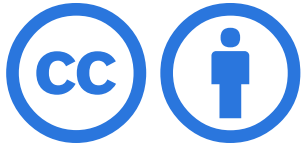
université
PARIS-SACLAY

DATA WAREHOUSE I

WEEK 4

BASED ON BENOÎT GROZ'S SLIDES



REQUÊTES SQL OLAP

TABLE DES MATIÈRES

REQUÊTES RÉCURSIVES EN SQL

INSTANCE DE BD UTILISÉE POUR ILLUSTRER LES REQUÊTES: EMPLOYEES

EMPLOYEE_ID	LASTNAME	REPORTS_TO	TITLE
1	Davolio	2	Sales Representative
2	Fuller	NULL	Vice President, Sales
3	Leverling	2	Sales Representative
4	Peacock	2	Sales Representative
5	Buchanan	2	Sales Manager
6	Suyama	5	Sales Representative
7	King	5	Sales Representative
8	Callahan	2	Inside Sales Coordinator
9	Dodsworth	5	Sales Representative

- On souhaite étudier les hiérarchies: qui supervise qui, directement ou non.
- La profondeur de la hiérarchie n'est pas forcément connue à l'avance (et en fait est peu pratique à exploiter, de toute façon).

CLAUSE WITH/CTE (COMMON TABLE EXPRESSIONS)

Syntaxe simplifiée:

```
WITH nom_sous_requête AS (
    <expression_de_sous_requête>
)
                <requête pouvant faire appel à nom_sous_requête>;
```

Disponible sous Oracle, SQL Server, IBM, MariaDB, PostgreSQL.

Avantages : Réutilisation d'expressions, évite le surcoût de réévaluation et les éventuelles incohérences, support pour les requêtes récursives.

CLAUSE WITH

```
WITH subquery_name_1 AS ( expressions_1 ) search_clause_1 cycle_clause_1 ,
    . . .
    subquery_name_n AS ( expressions_n ) search_clause_n cycle_clause_n ,
    query_expression
;
```

Exemple de requête récursive avec des alias de colonnes:

```
WITH reports_to_2 (eid, mgr_id, reportLevel) AS (
    SELECT employee_id, reports_to, 0 reportLevel
    FROM employees
    WHERE employee_id = 2

UNION ALL

    SELECT e.employee_id, e.reports_to, reportLevel + 1
    FROM employees e, reports_to_2 r
    WHERE r.eid = e.reports_to
)
SELECT eid, mgr_id, reportLevel
FROM reports_to_2
ORDER BY reportLevel, eid;
```

RÉSULTAT DE LA REQUÊTE:

Employees

EMPLOYEE_ID	LASTNAME	REPORTS_TO	TITLE
1	Davolio	2	Sales Representative
2	Fuller	NULL	Vice President, Sales
3	Leverling	2	Sales Representative
4	Peacock	2	Sales Representative
5	Buchanan	2	Sales Manager
6	Suyama	5	Sales Representative
7	King	5	Sales Representative
8	Callahan	2	Inside Sales Coordinator
9	Dodsworth	5	Sales Representative

Résultat de la requête

EID	MGR_ID	REPORTLEVEL
2		0
1	2	1
3	2	1
4	2	1
5	2	1
8	2	1
6	5	2
7	5	2
9	5	2

CLAUSE WITH: RÈGLES

```
WITH cte1(col1, col2) AS (...),
    cte2 (col3) AS (...),
    cte3 AS (Select col1, col3 FROM cte1, cte2 WHERE col1 > col3),
    cte4 AS (...)
SELECT ... ;
```

On peut avoir une liste de CTE: chaque CTE peut dépendre des précédents

La partie récursive (après le UNION ALL) ne peut contenir ni agrégation ni négation.

Sous postgresql: un CTE récursif doit être déclaré :

```
WITH RECURSIVE mon_cte1 ...
```

SQL (OLAP) GROUP BY EXTENSIONS

INTERROGER L'ENTREPÔT DANS LE MODÈLE RELATIONNEL

L'analyste formule une requête OLAP sur un client OLAP.

Le serveur doit retourner la réponse efficacement.

↪ dans le cas ROLAP: nécessaire de simuler les opérations OLAP.

