Data warehousing in Oracle

Materialized views and SQL extensions to analyze data in Oracle data warehouses

Data Base and Data Mining Group of Politecnico di Torino

SQL extensions for data warehouse analysis

Data Base and Data Mining Group of Politecnico di Torino

Available OLAP functions

- Computation windows
  - window
- Ranking functions
  - rank, dense rank, ...
- Group by clause extensions
  - rollup, cube, ...

Physical aggregation example

- Example table
  - SALES(**City**, **Date**, Amount)
- Select, separately for each city and for each date, the amount and the average amount over the current and the previous two rows

Physical aggregation example

```
SELECT Date, Amount,
   AVG(Amount) OVER (
        PARTITION BY City
        ORDER BY Date
        ROWS 2 PRECEDING
          ) AS MovingAverage
FROM Sales
ORDER BY Date;
```

Logical aggregation example

- Example table
  - SALES(**City**, **Date**, Amount)
- Select, separately for each city and for each date, the amount and the average amount over the current row and the sales of the two previous days

Logical aggregation example

```
SELECT Date, Amount,
   AVG(Amount) OVER (
        PARTITION BY City
        ORDER BY Date
        RANGE BETWEEN INTERVAL '2'
        DAY PRECEDING AND CURRENT ROW
        )  AS Last3DaysAverage
FROM Sales
ORDER BY Date;
```

Example tables

- Schema
  - SUPPLIERS(**Cod_S**, Name, SLocation)
  - ITEM(**Cod_I**, Type, Color, Weight)
  - PROJECTS(**Cod_P**, Name, PLocation)
  - FACTS(**Cod_S**, **Cod_I**, **Cod_P**, SoldAmount)

Ranking example

- Select for each item the total amount sold and the ranking according to the total amount sold

Ranking example

```
SELECT COD_I, SUM(SoldAmount),
RANK() OVER (
        ORDER BY SUM(SoldAmount)
        )  AS SalesRank
FROM Facts
GROUP BY COD_I;
```

Ranking example

COD_I	SUM(SoldAmount)	DenseSalesRank
I2	300	1
I5	1100	2
I4	1300	3
I6	1300	3
I1	1900	5
I3	4500	6

Dense ranking

```
SELECT COD_I, SUM(SoldAmount),
DENSE_RANK() OVER (
        ORDER BY SUM(SoldAmount)
        ) AS DenseSalesRank
FROM Facts
GROUP BY COD_I;
```

Ranking example

COD_I	SUM(SoldAmount)	DenseSalesRank
I2	300	1
I5	1100	2
I4	1300	3
I6	1300	3
I1	1900	4
I3	4500	5

Double ranking

- Select for each item the code, the weight, the total amount sold, the ranking according to the weight and the ranking according to the total amount sold

Double ranking

```
SELECT Item.COD_I, Item.Weight,
   RANK() OVER (ORDER BY Item.Weight
                         ) AS WeightRank
   RANK() OVER (ORDER BY SUM(SoldAmount)
                         ) AS SalesRank
FROM Facts, Item
WHERE Facts.COD_I = Item.COD_I
GROUP BY Item.COD_I, Item.Weight
ORDER BY WeightRank;
```

Double ranking

COD_I	Weigh	SUM(SoldAmount)	WeightRank	SalesRank
I1	12	1900	1	5
I5	12	1100	1	2
I4	14	1300	3	3
I2	17	300	4	1
I3	17	4500	4	6
I6	19	1300	6	3

Top N ranking selection

- Select
  - the **top two** most sold items
  - their code
  - their weight
  - the total amount sold
  - and their ranking according to the total amount sold

Top N ranking selection

- Returning only the top two items can be performed by nesting the ranking query inside an outer query
- The outer query uses the nested ranking query as a table (after the FROM clause)
- The outer query selects the requested values of the rank field

