

4. Data Warehouse data analysis

User interface

User may query the **DW** by using:

- **controlled query environment**
- **pre-defined query** and **report generation tools**
- **data mining tools**

OLAP Analysis

Roll up

Data detail reduction by decreasing detail in a dimension, by climbing up a hierarchy

e.g. : *group by store, month* → *group by city, month*

or dropping a whole dimension(not physically)

Drill down

Data detail increase by increasing detail in a dimension by walking down a hierarchy

e.g. : *group by city, month* → *group by store, month*

or adding a whole dimension(not physically).

It's the opposite of the roll up operation.

Often used on a subset of data produced by the initial query in order to focus more on certain aspects.

Slice and dice

Selection of a data subset by means of a selection predicate:

- **Slice:** equality predicate selecting a *slice*(una fetta praticamente)
- **Dice:** predicate expression selecting a *dice*(un ipercubo più piccolino).

Pivot

Rearrangement of the multidimensional structure without changing the detail level. Useful to make the information more readable.

Extended SQL Language

New **OLAP** functions characterized by a new **computation window** which inside aggregation functions are performed.

Also new aggregate functions are added.

Computation window

A moving **window** clause characterized by

- **partitioning**: rows are grouped without collapsing them (unlike group by)
- **row ordering** separately in each partition (like order by)
- **aggregation window**: for each row in the partition, it defines the row group on which the aggregate function is computed

Example

Show, for each city and month


- sale amount
- average on the current month and the two previous months, separately for each city

City	Month	Amount	MovingAvg	
Milano	7	110	110	Partition 1
Milano	8	10	60	
Milano	9	90	70	
Milano	10	80	60	
Milano	11	40	60	
Milano	12	140	90	
Torino	7	70	70	Partition 2
Torino	8	30	50	
Torino	9	80	60	
Torino	10	100	70	
Torino	11	50	60	
Torino	12	150	100	

```

SELECT City, Month, Amount,
       AVG(Amount) OVER (PARTITION BY City
                        ORDER BY Month
                        ROWS 2 PRECEDING)
       AS MovingAvg
FROM Sales

```



Cyan part: we compute the average **over** a moving window defined in the brackets. This window is defined as follows: partitioned by city, ordered by month and contains the current row + 2 preceding rows

Computation Window

It can be defined with 2 different approaches:

- **physical level:** builds the group by *counting rows*
- **logical level:** builds the group by defining an interval on the sort key

Physical interval definition

- Can be defined between a lower bound and the current row:
ROWS 2 PRECEDING → aggregation window of size 3: current row + 2 preceding rows, it may be incomplete and has a fixed size
- Can be defined between lower and upper bounds:
ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING → aggregation window of size 3: current row + preceding row + following row.
ROWS BETWEEN 3 PRECEDING AND 1 FOLLOWING → same as above but with 2 more rows.
- Between the beginning(or the end) of a partition and the current row
ROWS UNBOUNDED PRECEDING(o FOLLOWING) → current row + all preceding/following rows

This approach is mostly used when there are no gaps in the data(every month is present in the dataset).

Logical interval definition

For this kind of approach we use the **RANGE** keyword

It's appropriate for sparse data.

Unlike the physical definition, here we can only specify a single sort key(in physical the rows are ordered according to the **ORDER BY** clause)

Group By

Aggregation windows can be used with a **GROUP BY** clause.

The temporary table generated by the GROUP BY clause it's the element where the aggregation window performs the computation

Ranking functions

New aggregating functions that compute rank of a value inside a partition

- **rank()**: computes the rank by leaving an empty slot after a tie
example: after 2 first, the next rank is third
- **denserank()**: computes the rank by leaving an empty slot after a tie
example: after 2 first, the next rank is second

Order By

The **ORDER BY** clause inside a partition doesn't affect how the end result is visualized. It's purpose is to order the data inside a partition to perform a specific computation.

Group By Extension

- **rollup:**
- **cube:**
- **grouping sets**