⇒ langage de requête sur les données relationnelles: SQL.

INTERROGER L'ENTREPÔT DANS LE MODÈLE RELATIONNEL

L'analyste formule une requête OLAP sur un client OLAP.

Le serveur doit retourner la réponse efficacement.

↪ dans le cas ROLAP: nécessaire de simuler les opérations OLAP.

⇒ langage de requête sur les données relationnelles: SQL.

Pas de langage de requête spécifique pour les opérations OLAP mais SQL-99 étend SQL-92 avec (entre autres) des fonctions d'agrégation OLAP.

OLAP SOUS SQL-92

Les opérations OLAP usuelles:

- `DRILL-DOWN` : jointures entre faits et dimensions, `GROUP BY`
- `ROLL-UP` : jointures et agrégations cumulatives, `GROUP BY`
- `SLICE, DICE` : clause `WHERE`
- `DRILL ACROSS` : `JOIN`, `GROUP BY`

EXEMPLE DE TABLEAU CROISÉ DYNAMIQUE

tableau croisé dynamique (en anglais pivot table)

Quarters	Beverages	Produce	Condiments	Total
Q1	21	10	18	49
Q2	27	14	11	52
Q3	26	12	35	73
Q4	14	20	47	81
Total	**88**	**56**	**111**	**255**

Agrégations des ventes

Quarter Key	Category Key	Amount
Q1	B	21
Q1	P	10
Q1	C	18
Q1	NULL	49
Q2	B	27
Q2	P	14
Q2	C	11
Q2	NULL	52
Q3	B	26
Q3	P	12
Q3	C	35
Q3	NULL	73
Q4	B	14
Q4	P	20
Q4	C	47
Q4	NULL	81
NULL	B	88
NULL	P	56
NULL	C	111
NULL	NULL	255

Speaker notes

COMPRENDRE LES AGRÉGATIONS EN CUBE ET LES TABLEAUX CROISÉS

Dans cette diapositive, nous explorons l'utilisation des **agrégations en cube** et comment un **tableau croisé dynamique** aide à résumer les données de manière plus lisible.

1. TABLEAU CROISÉ DYNAMIQUE (CÔTÉ GAUCHE)

Le tableau croisé dynamique sur la gauche est un **résumé** des données de ventes par **catégories de produits** (Boissons, Produits frais, Condiments) et par **trimestres** (Q1, Q2, Q3, Q4). Il permet de voir rapidement :

- Le **total des ventes** pour chaque catégorie dans chaque trimestre.
- Les **totaux par ligne** (Total par trimestre) : le total des ventes pour toutes les catégories dans un trimestre spécifique.
- Les **totaux par colonne** (Total par produit) : le total des ventes pour chaque catégorie de produit sur tous les trimestres.
- Le **total général** dans le coin inférieur droit, représentant le total des ventes pour toutes les catégories et tous les trimestres (255).

2. AGRÉGATIONS DES VENTES (CÔTÉ DROIT)

Le tableau de droite montre les **données agrégées détaillées** pour chaque combinaison de :

- **Quarter Key** (le trimestre où les ventes ont eu lieu),
- **Category Key** (la catégorie de produit),
- **Amount** (le total des ventes pour cette combinaison).

De plus, ce tableau contient des lignes où soit **Quarter Key**, soit **Category Key** est `NULL`, représentant les **totaux** :

- **NULL** dans la colonne Category Key signifie qu'il s'agit du **total des ventes pour ce trimestre** dans toutes les catégories.
- **NULL** dans la colonne Quarter Key signifie qu'il s'agit du **total des ventes pour cette catégorie** sur tous les trimestres.
- Lorsque les deux sont `NULL`, cela donne le **total général** de toutes les ventes.

3. LE CONCEPT D'AGRÉGATIONS EN "CUBE"

Le tableau de droite est le résultat d'une **agrégation en cube**, qui calcule **toutes les combinaisons possibles** de trimestre et de catégorie. L'opération `CUBE` :

- Fournit les totaux des ventes pour chaque combinaison de trimestre et de catégorie.
- Inclut des **totaux partiels** par trimestre et par catégorie.
- Inclut le **total général** pour toutes les dimensions.

