Structured Query Language

SQL

SQL in a nutshell

1974 création chez IBM

1986 Norme ANSI puis ISO

Plusieurs normalisations

- Normes SQL-92 (SQL2)
- Normes SQL-99 (SQL3)
- Normes SQL-2011

Differents dialectes

- Oracle database, postgresql, mysql, mariaDB, sqlite…

SQL in a nutshell

SQL manipule des tables

- Table ≈ Relation
- Colonne ≈ Attribut
- Ligne ≈ tuple

Une table n'est pas un ensemble, il peut y avoir plusieurs lignes identiques

Gestion des doublons

- SQL construit des multi-ensembles (et non des ensembles comme en algèbre relationnelle).
- La même ligne peut apparaître plusieurs fois dans une table.
- Lorsqu'une requête SQL construit une nouvelle table elle n'élimine pas automatiquement les doublons.

SQL in a nutshell

- Langage de définition de données :
  CREATE, ALTER, DROP, RENAME
- Langage de manipulation de données
  INSERT, UPDATE, DELETE, SELECT
- Langage de contrôle de données
  GRANT, REVOKE
- Langage de contrôle des transactions
  SET TRANSACTION, COMMIT, ROLLBACK

Data Manipulation Language: INSERT, UPDATE, DELETE, SELECT

SQL DML

SELECT

LA commande SQL qui permet de consulter les données des tables d'une base de données relationnelle.

Le résultat d'une commande SELECT est zéro ou plusieurs lignes, avec éventuellement des répétitions.

SELECT

- Une commande SELECT décrit un jeu de résultats voulus, et non la manière de les obtenir
- Le système de gestion de base données (SGBD) transforme la requête en un plan d'exécution de requête, qui peut varier d'un système à l'autre

SELECT

- Opérations algébriques
  - Sélection
  - Projection
  - Jointure
  - Renommage des attributs
- Autres opérations
  - Tri
  - Renommage temporaire
  - Partitionnement

SELECT

SELECT	Spécification du schéma du résultat
FROM	Spécification des tables sur lesquelles porte l'ordre
WHERE	Filtre portant sur les données
GROUP BY	Définition de groupes (partition)
HAVING	Filtre portant sur les groupes
ORDER BY	Tri des données

SELECT

SELECT E1 FROM E2 WHERE E3 ORDER BY E4

E2 permet de calculer une table T

E3 permet de filtrer les lignes de T sur une condition

E4 permet d'ordonner les lignes

E1 permet de projeter pour ne garder qu'une partie des colonnes

SELECT

SELECT E1 **FROM E2** WHERE E3 ORDER BY E4

E2 est la ou les tables dans lesquelles on va chercher les données et comment on les assemble (jointure, produit cartésien, jointure naturelle…..)

Si on n'utilise qu'une seule table, E2 est simplement le nom de la table

SELECT

SELECT E1 **FROM E2** WHERE E3 ORDER BY E4

marque(IdM, NomM, Classe, Pays, Prop)
enreg (NumE, IdM, Pays, DateE, IdDeposant)

SELECT E1
FROM marque **NATURAL JOIN** enreg
WHERE E3 ORDER BY E4;

Jointure naturelle

La jointure se fait sur les deux attributs de même nom, Pays et IdM. On construit donc des tuples de longueur 8.

SELECT

SELECT E1 **FROM E2** WHERE E3 ORDER BY E4

marque(IdM, NomM, Classe, Pays, Prop)
enreg (NumE, IdM, Pays, DateE, IdDeposant)

SELECT E1
FROM marque **,** enreg
WHERE E3 ORDER BY E4;

Produit cartésien

On construit donc des tuples de longueur 10.

SELECT

SELECT E1 **FROM E2** WHERE E3 ORDER BY E4

marque(IdM, NomM, Classe, Pays, Prop)
societe (IdM, IdS, Pays)

SELECT E1
FROM marque **JOIN** societe **ON** marque.Prop = societe.IdS
WHERE E3 ORDER BY E4;

Jointure (non naturelle)

On joint un tuple s de societe avec un tuple m de marque ssi m.Prop=s.IdS
Dans le résultat les deux colonnes sont gardées

SELECT

SELECT E1 **FROM E2** WHERE E3 ORDER BY E4

marque(IdM, NomM, Classe, Pays, Prop)
societe (IdM, IdS, Pays)

SELECT E1
FROM marque **JOIN** societe **ON**
marque.Prop = societe.IdS **AND** marque.Pays=societe.Pays
WHERE E3 ORDER BY E4;

SELECT

SELECT E1 **FROM E2** WHERE E3 ORDER BY E4

Renommage des tables :

SELECT E1
FROM marque M1 JOIN marque M2 ON
(M1. NomM=M2. NomM);

SELECT E1
FROM marque **AS** M1 JOIN marque **AS** M2
USING (NomM);

SELECT

SELECT E1 **FROM E2** WHERE E3 ORDER BY E4

À suivre…

- Autres types de jointures
- Requêtes imbriquées

SELECT

SELECT E1 FROM E2 **WHERE E3** ORDER BY E4

E3 : filtre des lignes par sélection sur une condition
Sélection comme définie en Algèbre Relationnelle

SELECT E1 FROM marque WHERE Classe=14;

SELECT

SELECT E1 FROM E2 **WHERE E3** ORDER BY E4

S'il y a des attributs de même nom, il faut les distinguer :

```
SELECT E1
FROM marque JOIN societe ON marque.prop=societe.IdS
WHERE marque.Pays<> societe.Pays
```

```
SELECT E1
FROM marque M JOIN societe S ON M.prop=S.IdS
WHERE M.Pays<>S.Pays
```

SELECT

SELECT E1 FROM E2 **WHERE E3** ORDER BY E4

E3 peut utiliser :

- tous les attributs de la table construite dans le FROM
- les opérateurs logiques: AND, OR, NOT
- tous les opérateurs compatibles avec le type des attributs, par exemple :
  - Comparateurs: =, <>, <, >, <=, >=
  - Opérateurs arithmétiques: +, *, …
  - Concaténations de chaînes: 'foo'||'bar' a pour valeur 'foobar'
- … (à suivre)

SELECT

SELECT E1 FROM E2 WHERE E3 **ORDER BY E4**

E4 permet d'ordonner les lignes de la table résultat selon les valeurs d'attributs, par ordre croissant ou décroissant

SELECT E1 FROM marque ORDER BY Classe;

SELECT E1 FROM marque ORDER BY Classe DESC;

SELECT E1 FROM marque ORDER BY Nom, Classe;

SELECT

SELECT E1 FROM E2 WHERE E3 ORDER BY E4

E1 définit une projection :

SELECT NomM FROM marque;

Les doublons ne sont pas supprimés

SELECT **DISTINCT** NomM FROM marque;

Gestion des doublons

Pour éliminer les doublons :

SELECT **DISTINCT** nom FROM marque;

L'usage de DISTINCT a un coût important :
il faut stocker toute la table et la trier.

