de plusieurs schémas.
- Mot clé : **CREATE**
- Création de la base de données
create database BD1
- Création du schéma de la base de données
create schema GESTION_SCOL

B.EL HATIMI - SQL



Script généré par PostgreSQL à la création d'une base de données

```
CREATE DATABASE "Candidats"  
  WITH OWNER = postgres  
      ENCODING = 'UTF8'  
      TABLESPACE = pg_default  
      LC_COLLATE = 'French_France.1252'  
      LC_CTYPE = 'French_France.1252'  
      CONNECTION LIMIT = -1;
```

B.EL HATIMI - SQL



Remarque

- La syntaxe SQL n'est pas sensible à la casse (nom des tables, des attributs ...). Les valeurs des attributs par contre le sont.

B.EL HATIMI - SQL



Démarche d'exécution d'une requête SQL

1. Analyse syntaxique;
2. Optimisation de la requête;
3. Réécriture de la requête;
4. Générer un plan d'exécution (arbre avec des opérateurs);
5. Exécution de la requête.

B.EL HATIMI - SQL



Création d'une table

- Un schéma est composé de tables.
- Une table est créée comme suit :

```
create table ETUDIANT (      CNE char (10),  
                             Nom char (30),  
                             Prenom char (40),  
                             DateNais date)
```

- Le SGBDR interprète ETUDIANT comme:
DB.GESTION_SCOL.ETUDIANT
- La référence complète d'une table est sous la
forme : [Nom_BD].[Nom_Schéma].Nom_Table

B.EL HATIMI - SQL



Types de données

- Chaque SGBDR fournit un ensemble de
types de données plus ou moins riche.
- Les types de données ne sont pas
normalisés → Risque d'incompatibilités
entre SGBDR.
- Il faut vérifier le type de données par
rapport au SGBDR utilisé.

B.EL HATIMI - SQL



Types de données

- SMALLINT
- INTEGER (ou INT)
- DECIMAL(p,q)
- FLOAT(p) (ou REAL)
- CHAR(p)
- VARCHAR(p)
- DATE
- BLOB (Binary large Object) : contenant générique pouvant accueillir des chaînes de bits de grande taille (images, séquences vidéo...) ...

B.EL HATIMI - SQL



Domaine de données

- Le langage SQL offre la possibilité de définir de nouveaux types de données.
- Mot clé **domain**
- Exemple :
 - create domain** *MONTANT* **as** decimal (9,2)
- Le domaine est alors utilisé comme un type de données standard :

create table ACHAT (Code char(5),
Valeur *MONTANT*)

B.EL HATIMI - SQL



Exemple (Documentation PostgreSQL)

- `CREATE DOMAIN us_postal_code AS TEXT
CHECK (VALUE ~ '^d{5}$'
OR VALUE ~ '^d{5}-d{4}$');`

Create domain code as text check (VALUE ~
'char(1)\\int\\int\\int')

- `CREATE TABLE us_snail_addy (address_id
SERIAL PRIMARY KEY, street1 TEXT NOT NULL,
street2 TEXT, street3 TEXT, city TEXT NOT
NULL, postal us_postal_code NOT NULL);`

B.EL HATIMI - SQL



Exercice

CANDIDAT (Code_CNC clé primaire, Nom,
Prenom, Adresse, date_naissance)

Créer un domaine pour le code CNC

Format : XX000Y (XX : ville, 000 : numéro
d'ordre, Y : filière M, P, T, B)

Ex : SL028M (SL : Salé, 028 Ordre, M : MP)

Saisir des enregistrements pour tester

B.EL HATIMI - SQL



Clé primaire

- Mot clé : **primary key**
 - **create table** ETUD
(CNE char (10) **primary key**,
Nom char (30),
Prenom char (40),
DateNais date
Moyenne decimal(2,2))

B.EL HATIMI - SQL



Clé primaire

Les deux syntaxes suivantes sont équivalentes :

```
create table ETUD1  
  (CNE char (10) primary key,  
   Moyenne decimal(2,2))
```

```
create table ETUD1  
  (CNE char (10),  
   Moyenne decimal(2,2),  
   primary key (CNE))
```

Dans la première la clé primaire est une contrainte associée à la colonne et dans la deuxième elle est associée à la table.

B.EL HATIMI - SQL



Code généré par PostgreSQL

```
CREATE TABLE etud1
( cne character(10) NOT NULL,
  moyenne numeric(2,2),
  CONSTRAINT etud1_pkey PRIMARY KEY (cne))
WITH ( OIDS=FALSE );
ALTER TABLE etud1
  OWNER TO postgres;
```

B.EL HATIMI - SQL



Clé primaire composée

- On peut créer une clé primaire composée de plusieurs attributs.
- **create table** ETUD
(Nom char (30),
Prenom char (40),
DateNais date,
Moyenne decimal(2,2),
primary key (Nom,Prenom))

B.EL HATIMI - SQL



Contrainte d'unicité

- La contrainte d'unicité est précisée par le mot-clé **unique**.
- **create table** ETUDIANT
(CNE char (10) **primary key**,
CIN char (8) **unique**,
Nom char (30),
Prenom char (40),
DateNais date)

B.EL HATIMI - SQL



Contrainte d'unicité

- La contrainte d'unicité peut s'appliquer sur un ensemble d'attributs :
- **create table** ETUDIANT
(CNE char (10) **primary key**,
Nom char (30),
Prenom char (40),
DateNais date,
unique (Nom,Prenom))

B.EL HATIMI - SQL



Clé secondaire

- Le langage SQL ne permet pas de définir formellement une clé secondaire.
- On peut donc utiliser UNIQUE pour préciser les clés secondaires.

B.EL HATIMI - SQL



Attribut facultatif / obligatoire

- Tout attribut est facultatif par défaut → accepte la valeur NULL
- La valeur NULL est une valeur conventionnelle pour représenter une information inconnue ou inapplicable.
- Pour imposer le caractère obligatoire on ajoute la clause **not null** lors de la création de l'attribut.
- La valeur NULL n'enfreint pas la contrainte d'unicité.

B.EL HATIMI - SQL



Attribut facultatif / obligatoire

- **create table** ETUDIANT
(CNE char (10) **primary key**,
CIN char (8) **unique not null**,
Nom char (30) **not null**,
Prenom char (40),
DateNais date)
- Remarque : En pratique, les SGBDR considèrent la contrainte *primary key* comme équivalente à *unique* et *not null*.