Cela équivaut à exécuter la requête SQL suivante :

```sql
SELECT quarter_key, category_key, SUM(amount) AS total_sales
FROM sales_data
GROUP BY CUBE(quarter_key, category_key);
```

4. POURQUOI UTILISER UN TABLEAU CROISÉ DYNAMIQUE ?

Bien que vous puissiez calculer les totaux en utilisant des requêtes SQL individuelles ou un `GROUP BY CUBE`, le **tableau croisé dynamique** est un moyen pratique de résumer visuellement les données :

- Il organise les résultats dans un format de matrice lisible.
- Il montre plusieurs **agrégations simultanément**, ce qui facilite l'interprétation des données.

Alors que le tableau de droite fournit toutes les agrégations détaillées, le tableau croisé dynamique rend ces résultats plus faciles à **comprendre** et à **analyser**.

L'**agrégation en cube** est un outil puissant pour résumer les données sur plusieurs dimensions, et un **tableau croisé dynamique** est un moyen utile de présenter ces résultats. Le tableau de droite montre toutes les agrégations brutes, tandis que le tableau croisé dynamique du côté gauche présente ces agrégations de manière plus concise et facile à comprendre.

AUTRES REQUÊTES DIFFICILES À EXPRIMER

- comparer les valeurs agrégées d'une année sur l'autre
- moyennes glissantes
- agrégations cumulatives, etc

EXTENSIONS APPORTÉES PAR SQL 99

Objectif: rendre les requêtes OLAP plus simples et rapides.

Instructions d'agrégations supplémentaires apportées par SQL 99:

- `CUBE`: toutes les combinaisons
- `ROLLUP`: combinaisons le long d'une hiérarchie
- `GROUPING SETS`: la liste des combinaisons souhaitées est précisée explicitement

Disponibles sous Oracle, SQL Server, IBM DB2, PostgresQL, MariaDB, mais pas MYSQL.

Solution permettant les agrégations multiples OLAP. Pour ce qui est des autres pbs: il faut d'autres fonctions d'agrégations...

CUBE & ROLLUP

CUBE:

```
SELECT city, month, SUM(quantity) AS Quant
FROM Facts F, Time T, Product P, Location
WHERE P.PID = F.PID
    AND T.TID = F.TID
    AND L.LID = F.LID
    AND P.category = 'DVD'
    AND T.quarter = 'Q1 2010'
GROUP BY CUBE (month, city);
```

GROUP BY CUBE: toutes combinaisons

Agrégats pour toute paire `(month, NULL)`, `(NULL, city)`, `(month, city)`, et `(NULL, NULL)`.

CITY	MONTH	QUANTITY
		226
Lyon		32
Paris		85
Berlin		67
Stuttgart		42
Lyon	fev10	12
Paris	fev10	25
Berlin	fev10	25
Stuttgart	fev10	15
	jan10	70
Lyon	jan10	10
Paris	jan10	30
Berlin	jan10	20
Stuttgart	jan10	10
	mar10	79
Lyon	mar10	10
Paris	mar10	30
Berlin	mar10	22
Stuttgart	mar10	17

ROLLUP:

```
SELECT quarter, month, SUM(sales) AS s
FROM Facts F, Time T, Product P
WHERE P.PID = F.PID AND T.TID = F.TID
GROUP BY ROLLUP(quarter, month);
```

Résultat:

QUARTER	MONTH	S
Q1 2010	fev10	422
Q1 2010	jan10	675
Q1 2010	mar10	400
Q1 2010		1497
Q4 2010	dec10	253
Q4 2010		253
		1750

GROUPING SETS

Syntax:

```
... GROUP BY GROUPING SETS (
    (a_11, a_12, ..., a_1n1),
    (b_21, b_22, ..., b_2n2),
    ...
    (b_k1, b_k2, ..., b_knk)
);
```

Spécifie chaque groupement désiré par une liste `(b_i1, b_i2, ..., b_ini)`.

Équivalent à une `UNION ALL` des requêtes `GROUP BY` en complétant avec `NULL`.