SELECT

SELECT E1 FROM E2 WHERE E3 ORDER BY E4

Projection sur plusieurs attributs :

SELECT IdM , Pays FROM marque;

SELECT

SELECT E1 FROM E2 WHERE E3 ORDER BY E4

Pour ne pas projeter :

SELECT * FROM marque;

La table résultat a le même schéma que celle construite en E2

SELECT

SELECT E1 FROM E2 WHERE E3 ORDER BY E4

La projection sur E1 est la dernière opération réalisée :

SELECT NomM FROM marque ORDER BY IdM;

est possible

SELECT

SELECT E1 FROM E2 WHERE E3 ORDER BY E4

Renommage des attributs dans la table résultat :

SELECT NomM **AS** NomMarque , Prop **AS** Proprietaire
FROM marque;

SELECT

SELECT E1 FROM E2 WHERE E3 ORDER BY E4 **;**

Terminateur ;

En principe toute instruction doit être terminée par ";"

Certains SGBD permettent de l'omettre sur la dernière instruction

DELETE

Suppression de tuples dans une table

DELETE FROM *table* WHERE *condition*

DELETE FROM marque
WHERE NomM='Channel' AND Classe='14';

- Si une ligne est présente en plusieurs exemplaires DELETE supprime tous les exemplaires
- DELETE FROM marque; vide la table marque

INSERT

Insertion de tuples dans une table

INSERT INTO *table* (*column1*, *column2*, ...)
VALUES (*value1*, *value2*, ...) ;

INSERT INTO marque VALUES (1, 'Coca', 12, 'Fr', 123);

On peut ne pas spécifier tous les attributs sauf si spécifiquement interdit:
INSERT INTO marque (IdM,NomM) VALUES (1,'Coca');

INSERT

Combinaison de INSERT et SELECT:
Insertion de données provenant d'une autre table

data(numero, nom, classe, pays)
marque(idM, nom, classe, pays)

INSERT INTO marque
SELECT numero, nom, classe, pays FROM data
WHERE data.nom NOT IN
(SELECT NomM FROM marque);

Insertion de données sans INSERT

Import de données possible depuis un fichier (csv, excel…), dépendant du SGBD

UPDATE

UPDATE *table*
SET *column1* = *value1*, *column2* = *value2*, ...
WHERE *condition* ;

UPDATE marque
SET NomM='BeauNom'
WHERE marque.IdM=12;

Ne pas oublier la condition car sinon le update porte sur tous les tuples

SELECT (suite)

COUNT, SUM, AVG, MIN, MAX

Les fonctions d'agrégation agrègent en une seule valeur (un agrégat) toutes les valeurs d'une colonne

article(Id,Nom,Prix)

SELECT AVG(Prix) FROM article;

SELECT COUNT(*) FROM article;

SELECT COUNT(DISTINCT Nom) FROM article;

SELECT (suite)

COUNT, SUM, AVG, MIN, MAX

Les fonctions d'agrégation ne peuvent pas être utilisées dans la clause WHERE :

SELECT id FROM article WHERE Prix = MAX(Prix);

incorrect

SELECT id FROM article
WHERE PRIX = (SELECT MAX(Prix) FROM article);

À suivre...

SELECT

SELECT E1 FROM E2 WHERE E3
GROUP BY E5 ORDER BY E4

E5 est une liste d'attributs de la table construite en E2

Partitionnement des lignes en regroupant dans une même classe d'équivalence (dite regroupement) toutes les lignes qui coïncident sur tous les attributs dans E5

Il est alors possible d'agréger des résultats sur une classe d'équivalence

SELECT

SELECT E1 FROM E2 WHERE E3
GROUP BY E5 ORDER BY E4

movie(title,year,length,studioName,producer)

nombre total de minutes de films produits par chaque studio

SELECT studioName, **SUM**(length)
FROM movie GROUP BY studioName;

Seuls les agrégats et les attributs mentionnés en E5 peuvent apparaître en E1.

SELECT

SELECT E1 FROM E2 WHERE E3
GROUP BY E5 ORDER BY E4

movie(title,year,length,studioName,producer)

SELECT studioName, SUM(length), **year**
FROM movie GROUP BY studioName;

incorrect

Seuls les agrégats et les attributs mentionnés en E5 peuvent apparaître en E1.

SELECT

SELECT E1 FROM E2 WHERE E3
GROUP BY E5 ORDER BY E4

movie(title,year,length,studioName,producer)

SELECT studioName, SUM(length)
FROM movie
GROUP BY studioName, **year**;

Correct mais quelle signification ???

SELECT

SELECT E1 FROM E2 WHERE E3
GROUP BY E5 ORDER BY E4

movie(title,year,length,studioName,producer)

SELECT year, SUM(length)
FROM movie GROUP BY year **WITH ROLLUP**;

year	SUM(length)
2000	4525
2001	3010
NULL	7535

postgres

SELECT

SELECT E1 FROM E2 WHERE E3
GROUP BY E5 **HAVING E6** ORDER BY E4

E6 est une condition évaluée après le regroupement.

On peut utiliser dans E6 les fonctions d'agrégation (ce qui n'est pas possible dans E3)

SELECT

SELECT E1 FROM E2 WHERE E3
GROUP BY E5 **HAVING E6** ORDER BY E4

employe(nom, fonction, salaire)

Liste des salaires moyens par fonction pour les fonctions occupées par plus de deux employés.

SELECT fonction, COUNT(*), AVG(salaire)
FROM employe
GROUP BY fonction
HAVING **COUNT**(*) > 2;

fonction	COUNT(*)	AVG(salaire)
administratif	4	12 375
commercial	5	21 100

SELECT

Ordre d'écriture des clauses:

SELECT E1
FROM E2
Les deux seules obligatoires selon la norme
WHERE E3
GROUP BY E5
HAVING E6
ORDER BY E4

SELECT 1;
SELECT current_date;

hors norme

SELECT

Ordre d'exécution des clauses:

FROM E2	jointures
WHERE E3	sélection des tuples
GROUP BY E5	création des regroupements
HAVING E6	sélection des tuples sur regroupements
ORDER BY E4	tri
SELECT E1	projection

Pour optimiser, l'exécution de E2 et E3 peut être conjointe.
On ne se soucie pas de l'ordre d'exécution des clauses lorsqu'on écrit une requête.
Une requête décrit le résultat souhaité, pas comment l'obtenir.

Intersection, Union, Difference

Nom des marques déclarées
à la fois dans la Classe 14 et la Classe 10

(SELECT NomM FROM marque WHERE Classe=14)
INTERSECT
(SELECT NomM FROM marque WHERE Classe=10)

Intersection, Union, Difference

Nom des marques déclarées
dans la Classe 14 ou la Classe 10

(SELECT NomM FROM marque WHERE Classe=14)
UNION
(SELECT NomM FROM marque WHERE Classe=10)

Intersection, Union, Difference

Identifiant des marques
n'appartenant pas à la Classe 10

(SELECT IdM FROM marque)
EXCEPT
(SELECT IdM FROM marque WHERE Classe=10)

EXCEPT est parfois **MINUS** (e.g. ORACLE)

Gestion des doublons

UNION, INTERSECT et EXCEPT sont des opérations ensemblistes qui éliminent les doublons.

UNION ALL, INTERSECT ALL et EXCEPT ALL permet de travailler sur des multi-ensembles.