B.EL HATIMI - SQL



Exercice

- Soit le schéma relationnel de la table u décrit comme suit :
- u (nu : Entier, nomu : Chaine(100), ville : Chaine(50))
- nu est la clé primaire
- nomu est obligatoire
- Ville a comme valeur par défaut 'Casablanca'

B.EL HATIMI - SQL



Solution

```
CREATE TABLE u (nu integer primary key,  
  nomu character varying (100) not null,  
  Ville character varying(50) default  
  'Casablanca');
```

B.EL HATIMI - SQL



Valeur par défaut

- Mot Clé **default**
 - A la création de la table on peut associer une valeur par défaut à une colonne :
create table PRODUIT (... CAT char(1) *default* 'P' ...)
 - A la création du domaine on peut associer une valeur par défaut à celui-ci :
create domain MONTANT *decimal* (9,2) *default* 0.0

B.EL HATIMI - SQL



Clé étrangère

- Mots clés: FOREIGN KEY / REFERENCES
- Syntaxe avec contrainte de table :

```
create table ACHAT (  
    Code    char(10) primary key,  
    Prod char(8) not null,  
    Quantité smallint,  
    foreign key (Prod) references PRODUIT )
```

B.EL HATIMI - SQL



Clé étrangère

- Mots clés: FOREIGN KEY / REFERENCES
- Syntaxe avec contrainte de colonne :

```
create table ACHAT (  
    Code    char(10) primary key,  
    Prod char(8) not null references PRODUIT,  
    Quantité smallint)
```

B.EL HATIMI - SQL



Contrainte référentielle

- La clé étrangère précise une contrainte référentielle sur un ou plusieurs attributs de la table vers un ou plusieurs attributs d'une autre table qui sont soit une clé primaire ou une clé secondaire.

- Syntaxe complète :

FOREIGN KEY (*column* [, ...]) **REFERENCES** *reftable* [(*refcolumn* [, ...])]

- Si *refcolumn* est omis, la contrainte référentielle pointe sur la clé primaire.

B.EL HATIMI - SQL



Clé étrangère composée

- Une clé étrangère peut être composée de plusieurs colonnes :

create table TAB1 (
 Att1 Type1 primary key,
 Att2 Type2,
 Att3 Type3,
foreign key (Att2,Att3) **references** TAB2
 (Att4,Att5))

B.EL HATIMI - SQL



Exercice

- Créer une table achat décrit comme suit :
- Achat (id : entier, date_achat : date, usine : entier)
- Id est la clé primaire
- Date_achat est obligatoire
- Usine est une clé étrangère vers u

B.EL HATIMI - SQL



Solution

```
CREATE TABLE achat  
( id integer primary key,  
  Date_achat date not null,  
  Usine integer references u);
```

B.EL HATIMI - SQL



Contrainte référentielle

- Que se passe t-il en cas de modification ou de suppression des données en référence ? → Risque de violation de la contrainte référentielle.
- Afin d'assurer l'intégrité de la base de données, le langage SQL propose cinq actions possibles
- Syntaxe :

```
FOREIGN KEY ( column [, ... ] ) REFERENCES reftable [  
  ( refcolumn [, ... ] ) ] [ ON DELETE action ] [ ON  
  UPDATE action ]
```

B.EL HATIMI - SQL



Contrainte référentielle

- **NO ACTION** : Génère une erreur indiquant la violation de la contrainte référentielle. C'est l'action par défaut.
- **RESTRICT** : Génère une erreur indiquant la violation de la contrainte référentielle. La modification ou la suppression de la donnée en référence est interdite.
- La différence ? (voir documentation)

B.EL HATIMI - SQL



Contrainte référentielle

- **CASCADE** : En cas de suppression de la valeur référencée, supprime les lignes contenant une référence à celle-ci. En cas de modification de la valeur référencée, modifie la valeur de la clé étrangère dans les lignes contenant une référence à la valeur modifiée.

B.EL HATIMI - SQL



Contrainte référentielle

- **SET NULL** : Remplace la valeur référencée par NULL dans la clé étrangère.
- **SET DEFAULT** : Remplace la valeur référencée par la valeur par défaut (si elle existe) dans la clé étrangère. Il faut que la valeur par défaut existe dans la table de référence.

B.EL HATIMI - SQL



Contrainte référentielle

■ Exemple :

```
create table ACHAT (
  Code    char(10) primary key,
  Prod char(8) not null references PRODUIT on
update cascade on delete no action,
  Quantité smallint)
```

B.EL HATIMI - SQL



Contrainte référentielle

- | | |
|---|--|
| 1. Il est interdit de supprimer un produit s'il a été acheté. | 1. ON DELETE NO ACTION
⇔ Ne rien écrire |
| 2. Même si on supprime le produit on veut garder les achats correspondants. | 2. ON DELETE SET NULL / SET DEFAULT |
| 3. Un produit acheté ne peut pas être modifié. | 3. ON UPDATE NO ACTION
⇔ Ne rien écrire |
| 4. On change souvent le code de nos produits, dans ce cas il faut changer le code dans les achats correspondants. | 4. ON UPDATE CASCADE |
| 5. Si on supprime les produits on ne veut pas garder les achats. | 5. ON DELETE CASCADE |



Exercice

- Créer une table achat décrit comme suit :
- Achat (id : entier, date_achat : date, usine : entier)
- Usine est une clé étrangère vers u. Valeur par défaut 0. En cas de modification du code d'une usine modifier la valeur des clés étrangères. En cas de suppression d'une usine, mettre la valeur par défaut pour les clés étrangères.

B.EL HATIMI - SQL



Solution

```
CREATE TABLE achat  
( id integer primary key,  
  date_achat date not null,  
  usine integer default 0 references u on  
  delete set default on update cascade);
```

B.EL HATIMI - SQL



Contrainte référentielle

- Remarque : Certains SGBD acceptent la création d'une clé étrangère vers une table qui n'est pas encore créée (*référence en avant*)
- Si ce n'est pas possible, il faudra ajouter la contrainte après la création de la table de référence.


B.EL HATIMI - SQL



Exercice : Création du schéma PUF

- Donner le script SQL pour créer le schéma suivant :
- U (NU, NomU, Ville)
- P (NP, NomP, Couleur, Poids)
- F (NF, NomF, Statut, Ville)
- PUF (NP, NU, NF, Quantité)
- NP référence P.NP, NU référence U.NU, NF référence F.NF
- On suppose qu'on interdit la suppression si la clé est utilisée et qu'on cascade la modification.


