VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

OLAP Queries and SQL:1999

Lecturer : Dr. Pavle Mogin

SWEN 432
Advanced Database Design and Implementation

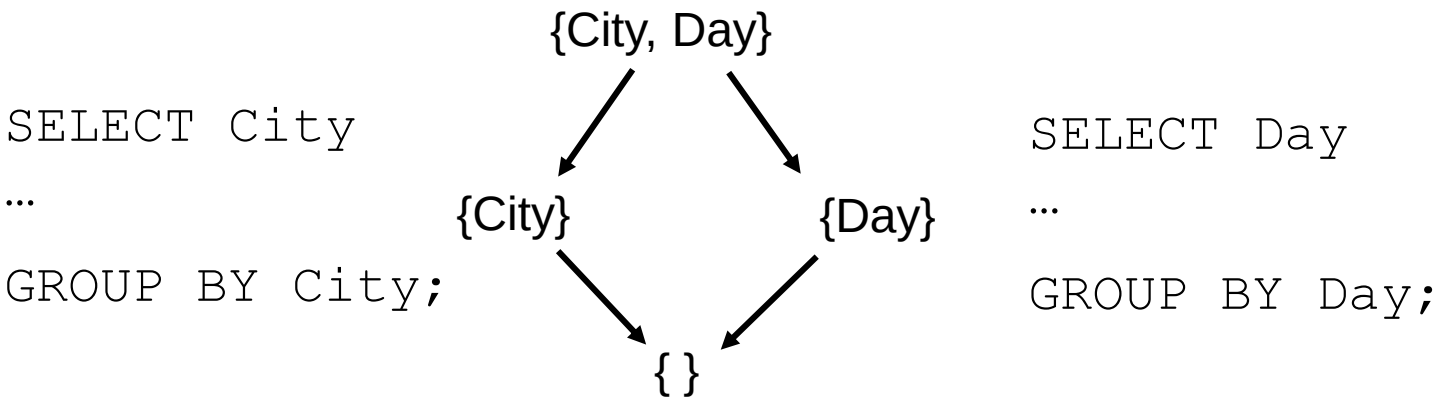
Plan for OLAP Queries and SQL:1999

- Inefficiency of implementing pivoting using SQL/92
- New SQL commands:
  - CUBE
  - ROLLUP
  - WINDOW
  - New aggregate functions

  - Reading
    - *Chaudhuri and Dayal* : “An Overview of Datawarehousing and OLAP Technologies”

Summary of the Cross Tab Example

```
SELECT City, Day, SUM(Amnt)… GROUP BY City, Day
```




```
SELECT City
…
GROUP BY City;
```

```
SELECT Day
…
GROUP BY Day;
```

```
SELECT SUM(Amnt) FROM Sales
```

Structures like that one are often called ***cube lattice***

Inefficiency of Multiple SELECTs

- To express pivoting over *n* dimensions, one has to declare 2^n `SELECT` SQL/92 statements
- Much worse, query optimizer will treat each `SELECT` as a separate block and evaluate it from the scratch, performing many joins and aggregations in vane
- Namely, very often, each node down the lattice may be evaluated from its ancestor

Using Ancestors

- Let us denote by V1 the (stored) result of the query

```
SELECT City, Day, SUM(Amnt) AS Sale
FROM Sales s, Location l, Time t
WHERE s.LocId = l.LocId AND s.TimId = t.TimId
GROUP BY City, Day;
```

- Then

```
SELECT City, '', SUM(Amnt) FROM Sales s, Location l
WHERE s.ShopId = l.ShopId GROUP BY City;
```

may be rewritten as

```
SELECT City, '', SUM(Sale) FROM V1 GROUP BY City;
```

and thus save computing a costly join (and some aggregating, as well)

SQL:1999 Extensions to GROUP BY

- Pivoting can be expressed by a series of SQL statements of the roll-up type:

```
SELECT <attr_list> SUM(Measure)
FROM <fact_table> NATURAL JOIN <dimension_table>
WHERE <condition>
GROUP BY <grouping_list>
```

- Recognizing the high level of similarities between such frequently used statements and the need to compute them efficiently has lad to extending the GROUP BY clause by CUBE and ROLLUP clauses

SQL:1999 CUBE Extension

- The GROUP BY clause with CUBE keyword is equivalent to a collection of GROUP BY statements, with one GROUP BY statement for each subset of *n* dimensions
- The statement

```
SELECT City, Month, SUM(Amnt) AS Sale
FROM Sales NATURAL JOIN Location
NATURAL JOIN Time t
GROUP BY CUBE (City, Month);
```

will produce the same result as four other SQL statements with all subsets of {City, Month} as grouping attributes