R EXCEPT ALL S
élimine autant d'occurrences d'un tuple t dans R que celui-ci à d'occurrences dans S

Sous-requête

Une sous-requête est une requête SQL qui apparait dans une requête SQL.

Où et comment dépendent du type de résultat attendu de la sous requête.

Sous-requête

Le résultat d'une requête est toujours une table composée de lignes et de colonnes, mais il y a des cas particuliers:

- Résultat vide
- Résultat composé d'une seule colonne
- Résultat composé d'une ligne
- Résultat composé d'une seule valeur (une seule ligne ET une seule colonne)

Sous-requête

qui produit une valeur unique

Une telle requête peut être utilisée partout où l'on pourrait utiliser une constante :

- Dans une clause SELECT
- Dans une clause WHERE
- Dans une clause HAVING

Sous-requête

qui produit une valeur unique dans un WHERE

le nom de la marque enregistrée sous le numéro 17

```
SELECT Nom
FROM marque
WHERE IdM = (SELECT IdM FROM enr WHERE NumE = 17)
```

Sous-requête

qui produit une valeur unique dans un HAVING

le propriétaire unique de toutes les marques de la Classe 12 – s'il existe

```
SELECT Prop, count(IdM) as Marques
FROM marque
WHERE Classe = 12
GROUP BY Prop
HAVING Marques = (SELECT count(IdM) FROM marque
                  WHERE Classe = 12)
```

Sous-requête
qui produit une valeur unique dans un SELECT

Combien de sociétés ne possèdent aucune marque ?

```
SELECT
    (SELECT COUNT(DISTINCT idS) FROM societe) –
    (SELECT COUNT(DISTINCT prop) FROM marque) ;
```

Sous-requête
qui produit une liste de valeurs (une colonne)

Une telle requête peut être utilisée dans :

- IN
- NOT IN
- ANY/SOME
- ALL

Sous-requête
qui produit une liste de valeurs dans un IN ou NOT IN

Les sociétés qui ne possèdent pas de marque

```
SELECT IdS
FROM societe
WHERE IdS NOT IN
    (SELECT Prop FROM marque);
```

Sous-requête
qui produit une liste de valeurs dans un ANY/SOME

Tous les propriétaires, sauf le plus gros

```
SELECT Prop FROM marque
GROUP BY Prop
HAVING count (IdM) < ANY (SELECT count(IdM) FROM marque
                          GROUP BY Prop)
```

la condition doit être vraie pour au moins une valeur de la liste
IN est équivalent à **= ANY**

Sous-requête
qui produit une liste de valeurs dans un ALL

Le plus gros propriétaire

```
SELECT Prop
FROM marque
GROUP BY Prop
HAVING count (IdM) >= ALL
                    (SELECT count(IdM)
                    FROM marque
                    GROUP BY Prop)
```

Sous-requête
quelconque

Une telle requête peut être utilisée dans un FROM :

De la manière dont on utiliserait une table.
Cela nécessite un parenthésage et un alias

```
SELECT * FROM table1, (requeteselect) as table2 WHERE …
```

Sous-requête
quelconque

Une telle requête peut être utilisée dans un WHERE ou un HAVING en utilisant EXISTS :

WHERE EXISTS (requeteselect)
est vrai si requeteselect retourne au moins un résultat

Sous-requête
corrélée

Il est possible d'utiliser dans une sous-requête un attribut qui provient de la requête principale :

```
SELECT title FROM movie AS Old
    WHERE year < ANY
        (SELECT year FROM movie
         WHERE title = Old.title);
```

Un nom de film sera listé une fois de moins que le nombre de films portant ce nom.
La sous requête corrélée est exécutée de multiples fois.

Jointures (suite et fin)

- CROSS JOIN = produit cartésien
- (INNER) JOIN
  Cas particulier: NATURAL JOIN
- OUTER JOIN

CROSS JOIN

SELECT * FROM t1 **CROSS JOIN** t2;

SELECT * FROM t1 **,** t2;

t1

num	name1
1	a
2	b
3	c

t2

num	name2
1	x
3	y
5	z

num	name1	num	name2
1	a	1	x
1	a	3	y
1	a	5	z
2	b	1	x
2	b	3	y
2	b	5	z
3	c	1	x
3	c	3	y
3	c	5	z

(INNER) JOIN

SELECT * FROM t1 **INNER JOIN** t2 **ON** t1.num = t2.num;

SELECT * FROM t1 **JOIN** t2 **ON** t1.num = t2.num;

SELECT * FROM t1 **,** t2 **WHERE** t1.num = t2.num;

Produit cartésien + Sélection

t1

num	name1
1	a
2	b
3	c

t2

num	name2
1	x
3	y
5	z

num	name1	num	name2
1	a	1	x
3	c	3	y

(INNER) JOIN

t1

num	name1
1	a
2	b
3	c

t2

num	name2
1	x
3	y
5	z

SELECT * FROM t1 JOIN t2
ON t1.num = t2.num;

num	name1	num	name2
1	a	1	x
3	c	3	y

SELECT * FROM t1 JOIN t2
USING num;

num	name1	name2
1	a	x
3	c	y

(INNER) JOIN

t1

num	name1
1	a
2	b
3	c

t2

num	name2
1	x
3	y
5	z

SELECT * FROM t1 **NATURAL JOIN** t2

num	name1	name2
1	a	x
3	c	y

LEFT (OUTER) JOIN

SELECT * FROM t1 **LEFT JOIN** t2 **ON** t1.num = t2.num;

t1

num	name1
1	a
2	b
3	c

t2

num	name2
1	x
3	y
5	z

num	name1	num	name2
1	a	1	x
2	b	null	null
3	c	3	y

tous les tuples de la table de gauche sont conservés, complétés par des null s'ils ne sont joignables avec aucun tuple de la table de droite

LEFT (OUTER) JOIN

SELECT * FROM t1 **LEFT JOIN** t2 **USING** num;

t1

num	name1
1	a
2	b
3	c

t2

num	name2
1	x
3	y
5	z

num	name1	name2
1	a	x
2	b	null
3	c	y

tous les tuples de la table de gauche sont conservés, complétés par des null s'ils ne sont joignables avec aucun tuple de la table de droite

RIGHT (OUTER) JOIN

SELECT * FROM t1 **RIGHT JOIN** t2 **ON** t1.num = t2.num;

t1

num	name1
1	a
2	b
3	c

t2

num	name2
1	x
3	y
5	z

num	name1	num	name2
1	a	1	x
3	c	3	y
null	null	5	z

tous les tuples de la table de droite sont conservés, complétés par des null s'ils ne sont joignables avec aucun tuple de la table de gauche

RIGHT (OUTER) JOIN

SELECT * FROM t1 **RIGHT JOIN** t2 **USING** num;

t1

num	name1
1	a
2	b
3	c

t2

num	name2
1	x
3	y
5	z

num	name1	name2
1	a	x
3	c	y
5	null	z

tous les tuples de la table de gauche sont conservés, complétés par des null s'ils ne sont joignables avec aucun tuple de la table de droite