B.EL HATIMI - SQL



Exercice : Création du schéma PUF

```
CREATE TABLE PUF (  
  NU integer references U on delete no action on  
  update cascade,  
  NF integer references F on delete no action on  
  update cascade,  
  NP integer references P on delete no action on  
  update cascade,  
  QUANTITE integer not null,  
  Primary key (NU, NP, NF))
```

B.EL HATIMI - SQL



Passage des contraintes du schéma en SQL

CONTRAINTE RELATIONNELLE	EQUIVALENT SQL
Contrainte d'entité / de clé	PRIMARY KEY
Contrainte d'unicité	UNIQUE
Contrainte de non nullité	NOT NULL
Contrainte référentielle	FOREIGN KEY / REFERENCES
Contrainte de domaine	Nom_Colonne Type_Domaine

B.EL HATIMI - SQL



Passage des contraintes du schéma en SQL

- Que faire dans le cas des contraintes comportementales ?

B.EL HATIMI - SQL



Contrainte générale : CHECK

- La clause CHECK permet de contraindre la valeur d'une colonne à vérifier un prédicat. Le prédicat correspond à une expression logique. Elle peut aussi vérifier un prédicat sur la valeur de deux colonnes.
- Exemple :

```
CREATE TABLE products  
( product_no integer primary key,  
  name text,  
  price numeric CHECK (price > 0) );
```

B.EL HATIMI - SQL



Contrainte générale : CHECK

```
CREATE TABLE FILM
(titre VARCHAR (50) NOT NULL,
annee INTEGER CHECK (annee BETWEEN 1890 AND 2000)
NOT NULL,
genre VARCHAR (10) CHECK (genre IN
('Histoire', 'Western', 'Drame')),
idMES INTEGER,
codePays INTEGER,
PRIMARY KEY (titre),
FOREIGN KEY (idMES) REFERENCES Artiste,
FOREIGN KEY (codePays) REFERENCES Pays);
```

B.EL HATIMI - SQL



CHECK sur deux colonnes

- Dans cet exemple la clause CHECK vérifie pour chaque enregistrement que la valeur de son attribut price est supérieur à la valeur de son attribut discounted_price
- CREATE TABLE products
(product_no integer primary key,
name text,
price numeric CHECK (price > 0),
discounted_price numeric CHECK
(discounted_price > 0),
CHECK (price > discounted_price));

B.EL HATIMI - SQL



Contrainte implicite Vs Contrainte explicite

- Le code suivant crée une contrainte d'entité implicite sur la colonne code
 - `CREATE TABLE films (code character(5) PRIMARY KEY,...)`
- On peut créer la contrainte de manière explicite, en lui associant un nom, avec le mot clé **CONSTRAINT**. La syntaxe deviant :
 - `CREATE TABLE films (code character(5) NOT NULL,..., CONSTRAINT films_pkey PRIMARY KEY (code))`

B.EL HATIMI - SQL



Pseudo-type SERIAL

- Sous PostgreSQL, le pseudo-type SERIAL permet de créer une séquence puis de l'associer aux valeurs d'une colonne.
- `CREATE TABLE etudiant (id serial primary key,...);`
- Il existe trois types SERIAL différents selon leur taille : smallserial sur 2 octets / serial sur 4 / bigserial sur 8.
- Le pseudo-type SERIAL est équivalent au mot-clé AUTO-INCREMENT de certains SGBD.

B.EL HATIMI - SQL



Exercice 1

- Schéma de la BD Gestion_ACHAT :
 1. CLIENT (NCLI, Nom, [Adresse], [Ville])
 2. COMMANDE (NCOM, Client, DateCommande, [DateLivraison])
 3. PRODUIT (NPRO, Libelle, Prix, [Stock])
 4. DETAIL (Commande, Produit, Quant)
- Client référence NCLI
- Commande référence NCOM
- Produit référence NPRO

B.EL HATIMI - SQL



Exercice 2

- Des éditeurs se réunissent pour créer une Base de Données sur leurs publications scientifiques. Dans de telles publications, plusieurs auteurs se regroupent pour écrire un livre en se répartissant les chapitres à rédiger. Après discussion, voici le schéma obtenu :
 1. LIVRE (**titreLivre**, année, éditeur, chiffreAffaire)
 2. CHAPITRE (**titreLivre**, **titreChapitre**, nbPages)
 3. AUTEUR (**nomAuteur**, prénom, annéeNaissance)
 4. REDACTION (**nomAuteur**, **titreLivre**, **titreChapitre**)
- Donnez les ordres CREATE TABLE pour le schéma, en spécifiant soigneusement clés primaires et étrangères
- Précisez les stratégies de modification et de suppression sur les clés étrangères?

B.EL HATIMI - SQL



Exercice 3

PERSONNE (ID entier, [CIN] chaine(10), nom chaine(50), prenom chaine(50), naissance date, sexe, [deces] date, [pere] entier, [mere] entier, [nationalite] chaine(30))

- ID est une clé artificielle
- CIN est unique
- PERSONNE.pere référence PERSONNE.ID
- PERSONNE.mere référence PERSONNE.ID
- Interdire la suppression d'une personne qui a des enfants
- En cas de modification de l'ID d'un parent, répercuter la modification sur les enfants
- nationalite a comme valeur par défaut 'MAROCAINE'
- La date de décès est supérieure à la date de naissance
- sexe prend deux valeurs possibles (M, F)


B.EL HATIMI - SQL



Solution Exercice 3

```
CREATE TABLE personne (
  ID serial primary key,
  Cin varchar(10) unique,
  Nom varchar(50) not null,
  Prenom varchar(50) not null,
  Naissance date not null,
  Deces date,
  Sexe char check (sexe IN ('M', 'F')),
  Pere integer references personne on update cascade,
  Mere integer references personne on update cascade,
  Nationalite varchar(50) default 'MAROCAINE',
  Check (naissance < deces))
```


B.EL HATIMI - SQL



Modification du schéma

- Après la création du schéma de base de données (clause CREATE) on peut le modifier en supprimant ou en modifiant des tables, des colonnes, des contraintes ...

