

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## SELECT

- ▶ Le langage d'interrogation de données (LID) permet d'établir une **combinaison d'opérations** portant sur des tables (relation). Le résultat de cette combinaison d'opérations est lui-même une table dont l'existence ne dure qu'un temps.
- ▶ La commande d'interrogation des données en SQL est: **SELECT**

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## SELECT

<b>SELECT</b>	NomChamp1, NomChamp2	Champs à projeter ou à calculer ou fonction d'agrégat
<b>FROM</b>	TABLE1, TABLE2...	Tables utiles à la requête
<b>WHERE</b>	Expression AND/OR	Jointures et restrictions
<b>GROUP BY</b>	NomChamp	Regroupement de résultat d'opérations d'agrégat
<b>HAVING</b>	Expression	Restriction sur l'affichage des résultats d'opérations d'agrégat
<b>ORDER BY</b>	NomChamp [ASC]/DESC	Critères de tri

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## SELECT

- ▶ Le **SELECT** est la commande de base du SQL destinée à **extraire des données** d'une base ou **calculer de nouvelles données** à partir d'existantes...
- ▶ Pour afficher des données d'une ou plusieurs colonnes d'une table, vous utilisez la commande **SELECT** avec la syntaxe suivante :

```
SELECT  
    column_1,  
    column_2,  
    ...  
FROM  
    table_name;
```

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## SELECT

### ► Exemple:

Soit la relation suivante appelée 'customers':

*Customers(customer\_id, name, adress, website, credit\_limit)*

### ► Créer la table 'customers'

CUSTOMERS
* CUSTOMER_ID
NAME
ADDRESS
WEBSITE
CREDIT_LIMIT

```
CREATE TABLE customers (  
    customer_id NUMBER,  
    name VARCHAR2(255) NOT NULL,  
    adress VARCHAR2(255) NOT NULL,  
    website VARCHAR2(255) NOT NULL,  
    credit_limit NUMBER(3,1) NOT NULL,  
    CONSTRAINT pk_customers PRIMARY KEY customer_id  
);
```

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## SELECT

- **Exemple:**

Soit la relation suivante appelée 'customers':

*Customers(customer\_id, name, adress, website, credit\_limit)*

- Afficher les noms des clients de la table 'customers'

```
SELECT
  name
FROM
  customers;
```

NAME
Kimberly-Clark
Hartford Financial Services Group
Kraft Heinz
Fluor
AECOM
Jabil Circuit
CenturyLink
General Mills
Southern
Thermo Fisher Scientific

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## SELECT

### ► Exemple:

Soit la relation suivante appelée 'customers':

*Customers(customer\_id, name, adress, website, credit\_limit)*

- Afficher les identifiants, les noms et les limites de crédit des clients de la table 'customers'

```
SELECT  
    customer_id,  
    name,  
    credit_limit  
FROM  
    customers;
```

CUSTOMER_ID	NAME	CREDIT_LIMIT
35	Kimberly-Clark	400
36	Hartford Financial Services Group	400
38	Kraft Heinz	500
40	Fluor	500
41	AECOM	500
44	Jabil Circuit	500
45	CenturyLink	500
47	General Mills	600
48	Southern	600
50	Thermo Fisher Scientific	700

- Remarque: Pour interroger les données de plusieurs colonnes, vous spécifiez une liste de noms de colonnes séparés par des virgules.

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## SELECT

### ► Exemple:

Soit la relation suivante appelée 'customers':

*Customers (customer\_id, name, adress, website, credit\_limit)*

- Si on veut renvoyer les données de toutes les colonnes de la table 'customers', on utilise l'astérisque abrégé (\*) :

```
SELECT * FROM customers;
```


CUSTOMER_ID	NAME	ADDRESS	WEBSITE	CREDIT_LIMIT
1	Raytheon	514 W Superior St, Kokomo, IN	http://www.raytheon.com	100
2	Plains GP Holdings	2515 Bloyd Ave, Indianapolis, IN	http://www.plainsallamerican.com	100
3	US Foods Holding	8768 N State Rd 37, Bloomington, IN	http://www.usfoods.com	100
4	AbbVie	6445 Bay Harbor Ln, Indianapolis, IN	http://www.abbvie.com	100
5	Centene	4019 W 3Rd St, Bloomington, IN	http://www.centene.com	100
6	Community Health Systems	1608 Portage Ave, South Bend, IN	http://www.chs.net	100
7	Alcoa	23943 Us Highway 33, Elkhart, IN	http://www.alcoa.com	100
8	International Paper	136 E Market St # 800, Indianapolis, IN	http://www.internationalpaper.com	100
9	Emerson Electric	1905 College St, South Bend, IN	http://www.emerson.com	100
10	Union Pacific	3512 Rockville Rd # 137C, Indianapolis, IN	http://www.up.com	200



# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## SELECT

- La syntaxe de commande SELECT :



```
SELECT           [liste des attributs]
FROM            [liste de table]
WHERE           [condition de recherche]
GROUP BY       [attributs de partitionnement]
HAVING         [condition de groupe]
ORDER BY      [liste de colonnes (ASC/DESC)]
```

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## WHERE

- La clause **WHERE** spécifie une condition de recherche pour les lignes renvoyées par la commande **SELECT**. Ce qui suit illustre la syntaxe de la clause **WHERE** :

```
SELECT
    column_list
FROM
    table_name
WHERE
    search_condition;
```

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## WHERE: les opérateurs

La condition de recherche (qualification) est spécifiée après la clause **WHERE** par un prédicat:

- un prédicat d'égalité ou d'inégalité (=, <> ou !=, <=, >=)
- un prédicat **LIKE**
- un prédicat **BETWEEN**
- un prédicat **IN**
- un test de valeur **IS NULL/ IS NOT NULL**
- un prédicat **EXISTS**
- un prédicat **ALL** ou **ANY** ou **SOME**

Un prédicat composé est construit à l'aide des connecteurs **AND**, **OR** et **NOT**

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## Opérateur LIKE

- L'opérateur LIKE est utilisé dans la clause WHERE des requêtes SQL. Ce mot-clé permet d'effectuer une recherche sur un modèle particulier. Il est par exemple possible de rechercher les enregistrements dont la valeur d'une colonne commence par telle ou telle lettre.

- **Syntaxe**

```
SELECT
    column_list
FROM
    table_name
WHERE
    column LIKE modèle;
```

- Le prédicat LIKE compare une chaîne de caractère avec un modèle
- ( ) remplace n'importe quel caractère (un seul caractère)
- (%) remplace n'importe quelle suite de caractères

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## Opérateur LIKE

% chaîne  
\_ un caractère

Dans cet exemple le “modèle” n’a pas été défini, mais il ressemble très généralement à l’un des exemples suivants:

- ▶ **LIKE ‘%a’** : le caractère “%” est un caractère joker qui remplace tous les autres caractères. Ainsi, ce modèle permet de rechercher toutes les chaînes de caractère qui se termine par un “a”.
- ▶ **LIKE ‘a%’** : ce modèle permet de rechercher toutes les lignes de “colonne” qui commence par un “a”.
- ▶ **LIKE ‘%a%’** : ce modèle est utilisé pour rechercher tous les enregistrement qui utilisent le caractère “a”.
- ▶ **LIKE ‘pa%on’** : ce modèle permet de rechercher les chaînes qui commence par “pa” et qui se terminent par “on”, comme “pantalon” ou “pardon”.
- ▶ **LIKE ‘a\_c’** : le caractère “\_” (underscore) peut être remplacé par n’importe quel caractère, mais un seul caractère uniquement (alors que le symbole pourcentage “%” peut être remplacé par un nombre incalculable de caractères). Ainsi, ce modèle permet de retourner les lignes “arc” ou “abc”.

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## Opérateur LIKE

- Imaginons une table “client” qui contient les enregistrement d'utilisateurs :

id	nom	ville
2	Odette	Nice
3	Vivien	Nantes

- On souhaite obtenir uniquement les villes des clients qui commencent par un ‘N’:

```
SELECT
    *
FROM
    client
WHERE
    ville LIKE 'N%';
```

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## Opérateur BETWEEN

- L'opérateur **BETWEEN** est utilisé dans une requête SQL pour sélectionner un intervalle de données dans une requête utilisant WHERE. L'intervalle peut être constitué de chaînes de caractères, de nombres ou de dates.

- **Syntaxe**

```
SELECT
    column_list
FROM
    table_name
WHERE
    column BETWEEN value1 AND value2;
```

- La requête suivante retournera toutes les lignes dont la valeur de la colonne "column" sera comprise entre **value1** et **value2**.

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## Opérateur BETWEEN

- **Exemple:** Imaginons une table “utilisateur” qui contient les membres d’une application en ligne.

id	nom	date_inscription
1	Maurice	2012-03-02
2	Simon	2012-03-05

id	nom	date_inscription
3	Chloé	2012-04-14
4	Marie	2012-04-15

- Si l’ont souhaite obtenir les membres qui se sont inscrit entre le 1 avril 2012 et le 20 avril 2012 :



# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## Opérateur IN

- L'opérateur logique **IN** dans SQL s'utilise avec la commande **WHERE** pour vérifier si une colonne est égale à une des valeurs comprise dans set de valeurs déterminés. C'est une méthode simple pour vérifier si une colonne est égale à une valeur OU une autre valeur OU une autre valeur et ainsi de suite, sans avoir à utiliser de multiple fois l'opérateur **OR**.