FULL (OUTER) JOIN

SELECT * FROM t1 **FULL JOIN** t2 **ON** t1.num = t2.num;

t1

num	name1
1	a
2	b
3	c

t2

num	name2
1	x
3	y
5	z

num	name1	num	name2
1	a	1	x
2	b	null	null
3	c	3	y
null	null	5	z

On complète les tuples non joignables par des valeurs null à droite et à gauche

FULL (OUTER) JOIN

SELECT * FROM t1 **FULL JOIN** t2 **USING** num;

t1

num	name1
1	a
2	b
3	c

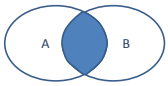
t2

num	name2
1	x
3	y
5	z

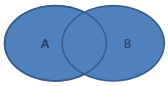
num	name1	name2
1	a	x
2	b	null
3	c	y
5	null	z

On complète les tuples non joignables par des valeurs null à droite et à gauche

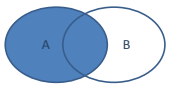
JOIN summary



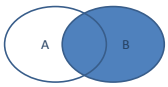
INNER JOIN



FULL JOIN



LEFT JOIN



RIGHT JOIN

FAIRE RAPPEL SOUS-REQUÊTES

Vues

- Les tables (créées avec CREATE TABLE) sont des relations persistantes :
  elles sont stockées physiquement et existent jusqu'à leur suppression explicite.
- Les vues sont des relations virtuelles qui peuvent être utilisées dans des requêtes comme des tables mais qui ne sont pas persistantes :
  elles n'ont pas d'existence physique, elles ne sont pas stockées.

Création et suppression d'une vue

CREATE VIEW nom de la vue
AS requête Q ;

Q est la définition de la vue : chaque fois que la vue est utilisée (e.g., dans une clause SELECT), SQL se comporte comme si Q était exécutée à ce moment-là.

DROP VIEW nom de la vue ;

Création et utilisation d'une vue

```
CREATE VIEW paramountmovie AS
        SELECT title, year FROM movie
        WHERE studioname='paramount';

SELECT title FROM paramountmovie
WHERE year=1979;
```

la relation paramountmovie ne contient aucun tuple; les tuples recherchés sont ceux de la table movie

Création et utilisation d'une vue

movie(title,year,length,studioname,idproducer)
movieExec(id,name,address,networth)

```
CREATE VIEW movieprod(movietitle,prodname) AS
  SELECT title, name
  FROM movie M JOIN movieExec E
  ON M.idproducer=E.id;

SELECT prodname FROM movieprod
WHERE movietitle = 'Gone with the wind';
```

Création et utilisation d'une vue

```
CREATE VIEW movieprod(movietitle,prodname) AS
  SELECT title, name
  FROM movie M JOIN movieExec E ON M.idproducer=E.id;

SELECT prodname
FROM movieprod
WHERE movietitle = 'Gone with the wind';
```

Équivaut à :

```
SELECT name AS prodname
FROM movie M JOIN movieExec E ON M.idproducer=E.id;
WHERE title = 'Gone with the wind';
```

Modification d'une vue

- En général il n'est pas possible de modifier une vue car on ne sait pas comment stocker l'information
- Il est possible de modifier une vue dans des cas restreints:
  - Vue construite par la sélection (avec SELECT et non SELECT DISTINCT) de certains attributs d'une relation R
  - dans laquelle la clause WHERE n'utilise pas R dans une sous-requête
  - et où les attributs de la clause SELECT doivent être suffisants pour pouvoir compléter le tuple avec des valeurs NULL

Modification d'une table via une vue

```
CREATE VIEW paramountmovie AS
      SELECT title, year FROM movie
      WHERE studioname='paramount';

INSERT INTO paramountmovie VALUES('Star Treck,1979);
```

requête correcte d'un point SQL mais le nouveau tuple dans movie aura NULL et non 'paramount' comme valeur de studioname !!

Modification d'une table via une vue

```
CREATE VIEW paramountmovie AS
      SELECT studioname, title, year FROM movie
      WHERE studioname='paramount';

INSERT INTO paramountmovie
VALUES('paramount', 'Star Treck, 1979);
```

Tuple inséré dans movie :

title	year	length	studioName	idProducer
Star Treck	1979	0	paramount	null

WITH

- Permet de créer une ou plusieurs tables temporaires dites tables d'expression communes (CTE) à usage unique, i.e qui ne peuvent être utilisées que dans la requête suivante
- Une clause WITH peut comporter plusieurs ordres dits auxiliaires SELECT, INSERT, UPDATE ou DELETE
- L'ordre dit primaire qui suit une clause WITH peut être SELECT, INSERT, UPDATE ou DELETE

WITH

```
WITH maCTE AS (
 SELECT * FROM maTable )
SELECT * FROM maCTE;
```

WITH

```
WITH maCTE AS (
  SELECT * FROM maTable )
, maCTE2 AS (
  SELECT * FROM maTable )

SELECT *
FROM maCTE INNER JOIN maCTE2
ON maCTE.Clé = maCTE2.Clé2;
```

WITH

```
WITH maCTE AS (
  SELECT * FROM maTable )
, maCTE2 AS (
  SELECT * FROM maCTE INNER JOIN maTable2
  ON maCTE.Clé = maTable2.Clé2 )

SELECT * FROM maCTE2;
```

WITH

```
WITH regional_sales AS (
   SELECT region, SUM(amount) AS total_sales
   FROM orders
   GROUP BY region )
, top_regions AS (
   SELECT region
   FROM regional_sales
   WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)

SELECT region, product, SUM(quantity) AS product_units,
   SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

WITH RECURSIVE

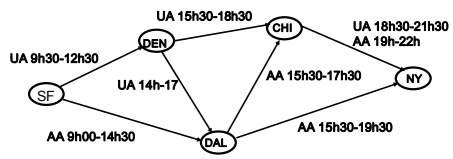
```
WITH RECURSIVE recursion(liste_champ) AS (
  -- Base de l'induction
  SELECT liste_champ FROM destables1
  UNION (ALL)
  -- Induction
  SELECT liste_champ FROM recursion, destables2
  WHERE condition )

SELECT * FROM recursion;
```

WITH RECURSIVE

exemple 1

Soit la relation vol(airline,from,to,duree,prix)



Requête: quels sont tous les voyages (connexions) possibles?

WITH RECURSIVE

exemple 1

Les paires de villes connectées par des vols de ce graphe sont définies par la relation récursive suivante :

Base

vol(airline,from,to,duree,prix) ⟹ voyage(from,to)

Induction

(vol(airline,x,z,duree,prix) ∧ voyage (z,y)) ⟹ voyage(x,y)

WITH RECURSIVE

exemple 1

```
WITH RECURSIVE voyage (from,to) AS
        ( SELECT from, to FROM vol
          UNION
          SELECT R1.from, R2.to
          FROM vol AS R1 JOIN voyage AS R2 ON R1.to=R2.from )
SELECT * FROM voyage ;
-- Attention aux boucles infinies si circuit dans le graphe
```

WITH RECURSIVE

exemple 2