CUBE

- Given a fact table

  `Sales(ProdId, LocId, TimeId, Amnt )`

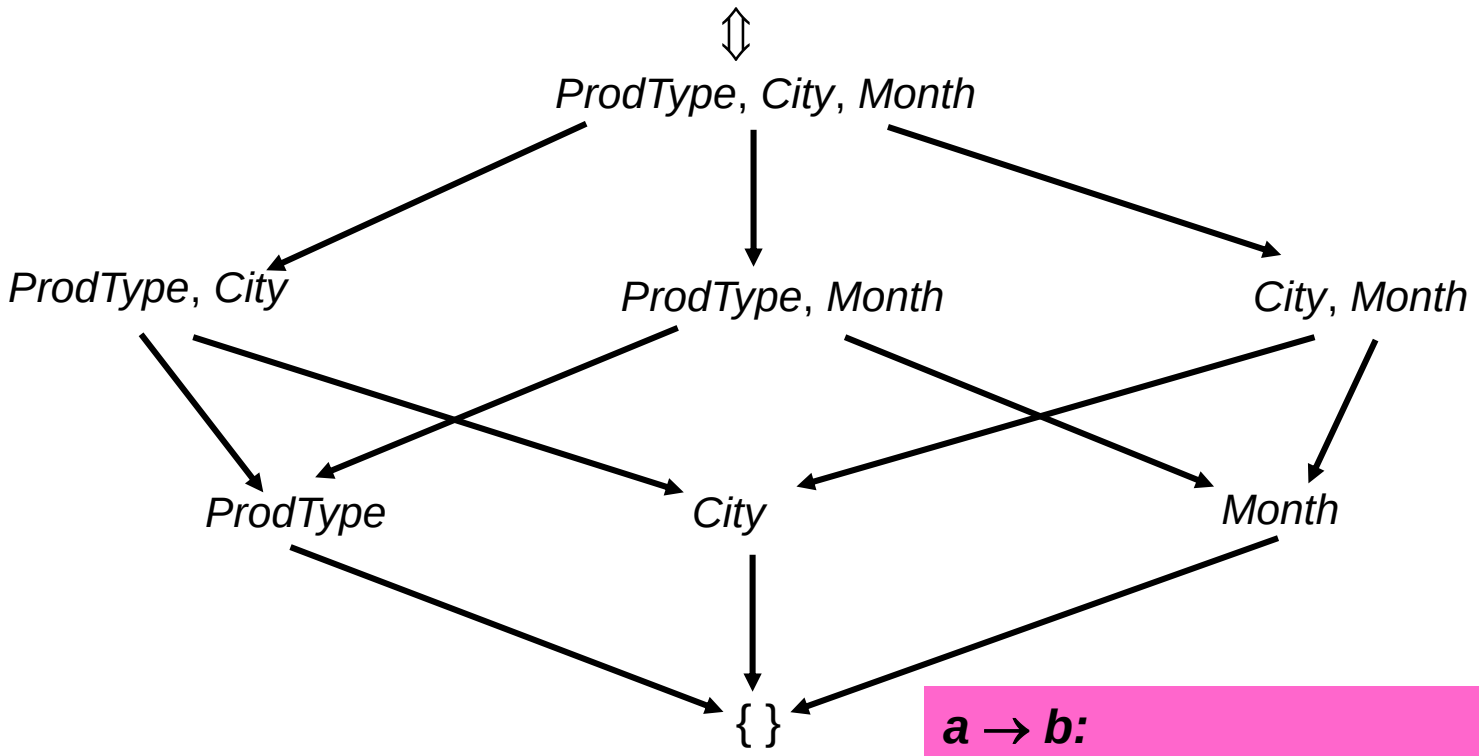
  with *Product*, *Location*, and *Time* dimensions
- The SQL statement:

```
SELECT ProdType, City, Month, SUM(Amnt)
FROM Slales NATURAL JOIN Time NATURAL JOIN
Location NATURAL JOIN Product
GROUP BY CUBE (ProdType, City, Month);
```

  will rollup the `Amnt` attribute on all eight subsets of the set {ProdType, City, Month}
- The CUBE clause will calculate the SUM of `Amnt` for each subset of the set {ProdType, City, Month}

CUBE Graph

```
SELECT ProdType, City, Month, SUM(Amnt) FROM Sales NATURAL
JOIN Product NATURAL JOIN Location NATURAL JOIN Time
GROUP BY CUBE (ProdType, City, Month);
```



$a \rightarrow b$:
b may be inferred from a

Question for you

- Given a star schema with four dimensions
- How many SELECT – SUM – GROUP BY statements will be replaced by a

  `SELECT – SUM – GROUP BY` **`CUBE`**

  statement?