EXEMPLE:

```
SELECT quarter, type, city, SUM(sales) AS s
FROM Facts F, Time T, Product P, Location L
WHERE P.PID = F.PID
    AND T.TID = F.TID
    AND L.LID = F.LID
    AND T.year = '2010'
GROUP BY GROUPING SETS (
    (quarter, type),
    (quarter, city)
);
```

Résultats:

QUARTER	TYPE	CITY	S
Q1 2010	Media	NULL	1200
Q1 2010	Livres	NULL	297
Q4 2010	Media	NULL	253
Q1 2010		Berlin	385
Q1 2010		Paris	385
Q1 2010		Stuttgart	372
Q1 2010		Lyon	355
Q4 2010		Lyon	103
Q4 2010		Berlin	150

GROUPING SETS VS ROLLUP/CUBE

GROUPING SETS permet d'exprimer ROLLUP, CUBE, GROUP BY (a, b).

```
SELECT T.year_id, T.month_id, T.day_id, SUM(amount)
FROM sales S, Time T
WHERE T.day_id = S.day_id
GROUP BY ROLLUP(T.year_id, T.month_id, T.day_id);
```

Exprimer la requête ci-dessus avec `GROUPING SETS`.

`GROUP BY year_id, month_id` avec `GROUPING SETS`?

GROUPING SETS AVEC ROLLUP/CUBE

SYNTAXE COMPLÈTE

```
... GROUP BY GROUPING SETS (
    <liste_d'attributs_1>,
    <liste_d'attributs_2>,
    ...
    <liste_d'attributs_i>,
    ROLLUP <liste_d'attributs_1>,
    ...
    ROLLUP <liste_d'attributs_j>,
    CUBE <liste_d'attributs_1>,
    ...
    CUBE <liste_d'attributs_k>
);
```

GROUPEMENTS CONCATÉNÉS

Produit cartésien de groupements.

```
GROUP BY X, Y = GROUP BY GROUPING SETS (
    {{(a1, ... , ai, b1, ... , bj) | <a1, ...="" ai=""> ∈ X , <b1, ...=""
);</b1,></a1,>
```

X, Y: listes d'attributs, `GROUPING SETS` ou `ROLLUP`.

La sémantique de `GROUP BY` est le produit cartésien de ses groupes.

COLONNES COMPOSITES (B, C)

Ensemble de colonnes traitées “en bloc” comme une unique colonne dans les groupements.

- `Colonne composite` : `ROLLUP(a,(b,c))` est équivalent à `GROUPING SETS((a,b,c),(a),())`

EXEMPLES DE GROUPEMENTS MULTIPLES ET COLONNES COMPOSITES

Identifiez les groupements:

- GROUP BY a, GROUPING SETS ((b), (c,d))
- GROUP BY a, ROLLUP (a, b)
- GROUP BY a, GROUPING SETS ((b), (c), ())
- GROUP BY GROUPING SETS((a,b), (b,c)), GROUPING SETS((d,e),(d),())
- GROUP BY CUBE((a,b),(b,c))

FONCTION GROUPING

IDENTIFICATION DES NULLS À L'AIDE DE GROUPING

SQL 99 apporte une instruction `GROUPING(a)` qui identifie si `a` fait parti d'une agrégation.

- `GROUPING(a)`
  - retourne 1 quand `a` ne fait pas partie du groupement pour ce tuple (i.e., `a` est agrégé).
  - retourne 0 sinon (`NULL` dans table d'origine, ou valeur non `NULL`).

```
SELECT quarter, DECODE(GROUPING(city),1,'multicity',city) city_desc,
    SUM(sales) sales, GROUPING(city), GROUPING(quarter)
FROM Facts F, Time T, Location L
WHERE T.TID = F.TID AND L.LID = F.LID AND L.country='France'
GROUP BY GROUPING SETS (
    (quarter),
    (quarter, city)
);
```

FONCTIONS ANALYTIQUES

Speaker notes

Les fonctions analytiques permettent d'appliquer des agrégats sur une "fenêtre" de tuples.

PARTITION

Partitionne les tuples en entrée et, pour chaque tuple, calcule la fonction d'agrégation sur la fenêtre associée.