```
WITH RECURSIVE puiss2(nombre) AS (
        VALUES (2)
        UNION
        SELECT nombre * 2 FROM puiss2
        WHERE nombre * 2 < 100
)
SELECT nombre FROM puiss2;
-- le WHERE permet d'arrêter la récursion
```

WITH RECURSIVE

exemple 1bis

On souhaite connaitre pour chaque voyage, le nombre d'escales et le prix du voyage

```
WITH RECURSIVE voyage (from,to,escales,prix) AS
        ( SELECT from, to, 0, prix FROM vol
          UNION
          SELECT R1.from, R2.to, R2.escales+1, R2.prix+R1.prix
          FROM vol AS R1 JOIN voyage AS R2 ON R1.to=R2.from )
SELECT * FROM voyage ;
```

Data Definition Language

SQL DDL

Création d'une table

Exemple 1

```
CREATE TABLE marque (
    IdM integer PRIMARY KEY ,
    NomM character varying(100) ,
    Classe integer,
    Pays character varying(255),
    Prop integer);
```

Création d'une table

Exemple 2

```
CREATE TABLE societe(
        IdS integer PRIMARY KEY ,
        NomS character varying(100) ,
        Pays character varying(255));
```

Création d'une table

Exemple 3

```
CREATE TABLE enreg(
        NumE INT,
        IdM INT PRIMARY KEY,
        Pays VARCHAR(255),
        DateE DATE,
        IdDeposant INT);
```

INT=integer et VARCHAR =character varying

Suppression d'une table

```
DROP TABLE marque;

DROP TABLE IF EXISTS "marque" ;
CREATE TABLE marque(…);
```

Modification d'une table

```
ALTER TABLE societe
   ADD Phone CHAR(6);

ALTER  TABLE societe
   ALTER COLUMN  Phone TYPE CHAR(10);

ALTER TABLE societe
   RENAME COLUMN Phone TO Telephone;

ALTER TABLE societe
   DROP Telephone ;
```

Principaux types de données SQL

Types numériques :

- Integer ou int entier long
- smallint entier court
- bigint
- doubleprecision, real, float : réels à virgule flottante dont la représentation est binaire
- Numeric, decimal : nombre décimal à représentation exacte, échelle et précision sont facultatifs

Principaux types de données SQL

Opérations sur les types numériques :

- Opérations arithmétiques + - * /
- Comparaisons = <> < >
- et selon le SGBD des fonctions mathématiques prédéfinies: trigonométrie, random, logarithme, et voir la doc du SGBD

Principaux types de données SQL

Types alphanumériques :

- varchar(n) ou character varying(n)
  longueur variable, bornée
- character(n) ou char(n)
  longueur fixe, comblé avec des espaces

Principaux types de données SQL

Opérations sur les types alphanumériques :

- Concaténation : Prenom || « » || Nom
- Lower, upper : upper(Nom)
- et beaucoup d'autres opérations de manipulation de chaînes de caractères

Principaux types de données SQL

Correspondances de motifs: LIKE
avec deux caractères joker _ et %

'abc' LIKE 'abc' true
'abc' LIKE 'a%' true
'abc' LIKE '_b_' true
'abc' LIKE 'c' false

Principaux types de données SQL

Correspondances de motifs: LIKE

WHERE NomM LIKE 'a%'	NomM doit commencer par 'a'
WHERE NomM LIKE '%a'	NomM doit terminer par 'a'
WHERE NomM LIKE '%or%'	NomM doit contenir la sous-chaine 'or'
WHERE NomM LIKE '_r%'	NomM doit avoir 'r' comme deuxième lettre
WHERE NomM LIKE 'a_%_%'	NomM doit commencer par 'a' et être de longueur supérieure ou égale à 3
WHERE NomM LIKE 'a%o'	NomM doit commencer par 'a' et terminer par 'o'

Principaux types de données SQL

Correspondances de motifs: SIMILAR TO
Similaire à LIKE
Supporte les expressions régulières POSIX

'abc' SIMILAR TO '%(b|e)%' true

Principaux types de données SQL

Types temporels:

- Date
- Time (avec et sans fuseau horaire),
- Timestamp (avec et sans fuseau horaire),

Domaines

Un domaine est un nouveau type avec ses valeurs par défaut et ses contraintes.

CREATE DOMAIN moviedomain
AS VARCHAR(50) DEFAULT 'unknown';

Domaines

Un domaine peut être modifié et/ou supprimé

ALTER DOMAIN moviedomain
SET DEFAULT 'no such title';

DROP DOMAIN moviedomain ;

NULL

NULL et valeurs par défaut

Une valeur par défaut pour un attribut peut être spécifiée lors de la création d'une table:

ma_date DATE **NOT NULL DEFAULT** CURRENT_DATE,

identifiant SERIAL, (valeur par défaut 1 si non précisée)

Si aucune valeur par défaut n'est spécifiée, c'est la valeur **NULL** qui est attribuée aux attributs non explicitement instanciés dans un INSERT

NULL

Opérations avec NULL

NULL est une valeur spéciale disponible pour *tous* les types de données

Le résultat des opérations arithmétiques et des comparaisons sur des expressions contenant NULL est inhabituel:

- Le résultat d'une opération arithmétique dont un des termes est NULL est NULL
- Le résultat d'une opération de comparaison dont un des termes est NULL est UNKNOWN

NULL

Opérations avec NULL

NULL est une valeur par défaut mais pas une constante: elle ne peut pas être utilisée explicitement dans une expression comme X=NULL

X IS NULL ou **X IS NOT NULL** : permettent de tester si une variable contient ou non la valeur NULL

NULL

Opérations avec NULL

Supposons que x a pour valeur NULL

x-x → NULL (et non 0)

x+5 → NULL

x=3 → UNKNOWN

Expressions incorrectes : NULL = x , NULL+5

NULL

Table de valeurs de vérité étendue avec UNKNOWN

X	Y	X AND Y	X OR Y	NOT Y
TRUE	UNKNOWN	UNKNOWN	TRUE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
FALSE	UNKNOWN	FALSE	UNKNOWN	UNKNOWN

NULL

Table de valeurs de vérité étendue avec UNKNOWN

On peut mémoriser de la manière suivante :

- TRUE = 1 FALSE= 0 UNKNOWN = ½
- X AND Y = min(X,Y)
- X OR Y = max(X,Y)
- NOT X = 1-X

NULL

Retour sur la sélection avec une clause WHERE

Seuls les tuples pour lesquels une clause WHERE a la valeur de vérité TRUE sont sélectionnés

```
SELECT *  FROM movie
WHERE length <= 120 OR length > 120;
```

Les films dont length est NULL ne sont pas sélectionnés car la clause WHERE est alors évaluée à UNKNOWN

Contraintes et Triggers

Lors d'insertions, de suppressions et de mises à jour de la base il faut s'assurer que la base reste correcte:

- Contraintes: clés, références externes, restrictions sur les domaines, assertions
- Triggers: éléments actifs qui seront déclenchés lors d'un évènement particulier

Clé

Un ou plusieurs attributs d'une table:

Interdiction d'avoir deux lignes dans la table qui coïncident sur tous les attributs de la clé

Déclaration lors de la création de la table

- PRIMARY KEY
- UNIQUE NOT NULL

Clé