B.EL HATIMI - SQL



Modification du schéma

- Suppression des objets de la base de données : TABLE, DOMAIN, INDEX, FUNCTION... → DROP
- Modification du schéma d'une table :
 - Renommer la table
 - Renommer une colonne
 - Ajouter/Supprimer une colonne
 - Modifier le type d'une colonne
 - Ajouter / Modifier / Supprimer une valeur par défaut
 - Ajouter / Modifier / Supprimer une contrainte : Clé primaire, Clé étrangère, Check ...

B.EL HATIMI - SQL



Suppression d'un domaine

- Commande DROP DOMAIN
- **DROP DOMAIN** nom_domaine [CASCADE | RESTRICT]
- RESTRICT : Option par défaut. Interdit la suppression tant que le domaine est utilisé.
- CASCADE : Force la suppression du domaine et les colonnes qui en dépendent sont supprimées.

B.EL HATIMI - SQL



Suppression d'une table

- Commande DROP TABLE
- **DROP TABLE** nom_table [CASCADE | RESTRICT]
- RESTRICT : Option par défaut. Interdit la suppression de la table tant qu'il y a des objets qui en dépendent (ex: clés étrangères).
- CASCADE : Forcer la suppression de la table et les objets qui en dépendent sont supprimés.

B.EL HATIMI - SQL



Renommer une table

- Syntaxe:

ALTER TABLE old_name RENAME TO new_name

Remarque : Dans PostgreSQL le renommage est impacté (en cascade) au niveau des tables en référence.

B.EL HATIMI - SQL



Renommer une colonne

Syntaxe PostgreSQL :

*ALTER TABLE [IF EXISTS] [ONLY] name
[*] RENAME [COLUMN] column_name
TO new_column_name*



Exemple :

*ALTER TABLE p RENAME COLUMN np TO
numero*

B.EL HATIMI - SQL



Insertion d'une colonne

- Syntaxe :
 - ALTER TABLE *produit* ADD COLUMN *description* TEXT;
 - ADD [COLUMN] *column_name data_type* [COLLATE *collation*] [*column_constraint* [...]]
- La colonne est insérée avec la valeur NULL pour les enregistrements existant dans la table,

B.EL HATIMI - SQL



Suppression d'une colonne

- Syntaxe :
 - ALTER TABLE *produit* DROP COLUMN *prix*;
- La colonne est supprimée avec toutes les contraintes associées, sauf si la colonne est référencée par d'autres objets,
- Exemple : Clé primaire : La clé primaire est supprimée sauf si cette dernière est référencée par une clé étrangère,

B.EL HATIMI - SQL



Modifier le type d'une colonne

- Syntaxe :
 - ALTER TABLE *nom_table* ALTER [COLUMN]
nom_colonne TYPE *type*
 - ALTER TABLE *puf* ALTER *qte* TYPE *numeric*;
- Les valeurs déjà existantes seront converties si possible. Si les valeurs existantes ne peuvent pas être converties, l'utilisateur doit spécifier la méthode de conversion,
- Remarque : Il est préférable de supprimer toutes les contraintes liées à une colonne avant de modifier son type puis de les réécrire après la modification.

B.EL HATIMI - SQL



Contrainte explicite

- On peut donner un nom aux contraintes avec le mot-clé **CONSTRAINT**.
- Exemple :

```
create table CLIENT
(  IdCli char(3),
   Nom char(30),
   Adresse char(100),
   Ville char(20),
   constraint C2 primary key (IdCli))
```

B.EL HATIMI - SQL



Contrainte explicite

- Ajouter une contrainte avec un nom. Exemple:
`alter table CLIENT`
`add constraint CS1 unique (Nom,Adresse,Ville)`
- Supprimer une contrainte par son nom :
`alter table CLIENT`
`drop constraint CS1`

B.EL HATIMI - SQL



Supprimer une contrainte clé primaire

- Syntaxe :
 - `ALTER TABLE produit DROP CONSTRAINT produit_pkey`
 - *produit_pkey* est le nom de la contrainte clé étrangère
- Conséquences : La table ne peut plus être éditée et donc on ne peut plus insérer de nouveaux enregistrements,

B.EL HATIMI - SQL



Ajouter une contrainte clé primaire

- Syntaxes :
 - ALTER TABLE produit ADD PRIMARY KEY (nproduit);
 - ALTER TABLE produit ADD CONSTRAINT PK1 PRIMARY KEY (nproduit);
- Conditions : La colonne (ou les colonnes) choisie comme clé primaire ne doivent pas contenir de valeur dupliquées ou NULL.

B.EL HATIMI - SQL



Supprimer une contrainte référentielle

- Syntaxe :
 - ALTER TABLE puf DROP FOREIGN KEY (nu); ????
 - ALTER TABLE puf DROP CONSTRAINT puf_nu_fkey;

B.EL HATIMI - SQL



Ajouter une contrainte référentielle

- Syntaxe :
 - ALTER TABLE puf ADD FOREIGN KEY (nu) REFERENCES u;
 - ALTER TABLE puf ADD CONSTRAINT ma_contrainte FOREIGN KEY (nu) REFERENCES u;
- Conditions : Il faut que toutes les valeurs dans la colonne existent dans la colonne de référence.

B.EL HATIMI - SQL



Ajout / Suppression d'une valeur par défaut

- Ajout ou changement de la valeur par défaut. Seuls les nouveaux enregistrements seront concernés :

ALTER [COLUMN] *column* SET DEFAULT *expression*

- Enlever la valeur par défaut. Seuls les nouveaux enregistrements seront concernés :

ALTER [COLUMN] *column* DROP DEFAULT

- Exemple :

```
alter table ETUDIANT  
alter column FILIERE set default 'SIG'
```

B.EL HATIMI - SQL



Colonne obligatoire/facultative

- Rendre une colonne obligatoire. Ceci n'est possible que si aucune valeur NULL n'existe dans la colonne :

```
ALTER TABLE distributors ALTER COLUMN street  
SET NOT NULL
```

- Pour rendre une colonne facultative :

```
ALTER TABLE distributors ALTER COLUMN street  
DROP NOT NULL
```

B.EL HATIMI - SQL



Exercice : Ajout d'une clé artificielle

- On veut modifier la table PUF en remplaçant la clé naturelle (NU, NP, NF) par une clé artificielle ID.
 1. Ajouter à la table PUF un attribut ID de type SERIAL. A quoi correspond ce type ?
 2. Déclarer ID comme la nouvelle clé primaire.
 3. Que se passe t-il si vous aviez dupliqué une valeur, avant de déclarer ID comme clé primaire ?