INPUT DATA

Group	Value
a	v9
a	v8
a	v7
b	v6
b	v5
b	v4
b	v3
b	v2
c	v1
c	v0

PARTITIONS AND SUM OVER PARTITION

PARTITION 1 (GROUP a):

Group	Value	SUM (Partition)
a	v9	v7 + v8 + v9
a	v8	v7 + v8 + v9
a	v7	v7 + v8 + v9

PARTITION 2 (GROUP b):

Group	Value	SUM (Partition)
b	v6	v2 + ... + v6
b	v5	v2 + ... + v6
b	v4	v2 + ... + v6
b	v3	v2 + ... + v6
b	v2	v2 + ... + v6

PARTITION 3 (GROUP c):

Group	Value	SUM (Partition)
c	v1	v0 + v1
c	v0	v0 + v1

Disponible sous Oracle, SQL Server, DB2, PostgreSQL, MariaDB (dev), mais pas MySQL.

PARTITION

```
window_function OVER ([PARTITION BY attribute_sequence])
```

Partitionne les tuples, puis agrège chaque partition de façon indépendante.

```
SELECT quarter, date, city, sales,
    SUM(sales) OVER(PARTITION BY quarter) AS SALES_QT
FROM Facts F, Time T, Product P, Location L
WHERE P.PID = F.PID
    AND T.TID = F.TID
    AND L.LID = F.LID;
```

RÉSULTATS

QUARTER	DATE	CITY	SALES	SALES_QT
Q1 2010	01Jan10	Berlin	120	1497
Q1 2010	04Feb10	Berlin	120	1497
Q1 2010	20Mar10	Berlin	145	1497
Q1 2010	01Jan10	Lyon	355	1497
Q1 2010	10Mar10	Paris	385	1497
Q1 2010	10Jan10	Stuttgart	372	1497
Q4 2010	18Nov10	Berlin	50	253

PARTITION

```
window_function OVER ([PARTITION BY attribute_sequence])
```

partitions en présence d'une clause GROUP BY:

```
SELECT quarter, date, city,
    SUM(SUM(sales)) OVER(PARTITION BY quarter) AS SALES_QT,
    SUM(sales) AS salesC
FROM Facts F, Time T, Product P, Location L
WHERE P.PID = F.PID
    AND T.TID = F.TID
    AND L.LID = F.LID
GROUP BY quarter, city;
```

RÉSULTATS

```
QUARTER  | CITY      | SALES_QT | SALESC
-------- | --------- | -------- | ------
Q1 2010  | Berlin    | 1497     | 385
Q1 2010  | Lyon      | 1497     | 355
Q1 2010  | Paris     | 1497     | 385
```

ORDONNER LES TUPLES D'UNE FENÊTRE

```
window_function OVER ([PARTITION BY ...] [ORDER BY ...])
```

La fonction trie les tuples à l'intérieur d'une partition et permet d'appliquer des agrégations restreintes aux tuples de la partition qui sont inférieurs ou égaux.

CLASSEMENT DES VILLES SELON LES VENTES, PAR TRIMESTRE

```
SELECT quarter, city,
    RANK() OVER(PARTITION BY quarter ORDER BY SUM(sales)) AS RG,
    SUM(sales) AS s
FROM Facts F, Time T, Product P, Location L
WHERE P.PID = F.PID AND T.TID = F.TID AND L.LID = F.LID
GROUP BY quarter, city;
```

QUARTER	CITY	RG	S
Q1 2010	Lyon	1	355
Q1 2010	Stuttgart	2	372
Q1 2010	Berlin	3	385

Speaker notes

Dans cet exemple, on utilise la fonction `RANK` pour classer les villes en fonction de leurs ventes par trimestre. La fonction `OVER()` permet de partitionner les ventes par trimestre (colonne `quarter`) et d'ordonner les ventes dans chaque partition par la somme des ventes (colonne `s`). Le classement est ensuite affiché avec la colonne `RG`.

RESTREINDRE L'AGRÉGATION À UNE PLAGE DE TUPLES : ROWS

- La clause `ROWS` limite l'ensemble de tuples sur lesquels porte l'agrégation.
- Avec `ROWS`, la plage de tuples est spécifiée par un nombre de lignes fixes précédant ou suivant la ligne courante.