```
CREATE TABLE marque1 (
    id INTEGER PRIMARY KEY,
    -- ou: id INTEGER NOT NULL UNIQUE
    nomM CHAR(30),
    classe INTEGER,
    pays CHAR(2),
    prop INTEGER );
```

Clé

```
CREATE TABLE marque2 (
        id  INTEGER,
        nom CHAR(30),
        classe INTEGER,
        pays CHAR(2),
        prop INTEGER,
        PRIMARY KEY (nom, classe, pays) );
```

Clé

Une relation ne peut contenir qu'une seule déclaration PRIMARY KEY mais plusieurs déclarations UNIQUE.

Clé

Un index est crée pour chaque clé primaire (et souvent pour les contraintes UNIQUE qui ne portent que sur une seule colonne).

Contrainte référentielle: clé externe

Il est possible d'indiquer qu'un ensemble d'attributs A1, d'une relation R1, est une clé externe en référençant un ensemble d'attributs A2 d'une relation R2 (R1 et R2 peuvent être la même relation).

- A2 doit avoir été déclaré comme clé primaire de R2
- Les attributs de A1 ne peuvent prendre que des valeurs existantes des attributs de A2
  (ce qui induit un ordre de création des tables)

Contrainte référentielle: clé externe

```
CREATE TABLE enr (
        marque INTEGER REFERENCES marque1;
        … );

CREATE TABLE enr (
        FOREIGN KEY (name,k,py)
                REFERENCES marque2(nom,classe,pays);
        … );
```

Contrainte référentielle: clé externe

Une clé externe ne peut référencer que des attributs qui ont été déclarés comme PRIMARY KEY.

Contrainte référentielle: clé externe

Maintien des contraintes référentielles lors de la mise à jour d'une BDR:

- Stratégie par défaut : rejet des modifications qui entraîneraient une incohérence de la base
- Propagation en cascade de suppressions ou modifications lors de la suppression ou modification de tuples référencés
- Affectation de NULL aux attributs des clés externes lorsque les tuples référencés sont supprimés ou modifiés

Contrainte référentielle: clé externe

Maintien des contraintes référentielles :

```
CREATE TABLE societe (
      id INTEGER PRIMARY KEY,
      nom VARCHAR(30),
      ville VARCHAR(30),
      pays CHAR(2) );

CREATE TABLE vente (
      marque INTEGER REFERENCES marque1,
      vendeur INTEGER REFERENCES societe
                        ON DELETE CASCADE,
      acquéreur INTEGER REFERENCES societe
                        ON DELETE SET NULL
                        ON UPDATE CASCADE,
      date_vente DATE );
```

Contrainte référentielle: clé externe

Maintien des contraintes référentielles :

societe

id	nom	ville	pays
11	a1	v1	FR
12	a2	v2	DE

vente

marque	vendeur	acquereur	date_vente
123	11	12	2000-10-10
125	12	11	2001-10-10

- Suppression de la marque 123 dans la table marque1 impossible tant que (v1) existe dans la table vente
- Suppression du tuple (s1) dans la table societe :
  - Suppression du tuple (v1) de la table vente
  - Modification du tuple (v2) de la table vente :
    125 12 NULL 2001-10-10

Contraintes sur les valeurs d'attributs

Il est possible de limiter les valeurs que peuvent prendre certains attributs en déclarant :

- des contraintes explicites sur les attributs
- des contraintes sur les domaines

Contraintes sur les valeurs d'attributs

```
CREATE TABLE vente (
      marque INTEGER REFERENCES marque(id) NOT NULL,
      ... );
```

- marque ne peut pas prendre la valeur NULL lors d'une mise à jour de la table vente
- Il n'est pas possible d'utiliser la stratégie ON DELETE/ON CASCADE SET NULL pour cet attribut
- Il n'est pas possible d'insérer un tuple dans la table vente sans spécifier la valeur de l'attribut marque

Contraintes sur les valeurs d'attributs

```
CREATE  TABLE product (
      product_no INTEGER,
      name TEXT,
      price NUMERIC CHECK (price > 0),
      discounted_price NUMERIC );
```

- Condition de CHECK: toute expression pouvant apparaître dans WHERE
  Condition de CHECK dans postgres: condition sur les colonnes de la table

Contraintes sur les valeurs d'attributs

CREATE DOMAIN sexe CHAR(1)
CHECK (VALUE IN ('F', 'M', 'X'));

Ce type de contraintes est utile pour définir des types énumérés ou pour factoriser des contrôles de type.

Contraintes sur les valeurs d'attributs

```
CREATE TABLE societe (
        id      INTEGER PRIMARY KEY, ...
        pays    CHAR(2)
        CHECK (pays IN (SELECT * FROM lespays));
```

- La contrainte CHECK est vérifiée lors de la modification de l'attribut sur lequel elle porte. Elle peut donc être violée !
- La contrainte est vérifiée lors de la modification de pays dans societe mais pas lors de la mise à jour de la table lespays
- Il possible de différer la vérification des contraintes jusqu'à la fin d'une transaction qui porte sur plusieurs tables (utilisation du mot clé DEFERRABLE dans la déclaration de la contrainte)

Contraintes globales sur les tuples

Il est possible de limiter les valeurs que peuvent prendre certains attributs en déclarant :

- des contraintes explicites sur les attributs
- des contraintes sur les domaines

Contraintes globales sur les tuples

```
CREATE TABLE movieStar (
                name CHAR(30),
                address VARCHAR(255),
                gender CHAR(1),
                birthdate DATE,
                CHECK (gender='F' OR name NOT LIKE 'Ms.%')    );

CREATE TABLE products (
        …
        price numeric    CHECK (price > 0),
        discounted_price numeric,
        CHECK (discounted_price > 0 AND price > discounted_price) );
```

Assertions

```
CREATE ASSERTION assertion_name CHECK (condition)

CREATE ASSERTION vente_enregistrée
        CHECK (NOT EXISTS (
                SELECT * FROM vente V
                WHERE  V.marque NOT IN (
                        SELECT E.marque FROM enr E)));
```

Les assertions doivent toujours être vérifiées : toute modification de la base qui violerait une assertion est rejetée

Les assertions ne sont pas supportées sous PostgresSQL ni sous ORACLE ---- peuvent être simulées par des triggers

Modification de contraintes

1. Nomage de contraintes

```
CREATE TABLE societe (
        id INTEGER CONSTRAINT idIsKey PRIMARY KEY, ... );

CREATE DOMAIN sexe CHAR(1)
        CONSTRAINT twoValues
        CHECK (VALUE IN ('F', 'M'));

CREATE TABLE movieStar (
        name CHAR(30),
        gender CHAR(1),
        …
        CONSTRAINT rightTitle
        CHECK (gender='F' OR name NOT LIKE 'Ms.%') );
```

Modification de contraintes

2. Modification de contraintes
   Cas particulier de modification de table ou de domaine

```
ALTER TABLE societe
        DROP CONSTRAINT idIsKey ;

ALTER DOMAIN sexe
        DROP CONSTRAINT twoValues;

ALTER TABLE movieStar
        DROP CONSTRAINT rightTitle;