SQL:1999 ROLLUP

- Standard SQL:1999 also offers a variant of GROUP BY CUBE clause to compute subsets of a cross-tabulation
- The SQL statement

```
SELECT City, Day, SUM(Amnt) AS Sale
FROM Sales NATURAL JOIN Location
 Natural join Time
GROUP BY ROLLUP (City, Day);
```

will compute SUM(Amnt) on {City, Day}, {City}, and {}

- Note, **ROLLUP** computes aggregates for the subsets produced by successive omitting the last component in the `GROUP BY` attribute list

Justification of the CUBE and ROLLUP

- The `CUBE` and `ROLLUP` clauses are introduced to provide for efficient computation of cross tabulation using cube cells of finer granularity to produce cube cells of coarser granularity
- If the `GROUP BY` list contained *n* attributes from different dimension tables, this way will be at least $2^n - 1$ joins saved, and aggregate functions will be computed using intermediate results (if possible), instead of starting each time from scratch

Some Queries not Supported by SQL/92

- Consider the following queries:
  1. Find the trailing *n* day moving average of sales,
  2. Find the top *n* products ranked by cumulative sales for every month over the past year
  3. Rank all products by total sales over the past year, and, for each product, display the difference in total sales relative to the product ranked behind it
- The first query asks to compute for each day the average daily sales over *n* preceding days
- All three queries have in common that they can not be computed using SQL/92
- SQL:1999 addresses this issue by introducing the `WINDOW` clause and some new aggregate functions

WINDOW Clause in SQL:1999

- `WINDOW` clause identifies an ordered window of rows around each tuple in a table
- This way we can apply an aggregate function on the rows around each row, and extend the row with the aggregate result
- Example:
  - Suppose each tuple records a day's sale
  - By applying `WINDOW` clause we can calculate the sum of sale for, say, past three days ($n = 3$) and append this result to each tuple
  - This gives a 3-day moving sum of sale

WINDOW Clause Versus GROUP BY

- `WINDOW` clause allows defining **overlapping** ordered groups of rows, to calculate some aggregates over the groups, and to append the result to each row

- `GROUP BY` defines disjoint partitions of tuples over a sorted table, calculates aggregates over the partitions, drops all but one tuple from each partition, and appends the result of an aggregate to the tuple retained

WINDOW – an Intuitive Example

TimeId	DayOfWeek	Sales	SUM(Sales)
1	Monday	55	00
2	Tuesday	66	55
3	Wednesday	33	121
4	Thursday	55	154
5	Friday	77	154
6	Saturday	22	165
7	Sunday	11	154
8	Monday	44	110

A moving SUM(Sales) window over past three days (ordering is done by TimeId)

WINDOW Syntax

```
SELECT <attribute_list_1>, <aggregate_function>
OVER W AS <moving_window_name>
FROM <table_list>
WHERE <join_and_select_conditions>
WINDOW W AS (PARTITION BY <attribute_list_2>
ORDER BY <attribute_list_3>
<window_frame_declaration> )
```

attribute_list_2 is a sub-list of attribute_list_1

attribute_list_3 is a sub-list of attribute_list_1

attribute_list_2 and attribute_list_3 are disjoint

window_frame_declaration defines the way of a window framing

NOTE: There is no `GROUP BY` clause in the syntax

Processing SELECT with WINDOW

- `FROM` and `WHERE` clauses are processed in the usual manner and they produce an intermediate table (call it *T*)
- There are three steps performed when defining window on *T* :
  1. Partitions are defined using `PARTITION BY` clause, which means that each partition contains tuples with the same `attribute_list_2` values
  2. Sorting of rows within a partition is defined using `ORDER BY` clause, so the partitions are sorted according to `attribute_list_3` values
  3. Framing the windows is done by establishing boundaries of the window associated with each row in a partition

Generating Query Result with WINDOW

- The result in the `moving_window_name` column is calculated by applying the aggregate function on all tuples in a window
- This result is appended to each answer tuple
- Note: tuples in the answer table are not sorted according to `attribute_list_3`
- An explicit `ORDER BY` clause at the end of the statement should be issued to perform sorting of the result
- Obviously, the result depends on the `window_frame_declaration`

Window Frame Declaration

- <window_frame_declaration> ::

```
ROWS | RANGE
       {
       {UNBOUNDED PRECEDING |
          <value expression> PRECEDING}
          | BETWEEN
          {UNBOUNDED PRECEDING |
           <value expression> PRECEDING}
          AND
          {CURENT ROW | UNBOUNDED FOLLOWING |
           <value expression > FOLLOWING}
         }
```