  Ventes moyennes sur les 3 derniers mois pour chaque ville et mois.

```
SELECT T.TID AS T, month, city,
    AVG(SUM(sales)) OVER (
    PARTITION BY city
    ORDER BY T.TID
    ROWS 2 PRECEDING) sales3,
    SUM(sales) AS s
FROM Facts F, Time T, Product P, Location L
WHERE P.PID = F.PID
    AND T.TID = F.TID
    AND L.LID = F.LID
GROUP BY month, T.TID, city
ORDER BY city, T.TID;
```

RESTREINDRE L'AGRÉGATION À UNE PLAGE DE TUPLES : RANGE

- La clause `RANGE` spécifie la plage par une plage de valeurs par rapport à la ligne courante. → `ORDER BY` doit être exprimé en termes numériques, date, etc.
- La plage est définie par une association “logique” plutôt que “physique”.

Compter les villes dont les ventes sont entre 90% et 100% de la ville courante, par trimestre.

```
SELECT quarter, city,
    COUNT(*) OVER (
    PARTITION BY quarter
    ORDER BY SUM(sales)
    RANGE 0.1*SUM(sales) PRECEDING) NB,
    SUM(sales) AS s
FROM Facts F, Time T, Product P, Location L
WHERE P.PID = F.PID
    AND T.TID = F.TID
```

QUARTER	CITY	NB	S
Q1 2010	Lyon	1	355
Q1 2010	Stuttgart	2	372
Q1 2010	Berlin	4	385
Q1 2010	Paris	4	385
Q4 2010	Lyon	1	103
Q4 2010	Berlin	1	150

PORTÉE DES AGRÉGATS

- `OVER()` : tous les tuples
- `OVER(PARTITION BY)` : les tuples de la partition
- `OVER(ORDER BY)` : les tuples inférieurs ou égaux au tuple courant, équivalent à `RANGE UNBOUNDED PRECEDING`

- `ROWS UNBOUNDED PRECEDING` : tuples inférieurs dans la partition
- `ROWS k PRECEDING` : `k` tuples inférieurs dans la partition
- `UNBOUNDED FOLLOWING` :
  - `k FOLLOWING` : `k` tuples suivants dans la partition
  - `CURRENT ROW` : ligne courante
  - `BETWEEN` :
    - `ROWS BETWEEN 2 PRECEDING AND CURRENT ROW` = `ROWS 2 PRECEDING`
- `RANGE` : similaire à `ROWS` mais définit la plage comme un intervalle par rapport à la valeur

⚠ `ROWS`, `RANGE` requièrent qu'un ordre soit spécifié.

FONCTIONS D'AGRÉGATION ET CLASSEMENT

- Fonctions d'agrégation (sémantique ensembliste → relationnelle)
  - `SUM`, `COUNT`, `MIN`, `MAX`, `AVG`, `STDDEV_POP`, `VAR_SAMP`
- Fonctions de classement (sur un ordre)
  - `RANK` : Classement avec ex-aequo. Les positions suivantes sont sautées (ex : 1, 2, 2, 4).
  - `DENSE_RANK` : Classement sans saut de positions (ex : 1, 2, 2, 3).
  - `ROW_NUMBER` : Numérotation sans tenir compte des ex-aequo.
  - `NTILE(n)` : Division des tuples en n parties égales. Retourne le numéro de tuile du tuple.
  - `LEAD(expr, offset, default)` : Renvoie la valeur de l'expression spécifiée dans la ligne suivante dans la fenêtre.
  - `LAG(expr, offset, default)` : Renvoie la valeur de l'expression spécifiée dans la ligne précédente dans la fenêtre.
  - `FIRST_VALUE(expr)` : Renvoie la première valeur de l'expression spécifiée dans la fenêtre.
  - `LAST_VALUE(expr)` : Renvoie la dernière valeur de l'expression spécifiée dans la fenêtre.

Speaker notes

Ces fonctions permettent d'appliquer des ordres et des classements sur des ensembles de données, et les fonctions d'agrégation servent à calculer des statistiques simples sur des ensembles de valeurs.