ALTER TABLE movieStar
        ADD CONSTRAINT NameIsKey PRIMARY KEY(name);
```

TRIGGER

Déclencheur

- Procédure activée lors d'un évènement particulier sur une table (INSERT, UPDATE, DELETE)
- N'importe quelle séquence d'instructions SQL
- Lorsqu'il est réveillé, un trigger vérifie d'abord une condition ; si elle est vraie le trigger s'exécute, si elle est fausse, rien ne se passe.

TRIGGER

Déclenchement et exécution d'un trigger

- L'action associée à un trigger peut être exécutée avant, après ou à la place de l'événement qui a déclenché le trigger
- L'action associée à un trigger peut accéder aux anciennes et aux nouvelles valeurs des tuples insérés, supprimés ou mis à jour par l'événement qui a déclenché le trigger
- Les événements de mise à jour peuvent se référer à une colonne particulière ou à un ensemble de colonnes

TRIGGER

Déclenchement et exécution d'un trigger

- Une condition peut être spécifiée avec une clause WHEN et dans ce cas l'action est exécutée uniquement si la condition est vérifiée
- Il est possible de spécifier une action qui s'applique
  - soit à chaque tuple *modifié* (une seule fois par tuple)
  - soit une seule fois à *l'ensemble* de la table

TRIGGER

```
CREATE TRIGGER name
        { BEFORE | AFTER } { event [ OR ... ] }
        ON table [ FOR [ EACH ] { ROW | STATEMENT } ]
        BEGIN … END

CREATE TRIGGER name
        { BEFORE | AFTER } { event [ OR ... ] }
        ON table [ FOR [ EACH ] { ROW | STATEMENT } ]
        EXECUTE PROCEDURE funcname ( arguments )
```

TRIGGER

```
CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    …
    salary NUMERIC (8, 2) NOT NULL,
    …
);

CREATE TABLE salary_changes (
    employee_id INT,
    changed_at DATE DEFAULT CURRENT_TIMESTAMP,
    old_salary DECIMAL(8 , 2 ),
    new_salary DECIMAL(8 , 2 ),
    PRIMARY KEY (employee_id , changed_at)
);
```

TRIGGER

```
CREATE OR REPLACE FUNCTION updatesalaries()
RETURNS trigger AS $updatesalaries$
BEGIN
  IF NEW.salary <> OLD.salary THEN
    INSERT INTO
       salary_changes(employee_id,old_salary,new_salary)
       VALUES(NEW.employee_id,OLD.salary,NEW.salary);
  END IF;
END; $updatesalaries$
LANGUAGE plpgsql;
```

TRIGGER

```
CREATE TRIGGER before_update_salary
      BEFORE UPDATE
      ON employees FOR EACH ROW
      EXECUTE PROCEDURE updatesalaries();
```

Séquences et fonctions

- Utiles pour de nombreuses opérations élémentaires
- Forte dépendance vis à vis de l'implémentation de SQL

Séquences

Compteur entier persistant à travers les sessions

- Portée:
  Accessible via toute la base mais en général une séquence est dédiée à une table
- Utilité:
  - Permet d'engendrer automatiquement un identificateur numérique
  - Génération de clé primaire

Séquences

```
CREATE [ TEMPORARY] SEQUENCE seqname
        [ INCREMENT increment ]
        [ MINVALUE minvalue ] [ MAXVALUE maxvalue ]
        [ START start ]
        [ CACHE cache ]
        [ CYCLE ]

DROP SEQUENCE seqname ;
```

Non conforme au standard SQL - Syntaxe Postgres

Séquences

nextval() , currval() et setval() permettent d'obtenir une nouvelle valeur du compteur, d'obtenir sa valeur courante, et de modifier la valeur du compteur.

```
CREATE SEQUENCE test_seq;
SELECT  currval('test_seq'); --0
SELECT nextval('test_seq'); --1
SELECT setval('test_seq', 100);
```

Séquences

nextval() , currval() et setval() peuvent figurer dans
- la clause SELECT d'une requête de type SELECT
- la clause VALUES d'une requête de type INSERT
- la clause SET d'une requête de type UPDATE

```
CREATE TABLE test (index INT, val char(1));
INSERT INTO test  (SELECT nextval('test_seq'), car FROM test1);
```

Séquences implicites

Le type SERIAL permet de créer implicitement une fonction sequence pour un attribut

```
CREATE TABLE test (index SERIAL, val CHAR(1));
INSERT INTO test(val) (SELECT car FROM test1);
SELECT * FROM test;
```

index	val
1	A
2	B
3	C
4	D

Fonctions

Nombreuses fonctions prédéfinies: mathématiques (e.g. SQRT()), manipulation de chaînes de caractères (e.g. UPPER()), aggrégats (e.g. SUM()))
-> cf. documentation!

Fonctions vs opérateurs:
- Les opérateurs sont des symboles (pas des noms)
- Les opérateurs sont en général binaires et peuvent s'écrire de manière infixe
- Opérateurs et fonctions peuvent être utilisés dans les clauses SELECT, INSERT et UPDATE ainsi que pour la définition de fonctions spécifiques (définies par l'utilisateur)

Fonctions

Des fonctions peuvent être définies dans différents langages: SQL, PL/PGSQL, PL/TCL, PL/Perl, C

```
CREATE FUNCTION name ( [ ftype [, ...] ] )
        RETURNS rtype
        AS definition
        LANGUAGE 'langname'
        [ WITH ( attribute [, ...] ) ]

CREATE FUNCTION name ( [ ftype [, ...] ] )
        RETURNS rtype
        AS obj_file , link_symbol
        LANGUAGE 'langname'
        [ WITH ( attribute [, ...] ) ]
```

Fonctions

Conversion d'une température de degrés Fahrenheits en degrés centigrades

```
CREATE FUNCTION FahrToCelc(float) RETURNS float
   AS 'SELECT ($1 - 32.0) * 5.0 / 9.0;'
   LANGUAGE SQL;

SELECT FahrToCelc(68); --20

INSERT INTO test(val, temp)
        (SELECT val, FahrToCelc(tp) FROM test2);
```

GROUPING SETS

Plusieurs regroupements dans une même requête
- Equivalent à l'union des résultats de plusieurs requêtes avec un regroupement simple GROUP BY
- Simplification d'écriture
- Meilleure performance

GROUPING SETS

```
SELECT warehouse, product, SUM (quantity) qty
FROM inventory
GROUP BY
   GROUPING SETS(
      (warehouse, product),
      (warehouse),
      (product),
      ()
   );
```

warehouse	product	qty
San Fransisco	iPhone	260
San Fransisco	Samsung	300
San Jose	iPhone	300
San Jose	Samsung	350
San Fransisco	NULL	560
San Jose	NULL	650
NULL	iPhone	560
NULL	Samsung	650
NULL	NULL	1210

ROLLUP

Génère des grouping sets en supposant une hiérarchie entre attributs

ROLLUP(c1,c2) génère les grouping sets (c1, c2), (c1) et ()

Permet d'effectuer des sous-totaux et totaux globaux

ROLLUP