Parameters of the WINDOW Clause

- `ROWS | RANGE` define the window used for calculating the function result. The function is applied on all rows in the window
  - `ROWS` clause specifies a window in physical units (rows),
  - `RANGE` clause specifies a window as a logical offset
  - `BETWEEN… AND…` clauses let you specify a start point and an end point for a window, respectively

Value Expressions of the WINDOW Clause

- Valid value expressions are:
  - Constants,
  - Columns,
  - Nonanalytic functions, …
- A logical offset can be specified:
  - With constants, such as:
    - `RANGE 10 PRECEDING`
  - Or by an interval specification like:
    - `RANGE INTERVAL n DAYS | MONTH | YEARS PRECIDING`

Parameters of the WINDOW Clause

- `UNBONDED PRECEDING` specifies that the window starts at the first row of the partition or the first row of the result set
- `UNBOUNDED FOLLOWING` specifies that the window ends at the last row of the partition or the result set
- `CURRENT ROW` can specify either a start or an end point. In both cases it denotes that the window starts or ends at the current row
- DEFAULT start point is `UNBOUNDED PRECIDING`
- DEFAULT end point is `CURRENT ROW`

WINDOW Clause Cumulative Aggregate

```
SELECT AcctNo, Date, Amnt, SUM (Amnt)
OVER W AS Balance
FROM Ledger
WINDOW W AS (
    PARTITION BY AcctNo
    ORDER BY Date
    ROWS UNBOUNDED PRECEDING
)
ORDER BY AcctNo, Date;
```

WINDOW Clause Cumulative Aggregate

AcctNo	Date	Amnt	Balance
73829	2003-7-01	113.5	113.5
73829	2003-7-05	-52.01	61.44
73829	2003-7-13	36.25	97.69
82930	2003-7-01	10.56	10.56
82930	2003-7-21	32.55	43.11
82930	2003-7-29	-5.02	38.09

WINDOW Clause Moving Average

```
SELECT AcctNo, Date, Amnt, AVG (Amnt)
OVER w AS Moving_7_Days
FROM Ledger
WINDOW w AS (
    PARTITION BY AcctNo
    ORDER BY Date
    RANGE INTERVAL '7' DAY PRECEDING
    )
ORDER BY AcctNo, Date;
```

WINDOW Clause Moving Average

AcctNo	Date	Amnt	Moving_7_Days
73829	2003-7-03	113.5	113.5
73829	2003-7-09	-52.01	30.75
73829	2003-7-13	36.25	-7.88
73829	2003-7-14	10.56	-1.73
73829	2003-7-20	32.55	21.56
82930	2003-7-01	100.25	100.25
82930	2003-7-10	10.01	10.01
82930	2003-7-25	11.02	11.02
82930	2003-7-26	100.56	55.79
82930	2003-7-30	-5.02	35.52

WINDOW Clause Centered Aggregate

```
SELECT AcctNo, Date, Amnt, SUM (Amnt)
OVER W AS Balance
FROM Ledger
WINDOW W AS (
    PARTITION BY AcctNo
    ORDER BY Date
    RANGE BETWEEN INTERVAL '1' MONTH
    PRECEDING AND INTERVAL '1' MONTH
    FOLLOWING
)
ORDER BY AcctNo, Date;
```

WINDOW Clause Centered Aggregate

AcctNo	Date	Amnt	Balance
73829	2003-7-01	113.50	61.49
73829	2003-8-05	-52.01	97.74
73829	2003-9-13	36.25	-15.76
82930	2003-7-01	10.56	43.11
82930	2003-8-21	32.55	43.11
82930	2003-10-29	-5.02	-5.02

Mind the definition of the RANGE Clause. It is relative 1 month before and 1 month after the month of the current tuple (disregarding days).

New Aggregate Functions

- SQL:1999 introduces aggregate functions like:
  - Standard deviation,
  - Variance, and
  - Rank
- An approach to overcome the lack of special statistical functions in a DBMS is to allow users to define new operators
- Three components of a new operator have to be defined:
  - Initialization (initializes variables to be used in computation),
  - Iteration (updates variables as each tuple is retrieved), and
  - Termination (outputs the result)
- This is possible thanks to the UDF (user defined functions) feature of Object-Relational DBMSs

Summary

- OLAP requires a number of new features and commands for efficient processing of DSS queries
- These are:
  - `CUBE`,
  - `ROLLUP`,
  - `WINDOW`,
  - `RANK`
  - Adding new statistical functions
- All these features are supported by SQL:1999
- We considered `WINDOW` function as defined by Oracle
- PostgreSQL supports
  - `WINDOW` in a slightly restricted way, and
  - A great number of new built-in statistical functions