FONCTIONS DE CLASSEMENT

- `ROW_NUMBER` : ex-aequos départagés arbitrairement
- `RANK vs DENSE_RANK` : ne “saute” pas de valeurs après des ex-aequos

```
SELECT month, city, SUM(sales) AS s,
    RANK() OVER (ORDER BY SUM(sales) DESC) AS RK,
    DENSE_RANK() OVER (ORDER BY SUM(sales) DESC) AS DENSE_RK,
    ROW_NUMBER() OVER (ORDER BY SUM(sales) DESC) AS ROW_NUM
FROM Facts F, Time T, Location L
WHERE T.TID = F.TID AND L.LID = F.LID
GROUP BY month, city;
```

MONTH	CITY	S	RK	DENSE_RK	ROW_NUM
jan10	Paris	185	1	1	1
jan10	Berlin	172	2	2	2
jan10	Stuttgart	150	3	3	3
jan10	Lyon	150	3	3	4
dec10	Berlin	150	4	4	5

FONCTIONS DE CLASSEMENT

```
CUME_DIST() OVER ( [query_partition_clause] order_by_clause )
```

- CUME_DIST() pour le tuple r dans un ensemble de tuples S : (nombre de lignes plus petites que r dans S) / N, où N = |S|

```
SELECT year, month, city,
    SUM(sales) AS sales,
    CUME_DIST() OVER (
    PARTITION BY year
    ORDER BY SUM(sales)) AS Cum_Dist
FROM Facts F, Time T, Location L
WHERE T.TID = F.TID
    AND L.LID = F.LID
GROUP BY year, month, city;
```

MONTH	CITY	SALES	CUM_DIST
fev10	Stuttgart	100	0.50
dec10	Lyon	103	0.50
fev10	Paris	100	0.50
dec10	Berlin	107	0.64

FONCTIONS DE CLASSEMENT

`NTILE(k)` : distribue les lignes en `k` groupes de même taille et retourne le n° de groupe (les derniers groupes peuvent avoir 1 de moins)

```
SELECT month, city, SUM(sales) AS s,
    NTILE(3) OVER (
        ORDER BY SUM(sales) DESC) AS
FROM Facts F, Time T, Product P, Loc
WHERE P.PID = F.PID
    AND T.TID = F.TID
    AND L.LID = F.LID
GROUP BY month, city;
```

MONTH	CITY	S	TILE3
jan10	Paris	185	1
jan10	Berlin	172	1
jan10	Stuttgart	172	1
jan10	Lyon	150	2
dec10	Berlin	150	2
dec10	Lyon	103	2
mar10	Paris	100	3
mar10	Lyon	100	3
mar10	Stuttgart	100	3
fev10	Berlin	100	3
fev10	Paris	100	3

FONCTIONS ANALYTIQUES FIRST_VALUE / LAST_VALUE

`FIRST_VALUE, LAST_VALUE` : accède à la première/dernière valeur dans la portée d'agrégation.

```
SELECT month, city, SUM(sales) AS s,
    FIRST_VALUE(SUM(sales)) OVER (
    PARTITION BY month
    ORDER BY SUM(sales) DESC
    ) AS sales_max
FROM Facts F, Time T, Location L
WHERE T.TID = F.TID
    AND L.LID = F.LID
GROUP BY month, city;
```

MONTH	CITY	S	SALES_MAX
dec10	Berlin	150	150
dec10	Lyon	103	150
fev10	Berlin	100	125
fev10	Lyon	125	125
fev10	Paris	100	125
fev10	Stuttgart	100	125
jan10	Paris	185	185
jan10	Berlin	172	185
jan10	Stuttgart	172	185
jan10	Lyon	150	185
mar10	Paris	100	100
mar10	Lyon	100	100
mar10	Berlin	100	100
mar10	Stuttgart	100	100

Quel résultat obtient-on en remplaçant FIRST_VALUE par LAST_VALUE ?

NAMED WINDOW FRAMES IN POSTGRESQL

PostgreSQL permet de définir des cadres de fenêtres nommés qui peuvent être réutilisés dans les requêtes.

```
SELECT sum(amount) OVER w, avg(amount) OVER w, RANK() OVER w2
FROM empsalary
WINDOW w AS (PARTITION BY depname ORDER BY salary DESC),
        w2 AS (ORDER BY salary DESC);
```