B.EL HATIMI - SQL



Exercice : Ajout d'une clé artificielle

- `Alter table puf add column id serial;`
- `Alter table puf drop constraint puf_pkey;`
- `Alter table puf add primary key (id);`
- Si on veut garder la contrainte d'unicité sur (NP, NU, NF) on ajoute la commande :
 - `ALTER TABLE products ADD CONSTRAINT unicate_npnunf UNIQUE (np, nu, nf);`

B.EL HATIMI - SQL



Pseudo-type SERIAL

- SERIAL est un pseudo-type entier permettant au SGBD de créer une séquence puis d'associer à la colonne des valeurs issues de la séquence.
- Syntaxe: `CREATE TABLE tablename (colname SERIAL);`
- Selon la taille voulue on peut utiliser SMALLSERIAL, SERIAL ou BIGSERIAL.

B.EL HATIMI - SQL



Les séquences

- Une séquence permet de créer un générateur de valeurs entières sous forme d'une énumération de valeurs uniques à partir d'une valeur de départ (par défaut 1) et d'un pas d'incrémentation (par défaut 1).
- Syntaxe : `CREATE SEQUENCE seq START 101;`
- La fonction `nextval` permet de récupérer la valeur courante de l'énumération :
- `select nextval('seq');`

B.EL HATIMI - SQL



SERIAL Vs SEQUENCE

```
CREATE TABLE etudiant (id serial  
primary key, nom varchar(50), prenom  
varchar(50));
```


```
CREATE SEQUENCE seq;  
CREATE TABLE etudiant (id integer  
default nextval('seq') primary key, nom  
varchar(50), prenom varchar(50));
```

B.EL HATIMI - SQL



Langage SQL

Partie 2 : SQL – Langage de manipulation de données



Manipulation des données

- Insérer des enregistrements : INSERT
- Modifier des enregistrements : UPDATE
- Supprimer des enregistrements : DELETE
- Sélectionner des enregistrements : SELECT

B.EL HATIMI - SQL



INSERT

- En respectant l'ordre des colonnes de la table, on peut ajouter des enregistrements :

INSERT INTO NomTable VALUES (Val1, Val2, Val3...)

- On peut changer l'ordre d'entrée des colonnes :

INSERT INTO NomTable (Col1, Col2, Col3)
VALUES (Val1, Val2, Val3)


B.EL HATIMI - SQL



INSERT

- DETAIL (Commande, Produit, Quant)
- *insert into* DETAIL *values* ('22547', 'P147', 25)
- Cas de l'insertion d'une valeur NULL :
 - *insert into* DETAIL *values* ('22548', 'P130', NULL) ⇔ *insert into* DETAIL (Commande, Produit) *values* ('22548', 'P130')
 - *insert into* DETAIL *values* ('22549', 'P160',)

B.EL HATIMI - SQL




INSERT

- On peut ne pas introduire toutes les valeurs. Dans ce cas il faut spécifier les colonnes voulues :
- CLIENT (NCLI, Nom, Adresse, Ville, [Type], Compte)

```
insert into CLIENT (Nom, NCLI, Ville, Compte)  
values ('CASALUX', 'C312', 'Casablanca', 2000)
```

B.EL HATIMI - SQL



INSERT

- On peut ajouter à une table **existante** des données extraites d'une autre table :

```
insert into CLIENT_TANGER  
select NCLI, Nom, Adresse  
from CLIENT  
where Ville = 'Tanger'
```

B.EL HATIMI - SQL



Commande COPY

- Lorsqu'on veut insérer un grand nombre de lignes ou importer des données à partir d'un fichier texte, il est préférable d'utiliser la commande COPY.
- La commande COPY ne fait pas partie du standard SQL. C'est une commande propre à PostgreSQL.

B.EL HATIMI - SQL



DELETE


- Les enregistrements ne sont pas référencés individuellement aussi n'y a-t-il pas de commande pour supprimer un enregistrement en particulier.
- Par contre, on peut supprimer toutes les lignes d'une table :

DELETE FROM films;

- Ou on peut supprimer les lignes répondant à une condition :

DELETE FROM films **WHERE** genre = 'Horreur'

B.EL HATIMI - SQL




DELETE

- Suppression de lignes : *delete*
- Exemple on veut supprimer les détails de commande qui spécifient des produits en rupture de stock :

```
delete from DETAIL  
      where Produit in ( select NPRO  
                        from PRODUIT  
                        where Qstock = 0)
```

B.EL HATIMI - SQL



TRUNCATE

- TRUNCATE quickly removes all rows from a set of tables. It has the same effect as an unqualified DELETE on each table, but since it does not actually scan the tables it is faster. Furthermore, it reclaims disk space immediately, rather than requiring a subsequent VACUUM operation. This is most useful on large tables.

B.EL HATIMI - SQL



VACUUM

- VACUUM reclaims storage occupied by dead tuples. In normal PostgreSQL operation, tuples that are deleted or obsoleted by an update are not physically removed from their table; they remain present until a VACUUM is done. Therefore it's necessary to do VACUUM periodically, especially on frequently-updated tables.


B.EL HATIMI - SQL



UPDATE

- Modifier les valeurs : *update*
update CLIENT
 set Ville = 'Casablanca'
 where Ville = 'Casa'


B.EL HATIMI - SQL



Exemple

- Schéma de la BD Gestion_ACHAT :
 - CLIENT (NCLI, Nom, Adresse, Ville, [Type], Compte)
 - COMMANDE (NCOM, Client, Date)
 - PRODUIT (NPRO, Libelle, Prix, QStock)
 - DETAIL (Commande, Produit, Quant)
 - Client référence NCLI
 - Commande référence NCOM
 - Produit référence NPRO


B.EL HATIMI - SQL



Consultation et extraction de données

- Syntaxe de la requête : **select ... from ... [where]**
...
- **Select** : précise les nom des colonnes qui constituent chaque ligne du résultat.
- **From** : précise les tables desquelles sont extraites les données.
- **Where** : donne les conditions de sélection des lignes cibles.
- Le résultat d'une requête est une table temporaire.