- **Syntaxe**

```
SELECT  
    column  
FROM  
    table_name  
WHERE  
    column IN (value1, value2,...) ;
```

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## Opérateur IN

- **Exemple:** soit une table “adresse” qui contient une liste d'adresse associée à des utilisateurs d'une application.

id	id_utilisateur	addr_rue	addr_code_postal	addr_ville
1	23	35 Rue Madeleine Pelletier	25250	Bourmois
2	43	21 Rue du Moulin Collet	75006	Paris
3	65	28 Avenue de Cornouaille	27220	Mousseaux-Neuville
4	67	41 Rue Marcel de la Provoté	76430	Graimbouville
5	68	18 Avenue de Navarre	75009	Paris

- Si l'ont souhaite obtenir les enregistrements des adresses de 'Paris' et de 'Graimbouville'.

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## Opérateur IN

### ► Exemple:

```
SELECT
    *
FROM
    adresse
WHERE
    addr_ville IN ('Paris', 'Graimbouville') ;
```

id	id_utilisateur	addr_rue	addr_code_postal	addr_ville
2	43	21 Rue du Moulin Collet	75006	Paris
4	67	41 Rue Marcel de la Provoté	76430	Graimbouville
5	68	18 Avenue de Navarre	75009	Paris

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## Opérateur IS NULL/IS NOT NULL

- ▶ Dans le langage SQL, l'opérateur IS permet de filtrer les résultats qui contiennent la valeur NULL. Cet opérateur est indispensable car la valeur NULL est une valeur inconnue et ne peut par conséquent pas être filtrée par les opérateurs de comparaison (cf. égal, inférieur, supérieur ou différent).
- ▶ **Syntaxe:** Pour filtrer les résultats où les champs d'une colonne sont **NULL** il convient d'utiliser la syntaxe suivante:

```
SELECT
    *
FROM
    table_name
WHERE
    column IS NULL ;
```

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## Opérateur IS NULL/IS NOT NULL

- **Syntaxe:** Pour filtrer les résultats et obtenir uniquement les enregistrements qui ne sont pas null, il convient d'utiliser la syntaxe suivante:

```
SELECT
    *
FROM
    table_name
WHERE
    column IS NOT NULL ;
```

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## Opérateur IS NULL/IS NOT NULL

- **Exemple:** Soit une table contenant les utilisateurs. Cette table possède 2 colonnes pour associer les adresses de livraison et de facturation à un utilisateur (grâce à une clé étrangère). Si cet utilisateur n'a pas d'adresse de facturation ou de livraison, alors le champ reste à NULL.

id	nom	date_inscription	fk_adresse_livraison_id	fk_adresse_facturation_id
23	Grégoire	2013-02-12	12	12
24	Sarah	2013-02-17	NULL	NULL
25	Anne	2013-02-21	13	14
26	Frédérique	2013-03-02	NULL	NULL

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## Opérateur IS NULL/IS NOT NULL

- Il est possible d'obtenir la liste des utilisateurs qui ne possèdent pas d'adresse de livraison

```
SELECT *  
  
FROM  
    utilisateur  
WHERE  
    fk_adresse_livraison_id IS NULL ;
```

id	nom	date_inscription	fk_adresse_livraison_id	fk_adresse_facturation_id
24	Sarah	2013-02-17	NULL	NULL
26	Frédérique	2013-03-02	NULL	NULL

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## Les fonctions d'agrégation

Les fonctions d'agrégation sont des fonctions idéales pour effectuer quelques statistiques de bases sur des tables. Les principales fonctions sont les suivantes :

- AVG() pour calculer la moyenne sur un ensemble d'enregistrement.
- COUNT() pour compter le nombre d'enregistrement sur une table ou une colonne.
- MAX() pour récupérer la valeur maximum d'une colonne sur un ensemble de ligne.
- MIN() pour récupérer la valeur minimum de la même manière que MAX().
- SUM() pour calculer la somme sur un ensemble d'enregistrement.