```
SELECT warehouse, product, SUM(quantity)
FROM inventory
GROUP BY ROLLUP(warehouse , product);
```

warehouse	product	SUM(quantity)
San Fransisco	iPhone	260
San Fransisco	Samsung	300
San Fransisco	NULL	560
San Jose	iPhone	300
San Jose	Samsung	350
San Jose	NULL	650
NULL	NULL	1210

Total inventory in San Fransisco

Total inventory in San Jose

Total inventory in all warehouses

ROLLUP

Cas simple avec un seul attribut:

```
SELECT warehouse, SUM(quantity)
FROM inventory
GROUP BY ROLLUP(warehouse);
```

warehouse	SUM(quantity)
San Fransisco	560
San Jose	650
NULL	1210

CUBE

- Génère des sous-totaux - comme ROLLUP
- Pour toutes les combinaisons d'attributs dans le GROUP BY - à la différence de ROLLUP

```
SELECT warehouse, product, SUM(quantity)
FROM inventory
GROUP BY CUBE(warehouse , product);
```

CUBE

SELECT warehouse, product, SUM(quantity) FROM inventory

GROUP BY ROLLUP(warehouse, product);

WAREHOUSE	PRODUCT	SUM(QUANTITY)
San Francisco	Samsung	300
San Francisco	iPhone	260
San Francisco	(null)	560
San Jose	Samsung	350
San Jose	iPhone	300
San Jose	(null)	650
(null)	(null)	1210

GROUP BY CUBE(warehouse, product);

WAREHOUSE	PRODUCT	SUM(QUANTITY)
San Francisco	Samsung	300
San Francisco	iPhone	260
San Francisco	(null)	560
San Jose	Samsung	350
San Jose	iPhone	300
San Jose	(null)	650
(null)	Samsung	650
(null)	iPhone	560
(null)	(null)	1210

Fenêtrage

```
window_function_name(expression) OVER(
   PARTITION BY expr1, expr2,...
   ORDER BY expr1 [ASC | DESC], expr2,...)
   frame_clause
)
```

LEAD() / LAG()

Accès à une ligne suivante (LEAD) ou précédente (LAG)

- Par défaut la suivante ou précédente (sinon OFFSET)
- En considérant toute la table ou des regroupements regroupements de lignes (PARTITION BY)
- Les lignes peuvent être triées (ORDER BY)
- Permet ensuite d'effectuer facilement des calculs entre valeurs sur la ligne courante et une autre ligne

```
LEAD/LAG(return_value [,offset[, default ]]) OVER (
   PARTITION BY expr1, expr2,...
   ORDER BY expr1 [ASC | DESC], expr2,...)
```

LEAD() / LAG()

Pour chaque employé, date d'embauche de l'employé dans le même département embauché juste après lui.

```
SELECT  first_name, last_name, department_name, hire_date,
        LEAD(hire_date, 1, 'N/A') OVER(
          PARTITION BY department_name
          ORDER BY hire_date
        ) AS next_hire_date
FROM employees e INNER JOIN departments d
  ON d.department_id = e.department_id;
```

LEAD() / LAG()

Pour chaque employé, date d'embauche de l'employé dans le même département embauché juste après lui.

```
SELECT  first_name, last_name, department_name, hire_date,
        LEAD(hire_date, 1, 'N/A') OVER(
          PARTITION BY department_name
          ORDER BY hire_date
        ) AS next_hire_date
FROM employees e INNER JOIN departments d
  ON d.department_id = e.department_id;
```

LEAD() / LAG()

Pour chaque employé, date d'embauche de l'employé dans le même département embauché juste après lui.

first_name	last_name	department_name	hire_date	next_hire_date
Shelley	Higgins	Accounting	1994-06-07	1994-06-07
William	Gietz	Accounting	1994-06-07	N/A
Jennifer	Whalen	Administration	1987-09-17	N/A
Steven	King	Executive	1987-06-17	1989-09-21
Neena	Kochhar	Executive	1989-09-21	1993-01-13
Lex	De Haan	Executive	1993-01-13	N/A
Daniel	Faviet	Finance	1994-08-16	1994-08-17
Nancy	Greenberg	Finance	1994-08-17	1997-09-28
John	Chen	Finance	1997-09-28	1997-09-30
Ismael	Sciarra	Finance	1997-09-30	1998-03-07
Jose Manuel	Urman	Finance	1998-03-07	1999-12-07
Luis	Popp	Finance	1999-12-07	N/A

LEAD() / LAG()

Pour chaque employé et chaque année fiscale, salaire de l'année et salaire de l'année précédente.

```
SELECT   employee_id, fiscal_year, salary,
         LAG(salary) OVER(PARTITION BY employee_id
                          ORDER BY fiscal_year) previous_salary
FROM salary_history;
```

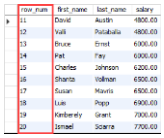
employee_id	fiscal_year	salary	previous_salary
100	2017	24000.00	NULL
100	2018	25920.00	24000.00
100	2020	26179.20	25920.00
101	2017	17000.00	NULL
101	2018	18190.00	17000.00
101	2020	19463.30	18190.00

ROW_NUMBER()

Ajout de numéros de ligne dans la table résultat

```
ROW_NUMBER() OVER(
  [PARTITION BY expr1, expr2,...]
  ORDER BY expr1 [ASC | DESC], expr2,... )

SELECT *
FROM (
  SELECT
    ROW_NUMBER() OVER(ORDER BY salary) row_num,
    first_name,
    last_name,
    salary
  FROM employees) t
WHERE row_num > 10 AND row_num <=20;
```



row_num	first_name	last_name	salary
11	David	Austin	4800.00
12	Valli	Pataballa	4800.00
13	Bruce	Ernst	6000.00
14	Pat	Fay	6000.00
15	Charles	Johnson	6200.00
16	Shanta	Vollman	6500.00
17	Susan	Mavris	6500.00
18	Luis	Popp	6900.00
19	Kimberely	Grant	7000.00
20	Ismael	Sciarra	7700.00

ROW_NUMBER()

Les employés ayant les plus hauts salaires dans leur département

```
SELECT department_name, first_name, last_name, salary
FROM (
    SELECT
      department_name,
      ROW_NUMBER() OVER(
        PARTITION BY department_name ORDER BY salary DESC) row_num,
      first_name,
      last_name,
      salary
    FROM employees e INNER JOIN departments d
        ON d.department_id = e.department_id  ) t
WHERE row_num = 1;
```

COALESCE()

COALESCE(arg1, arg2, … argn)
évalue ses arguments de gauche à droite et renvoie le premier argument non NULL

CASE

```
SELECT first_name, last_name, hire_date,
  CASE (2000 - YEAR(hire_date))
    WHEN 1 THEN '1 year'
    WHEN 3 THEN '3 years'
    WHEN 5 THEN '5 years'
    WHEN 10 THEN '10 years'
  END aniversary
FROM employees
ORDER BY first_name;
```

CASE

```
SELECT first_name, last_name,
  CASE
    WHEN salary < 3000 THEN 'Low'
    WHEN salary >= 3000 AND salary <= 5000
      THEN 'Average'
    WHEN salary > 5000 THEN 'High'
  END evaluation
FROM employees;
```