B.EL HATIMI - SQL




Extraction simple

- Requête :

```
select NCLI, Nom, Ville  
from CLIENT
```
- Equivalent en algèbre relationnelle à une projection :
$$\Pi [NCLI, Nom, Ville] CLIENT$$

B.EL HATIMI - SQL




Extraction simple

- Select * permet l'extraction de toutes les colonnes des tables cibles.
- Exemple, si on veut afficher tous les enregistrements de la table CLIENT :

```
select *  
from CLIENT
```

B.EL HATIMI - SQL




Extraction de lignes sélectionnées

- La clause WHERE permet d'inclure un prédicat lors de la sélection des enregistrements :

```
select NCLI, Nom  
from CLIENT  
where Ville = `Tanger`
```
- Equivalent en algèbre relationnel à :
 $\Pi [NCLI, Nom] \sigma [Ville = `Tanger`] CLIENT$

B.EL HATIMI - SQL



Extraction de lignes sélectionnées

- Si une clé entière n'est pas reprise dans la clause SELECT le résultat peut contenir des lignes identiques.

```
select Ville  
from CLIENT  
where Type = `C1`
```

→

Ville
Tanger
Casablanca
Rabat
Casablanca

B.EL HATIMI - SQL



Extraction de lignes sélectionnées

- On pourra éliminer les lignes en double par la clause **distinct** :

```
select distinct Ville  
from CLIENT  
where Type = 'C1'
```



Ville
Casablanca
Tanger
Rabat

B.EL HATIMI - SQL



Extraction de lignes sélectionnées

Remarque importante :

- Si la clause SELECT cite toutes les colonnes d'une table ou contient une clé entière (primaire ou secondaire), l'unicité des lignes est garantie.

B.EL HATIMI - SQL



Prédicat de sélection

- Le prédicat de sélection a la forme d'une expression logique. Exemples :
 - **WHERE** Valeur = 12.5
 - **WHERE** Nom = 'InfoNord'
 - **WHERE** date > '10/08/2006'
- Syntaxe : **WHERE** [Colonne/Constante] {=,<,>,<=,>=,<>} [Colonne/Constante]

B.EL HATIMI - SQL



Prédicat de sélection

- Un prédicat peut combiner plusieurs conditions.
- Soient P1, P2 et P3 des expressions logiques :
 - **Where** P1 **and** P2
 - **Where** P1 **or** P2
 - **Where not** P1
 - **Where** P1 **and** (P2 **or** P3)

B.EL HATIMI - SQL



Prédicat de sélection

- Pour afficher Nom, Adresse et Compte des clients de Casablanca ayant un compte négatif.

```
Select Nom, Adresse, Compte  
From CLIENT  
Where Ville = 'Casablanca' and Compte <= 0
```

B.EL HATIMI - SQL



Condition sur la valeur NULL

- La clause SELECT peut inclure un prédicat sur la valeur NULL.
- Syntaxe : WHERE NomColonne [**is null** / **is not null**]
- Exemple :

```
Select Nom, Adresse, Compte  
From CLIENT  
Where Ville is null
```

B.EL HATIMI - SQL



Autres prédicats de sélection

- Condition sur l'appartenance à une liste :

Type **in** ('C1','B2','A2')

Ville **not in** ('Casablanca','Rabat','Tanger')

- Condition sur l'appartenance à un intervalle de valeurs :

Compte **between** 10000 **and** 50000

Date **not between** '01/01/2001' **and** '31/12/2005'

B.EL HATIMI - SQL



Autres prédicats de sélection

- Condition sur les chaînes de caractères :

Type **like** '_1' → _ remplace un caractère

Nom **not like** '%sarl%' → % remplace une chaîne de caractères

- Si on veut inclure les caractères % et _ dans les valeurs :

Nom **like** '%\$_GEO%' **escape** '\$'

Rq. : on peut utiliser un autre caractère spécial que \$.

B.EL HATIMI - SQL



Ordre des lignes du résultat

- La clause **ORDER BY** permet de trier le résultat d'une requête selon une ou plusieurs colonnes.
- La clause ORDER BY vient après la clause WHERE.
- Exemple : Afficher les clients triés par ville.

```
select * from CLIENT  
order by Ville
```


B.EL HATIMI - SQL



Ordre des lignes du résultat

- Le tri peut se faire selon plusieurs colonnes :
select *
from CLIENT
order by Ville, Nom
- Les clients seront triés par ville et pour une même ville par nom.

B.EL HATIMI - SQL




Ordre des lignes du résultat

- Par défaut le tri se fait par ordre croissant des valeurs. On peut spécifier l'ordre explicitement avec les mots clés **asc** (croissant) et **desc** (décroissant).
- Donner les clients triés par ordre décroissant de nom :

```
Select *  
from CLIENT  
order by Nom desc
```

B.EL HATIMI - SQL



Clause LIMIT

- `LIMIT { count | ALL } OFFSET start`
- *count* specifies the maximum number of rows to return, while *start* specifies the number of rows to skip before starting to return rows. When both are specified, *start* rows are skipped before starting to count the *count* rows to be returned.

B.EL HATIMI - SQL



Données dérivées

- Pour les sélections simples, les données extraites proviennent directement des tables.
- On peut aussi spécifier dans la clause SELECT des données dérivées c'est-à-dire issues d'un calcul ou des constantes.

B.EL HATIMI - SQL



Données dérivées

- Exemple : Donner le montant de la TVA des produits en stock dont la quantité en stock est supérieure à 500.

```
Select 'TVA de', NPRO, '=', 0.2*Prix*QStock  
From PRODUIT  
Where QStock > 500
```

B.EL HATIMI - SQL



Données dérivées

TVA de	NPRO	=	0.2*Prix*QStock
TVA de	P402S	=	66520.00
TVA de	P384T	=	12478.00
TVA de	P55T	=	110540.00
TVA de	P121R	=	87456.00

B.EL HATIMI - SQL



Alias de colonne

- Lors de l’affichage du résultat, les colonnes reçoivent un nom qui est celui indiquée dans la clause SELECT.
- Le nom peut être encombrant et peu significatif. On peut alors définir explicitement un nom de colonne qui apparaîtra à l’affichage : **alias de colonne**.

B.EL HATIMI - SQL



Alias de colonne

```
Select NPRO as Produit, 0.2*Prix*QStock as  
Valeur_TVA  
From PRODUIT  
Where QStock > 500
```

B.EL HATIMI - SQL



Alias de colonne

Produit	Valeur_TVA
P402S	66520.00
P384T	12478.00
P55T	110540.00
P121R	87456.00

B.EL HATIMI - SQL



Fonctions et opérateurs SQL

- Le langage SQL offre un grand nombre de fonctions et d'opérateurs. Exemple, les fonctions mathématiques (COS(x), SQRT(x) ...) ou les opérateurs arithmétiques : *, +, /, ...
- L'appel à ces fonctions et à ces opérateurs se fait dans les clauses SELECT ou WHERE. Les arguments sont des noms de colonnes, des constantes ou des expressions.