Syntaxe:

```
SELECT aggregation function(column) FROM table_name;
```



# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## GROUP BY

- La clause **GROUP BY** permet de partitionner une table en plusieurs groupes

```
SELECT
    column1, aggregation function(column2)
FROM
    Table_name
GROUP BY column1;
```

- Exemple:

**Employee**

EmployeeID	Ename	DeptID	Salary
1001	John	2	4000
1002	Anna	1	3500
1003	James	1	2500
1004	David	2	5000
1005	Mark	2	3000
1006	Steve	3	4500
1007	Alice	3	3500

```
SELECT DeptID, AVG(Salary)
FROM Employee
GROUP BY DeptID;
```

GROUP BY  
Employee Table  
using DeptID

DeptID	AVG(Salary)
1	3000.00
2	4000.00
3	4250.00

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## HAVING

- La clause **HAVING** permet de spécifier une condition de restriction des groupes. On utilise la clause **HAVING** avec la clause **GROUP BY** pour filtrer les groupes de résultats qui ne satisfassent pas une condition précise

Syntaxe:

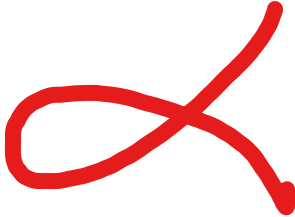
```
SELECT
    column1, aggregation function(column2)
FROM
    Table
GROUP BY
    column1
HAVING
    condition;
```

- **HAVING** est différent de **WHERE**. La clause **HAVING** applique des conditions sur les groupes de résultat créés par **GROUP BY** alors que **WHERE** applique des conditions sur les lignes individuelles.

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## HAVING

Exemple:



```
SELECT
    DeptID, AVG(Salary)
FROM
    Employee
GROUP BY
    DeptID
HAVING
    AVG(Salary) > 3000;
```

**Employee**

EmployeeID	Ename	DeptID	Salary
1001	John	2	4000
1002	Anna	1	3500
1003	James	1	2500
1004	David	2	5000
1005	Mark	2	3000
1006	Steve	3	4500
1007	Alice	3	3500

**SELECT** DeptID, AVG(Salary)  
**FROM** Employee  
**GROUP BY** DeptID;

**GROUP BY**  
Employee Table  
using DeptID

DeptID	AVG(Salary)
1	3000.00
2	4000.00
3	4250.00

**SELECT** DeptID, AVG(Salary)  
**FROM** Employee  
**GROUP BY** DeptID  
**HAVING** AVG(Salary) > 3000;

**HAVING**

DeptID	AVG(Salary)
2	4000.00
3	4250.00

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## ORDER BY

- La clause **ORDER BY** permet de trier les résultats par ordre croissant [ASC] ou décroissant [DESC],

Syntaxe:

```
SELECT
    column1, column2
FROM
    Table
ORDER BY
    column1 [ASC/DESC]
```

- Par défaut les résultats sont classés par ordre ascendant, toutefois il est possible d'inverser l'ordre en utilisant le suffixe **DESC** après le nom de la colonne. Par ailleurs, il est possible de trier sur plusieurs colonnes en les séparant par une virgule. Une requête plus élaborée ressemblerait à cela :

```
SELECT
    column1, column2, column3
FROM
    Table
ORDER BY
    column1 DESC, column2 ASC
```

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## ORDER BY

- Pour l'ensemble de nos exemples, nous allons prendre la table “utilisateur”, qui contient les données suivantes :

id	nom	prenom	date_inscription	tarif_total
5	Dubois	Simon	2012-02-23	27
4	Dubois	Chloé	2012-02-16	98
2	Dupond	Fabrice	2012-02-07	65
3	Durand	Fabienne	2012-02-13	90
1	Durand	Maurice	2012-02-05	145

```
SELECT * FROM utilisateur
ORDER BY
    nom, date_inscription DESC
```

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## JOINTURE

- ▶ Les jointures en SQL permettent d'associer plusieurs tables dans une même requête. Cela permet d'exploiter la puissance des bases de données relationnelles pour obtenir des résultats qui combinent les données de plusieurs tables de manière efficace.

```
SELECT
    column_list
FROM
    table1 alias1, table2 alias2
WHERE
    alias1.attribut=alias2.attribut
```

### Exemple:


Soient les 3 relations suivantes:

- ▶ Agence (NumAg, NomAg, VilleAg)
- ▶ Client (NumCl, NomCl, VilleCl)
- ▶ Compte (NumC, #NumAG, #NumCl, Solde)

# LANGAGE D'INTERROGATION DE DONNÉES (LID)

## JOINTURE

- Donner le total de soldes des comptes des clients de la ville de 'sousse'



```
SELECT
    SUM(solde)
FROM
    Compte C, Client C1
WHERE
    C1.VilleC1 = 'sousse' AND C1.NumC1=C.NumC1;
```