B.EL HATIMI - SQL



Fonctions et opérateurs SQL

- Fonctions sur les chaînes de caractères :
 - Longueur de la chaîne : char_length(ch)
 - Concaténation : ch1 || ch2
 - Transformation en minuscules : lower(ch)
 - Transformation en majuscules : upper(ch)
 -

B.EL HATIMI - SQL



Fonctions SQL

- Exemple : Pour concaténer le nom en majuscules de la société et son adresse en minuscules :
- `select upper(Nom) || '@' || lower(Adresse)`
`as Identification`
`from Client`

B.EL HATIMI - SQL



Données agrégées et fonctions agrégatives

- Les fonctions simples retournent un résultat par ligne sélectionnée.
- Les fonctions dites agrégatives retournent une seule valeur agrégée calculée à partir de toutes les lignes sélectionnées.
- Exemple 1 : la fonction `Count(*)` retourne le nombre de lignes trouvées par une requête de sélection.
- Exemple 2 : la fonction `Min(NomColonne)` retourne le minimum des valeurs de la colonne.
- Remarque : les fonctions agrégatives ignorent les valeurs NULL.

B.EL HATIMI - SQL



Fonctions agrégatives

- Principales fonctions agrégatives :
 - Count(*) : nombre de lignes trouvées.
 - Count (nom_col) : nombre de lignes avec valeur de la colonne nom_col non nulle.
 - Avg(nom_col) : moyenne des valeurs de la colonne.
 - Sum(nom_col) : somme des valeurs de la colonne.
 - Min(nom_col) : minimum des valeurs de la colonne.
 - Max(nom_col) : maximum des valeurs de la colonne.

B.EL HATIMI - SQL




Fonctions agrégatives

- Donner le nombre et la moyenne des comptes des clients de Casablanca ?

```
Select count (*) as Nombre, avg (Compte) as  
Moyenne  
From CLIENT  
Where Ville = 'Casablanca'
```

Nombre	Moyenne
32	4355.13

B.EL HATIMI - SQL




Fonctions agrégatives

- Donner la valeur totale du stock ?

```
select sum (Qstock*Prix) as Valeur_Stock
from PRODUIT
```

Valeur_Stock
82500.00

B.EL HATIMI - SQL



Fonctions agrégatives

```
select sum (Qstock) as Total_Stock, Libelle
from PRODUIT
```

→ FAUX !!! On ne peut pas utiliser une sélection simple sur une colonne avec une fonction agrégative.

B.EL HATIMI - SQL



Fonctions agrégatives

- Attention aux valeurs dupliquées !
- La requête suivante, ne donne pas le nombre de clients ayant fait au moins une commande mais le nombre de commandes où Client n'est pas NULL :

```
select count (Client)  
from COMMANDE
```

B.EL HATIMI - SQL



Fonctions agrégatives

- Pour avoir le nombre de clients ayant fait au moins une commande on utilisera :

```
Select count (distinct Client)  
From COMMANDE
```

- A ne pas confondre avec :

```
Select distinct count (Client)  
From COMMANDE
```

B.EL HATIMI - SQL



Fonctions agrégatives

- On veut sélectionner les produits ayant comme prix le prix maximal de tous les produits.
- La requête 1 est fausse car on ne peut pas utiliser une fonction agrégative dans la clause WHERE :

Select NPRO

From Produit

Where Prix = Max(Prix)

→ Solution utiliser une sous-requête :

Select NPRO

From Produit

Where Prix = (Select Max(Prix) From Produit)

B.EL HATIMI - SQL



Données groupées (group by)

- Mot clé : group by
- Donner pour chaque ville le nombre de ses clients et la moyenne de leurs comptes ?

select Ville,


count(*) as NombCli,

avg(Compte) as MoyCompte

from CLIENT

group by Ville


B.EL HATIMI - SQL



Données groupées (group by)

- Remarque :
 - PostgreSQL prend en compte la valeur NULL lors d'un GROUP BY.

B.EL HATIMI - SQL



Données groupées

- On ne peut spécifier dans la clause SELECT que des noms de colonnes et des fonctions dont le résultat est une valeur unique par groupe : critère de groupement, fonctions agrégatives, constantes...

B.EL HATIMI - SQL



Sélection sur des données groupées

- Clause **having** au lieu de where (sélection sur des lignes).
- Sur la même requête précédente on ne veut garder que les villes ayant plus de trois clients.

B.EL HATIMI - SQL



Sélection sur des données groupées

```
select Ville,  
count(*) as NombCli,  
avg(Compte) as MoyCompte  
from CLIENT  
where compte > 50000  
group by Ville  
having count(*) >= 3
```

B.EL HATIMI - SQL



Extraction de données de plusieurs tables

- On a traité des requêtes simples dans les exemples précédents avec extraction de données à partir d'une seule table ?
- Comment extraire des requêtes à partir de plusieurs tables ?
 - Sous-requêtes
 - Jointures implicites ou explicites

B.EL HATIMI - SQL



Sous-requêtes

- Donner les commandes des clients de Casablanca.
- On commence par trouver les clients de Casablanca avec la requête :

```
Select NCLI  
From CLIENT  
Where Ville = 'Casablanca'
```
- Qui donne les clients {'C201', 'C805', 'C47', 'C214'}.

B.EL HATIMI - SQL



Sous-requêtes

- On peut alors trouver leur commande avec la requête :

Select NCOM, Date

From COMMANDE

Where Client **in** ('C201', 'C805', 'C47', 'C214')

➔ Méthode non satisfaisante !!!

B.EL HATIMI - SQL



Sous-requêtes

- On écrira :

Select NCOM, Date


From COMMANDE

Where Client **in** (**Select** NCLI
From CLIENT

Where Ville = 'Casablanca')

- La requête incluse dans la clause Where est appelée sous-requête.


B.EL HATIMI - SQL



Sous-requêtes

- Donner les produits qui ont été commandés par au moins un client de Casablanca ?
- CLIENT (NCLI, Nom, Adresse, Ville, [Type], Compte)
- COMMANDE (NCOM, Client, Date)
- PRODUIT (NPRO, Libelle, Prix, QStock)
- DETAIL (Commande, Produit, Quant)

B.EL HATIMI - SQL



Sous-requêtes

- On écrira :

```
Select Produit From DETAIL  
Where Commande in  
(Select NCOM From COMMANDE  
Where Client in (Select NCLI  
                  From CLIENT  
                  Where Ville = 'Casablanca'))
```
- On peut avoir plusieurs niveaux de sous-requêtes.

B.EL HATIMI - SQL



Sous-requêtes : Références à une même table

- Quels sont les clients habitant dans la même ville que le client 'C805' ?

- Requête :

```
select * from CLIENT
where Ville = ( Select Ville from CLIENT
               where NCLI = 'C805')
```

B.EL HATIMI - SQL



Sous-requêtes

- Si la sous-requête renvoie une seule ligne, on peut utiliser les opérateurs de comparaison.

- Exemple :

```
select * from CLIENT
where Compte > ( select Compte
                  from CLIENT
                  where NCLI = 'C805')
```

B.EL HATIMI - SQL



Sous-requêtes

- Exercice : Ecrire la requête qui donne les commandes avec une quantité commandée du produit 'P001' inférieure à celle de la commande 'CO001' (pour le même produit) ?
- CLIENT (NCLI, Nom, Adresse, Ville, [Type], Compte)
- COMMANDE (NCOM, Client, Date)
- PRODUIT (NPRO, Libelle, Prix, QStock)
- DETAIL (Commande, Produit, Quant)

B.EL HATIMI - SQL



Sous-requêtes : Confusion sur les références

- En règle général, la référence se rapporte à la sous-requête la plus emboîtée dont une table contient ce nom de colonne.
- En cas de confusion on peut créer un alias de table :

```
select * from PRODUIT as P  
where P.Prix > 200
```
- Remarque : Le mot-clé **as** est facultatif.

B.EL HATIMI - SQL



Sous-requêtes : Confusion sur les références

- Exemple : Trouver les clients dont le compte est supérieur à la moyenne des comptes des clients de la même ville.
- ```
Select * from CLIENT AS C
where Compte > (select avg(Compte)
 from CLIENT
 where Ville = C.Ville)
```

B.EL HATIMI - SQL



## Recherche avec condition sur les ensembles

- exists / not exists : Teste si l'ensemble des lignes retournées est vide ou pas.
- Quels sont les produits qui n'ont pas été commandés ?  

```
select * from PRODUIT
where not exists (select *
 from DETAIL
 where Produit = NPRO)
```

B.EL HATIMI - SQL



## Sous-requêtes avec quantificateurs

- **all / any / some** : permettent de comparer une valeur avec celles d'un ensemble défini par une sous-requête.
- Syntaxe :  
<valeur attribut> <opérateur de comparaison> < **all** / **any** / **some** > <ensemble>
- Donner les détails de la (ou des) commande avec la quantité commandée minimale pour le produit 'PP21' ?

B.EL HATIMI - SQL



## Sous-requêtes avec quantificateurs

```
select *
from DETAIL
where Quant <= all
 (select Quant
 from DETAIL
 where Produit = 'PP21')
and Produit = 'PP21'
```

B.EL HATIMI - SQL



## Sous-requêtes avec quantificateurs

- any / some : ce sont des synonymes. Ils testent s'il existe dans un ensemble au moins un élément qui satisfait une condition:
- Ensemble des numéros des fournisseurs de produits rouges :

```
SELECT NF FROM PUF
WHERE NP = ANY (SELECT NP FROM P
WHERE couleur = 'rouge')
```

B.EL HATIMI - SQL



## Sous-requêtes avec quantificateurs

- Remarque :
  - $IN \Leftrightarrow = ANY$
  - $NOT IN \Leftrightarrow <> ANY$

B.EL HATIMI - SQL



## Jointures

---

- Une jointure contrairement aux sous-requêtes permet d'extraire simultanément des données de plusieurs tables.

B.EL HATIMI - SQL



## Jointures

---

- Exemple : Compléter les commandes avec les informations du client :

```
select NCOM, NCLI, DateCommande,
Nom, Adresse, Ville
from COMMANDE, CLIENT
where Client = NCLI
```

B.EL HATIMI - SQL



## Jointures

- La condition *NCLI = Client* est dite **condition de jointure**.
- Elle est dans ce cas sous la forme de Clé étrangère = Clé primaire → Equi-jointure

B.EL HATIMI - SQL



## Jointures

```
select NCOM, CLIENT.NCLI, Date, Nom,
 Adresse, Ville
from COMMANDE, CLIENT
where NCLI = Client
and CAT = 'C1'
and Date < '01/01/2005'
```

B.EL HATIMI - SQL



## Jointures sans conditions :

```
select NCOM, CLIENT.NCLI, Date, Nom,
 Adresse, Ville
```

```
from COMMANDE, CLIENT
```

- C'est le produit cartésien des relations CLIENT et COMMANDE.

B.EL HATIMI - SQL




## Vues (*VIEW*)

- Définition : « Une vue est une table virtuelle dont le schéma et les tuples sont dérivés de la base de données réelle à partir d'une requête. »

- Syntaxe :

```
CREATE VIEW ma_vue AS Requête
```


B.EL HATIMI - SQL



## Gestion des vues

```
CREATE VIEW Commande_Casa AS
SELECT NCom, Date
FROM Client, Commande
WHERE NCLI = Client
And Ville = 'Casablanca'
```

B.EL HATIMI - SQL



## Gestion des vues

- Une vue peut être utilisée (presque) partout où une table peut être utilisée.

```
SELECT * FROM Commande_Casa
```

- Il est possible de construire une vue à partir d'autres vues.

B.EL HATIMI - SQL



## Gestion des vues

- Supprimer une vue :  
**DROP VIEW** ma\_vue
- Modifier une vue (renommer, modifier le schéma...) :  
**ALTER VIEW** ma\_vue [OPTIONS]

B.EL HATIMI - SQL



## Opérateurs ensemblistes

- UNION / EXCEPT (ou MINUS) / INTERSECT
- Syntaxe : Requete1 <Opérateur> Requete2 avec Requete1 et Requete2 retournant le même schéma.
  1. Requete1 **UNION** Requete2 → Union des résultats des deux requêtes.
  2. Requete1 **EXCEPT** Requete2 → Différence entre les résultats des deux requêtes.
  3. Requete1 **INTERSECT** Requete2 → Intersection des résultats des deux requêtes.

B.EL HATIMI - SQL