Top N ranking selection

```
SELECT * FROM
   (SELECT COD_I, SUM(SoldAmount),
     RANK() OVER (ORDER BY SUM(SoldAmount))
                          AS SalesRank
    FROM Facts
    GROUP BY COD_I)
WHERE SalesRank<=2;
```

SUPPLIERS(**Cod_S**, Name, SLocation)
ITEM(**Cod_I**, Type, Color, Weight)
PROJECTS(**Cod_P**, Name, PLocation)
FACTS(**Cod_S**, **Cod_I**, **Cod_P**, SoldAmount)

Top N ranking selection

```
SELECT * FROM
    (SELECT COD_I, SUM(SoldAmount),
      RANK() OVER (ORDER BY SUM(SoldAmount))
                          AS SalesRank
     FROM Facts
     GROUP BY COD_I)
WHERE SalesRank<=2;
```

Temporary table created at runtime
and dropped at the end of the outer query

ROW_NUMBER

- ROW_NUMBER
  - in each partition it assigns a progressive number to each row

- Partition the items according to their type and enumerate in progressive order the data in each partition. In each partition the rows are sorted according to the weight

ROW_NUMBER

```
SELECT Type, Weight,
    ROW_NUMBER() OVER (
        PARTITION BY Type
        ORDER BY Weight
        )  AS RowNumberWeight
FROM Item;
```

ROW_NUMBER

Type	Weight	RowNumberWeight	
Bar	12	1	Partition 1
Gear	19	1	Partition 2
Screw	12	1	Partition 3
Screw	14	2	
Screw	16	3	
Screw	16	4	
Screw	16	5	
Screw	16	6	
Screw	17	7	
Screw	17	8	
Screw	18	9	
Screw	20	10	

CUME_DIST

- CUME_DIST
  - in each partition it assigns a weight between 0 and 1 to each row according to the number of values which precede the value of the attribute employed for the sorting in the partition
- Given a partition with N rows, for each row x the CUME_DIST is computed as follows:
  - CUME_DIST(x) = number of values, which precede or have the same value of the attribute employed for the sorting, divided by N

CUME_DIST example

- Partition the items according to the type and sort in each partition according to the weight of items. Assign to each row the corresponding value of CUME_DIST

CUME_DIST example

```
SELECT Type, Weight,
   CUME_DIST() OVER (
      PARTITION BY Type
      ORDER BY Weight
      )  AS CumeWeight
FROM Item;
```

Example CUME_DIST

Type	Weight	RowNumberWeight		
Bar	12	1	(=1/1)	Partition 1
Gear	19	1	(=1/1)	Partition 2
Screw	12	0.1	(=1/10)	Partition 3
Screw	14	0.2	(=2/10)	
Screw	16	0.6	(=6/10)	
Screw	16	0.6	(=6/10)	
Screw	16	0.6	(=6/10)	
Screw	16	0.6	(=6/10)	
Screw	17	0.8	(=8/10)	
Screw	17	0.8	(=8/10)	
Screw	18	0.9	(=9/10)	
Screw	20	1	(=10/10)	

NTILE

- NTILE(n)
  - Allows splitting each partition in n subgroups (if it is possible) containing the same number of records. An identifier is associated to each subgroup.

NTILE example

- Partition the items according to the type and split each partition in 3 sub-groups with the same number of data. In each partition the rows are ordered by the weight of items

NTILE example

```
SELECT Type, Weight,
   NTILE(3) OVER (
      PARTITION BY Type
      ORDER BY Weight
      )  AS Ntile3Weight
FROM ITEM;
```

NTILE example

Type	Weight	RowNumberWeight		
Bar	12	1		Partition 1
Gear	19	1		Partition 2
Screw	12	1		Partition 3
Screw	14	1	Subgroup 1	
Screw	16	1		
Screw	16	1		
Screw	16	2		
Screw	16	2	Subgroup 2	
Screw	17	2		
Screw	17	3		
Screw	18	3	Subgroup 3	
Screw	20	3		

Materialized views

Data Base and Data Mining Group of Politecnico di Torino

Materialized views

- The result is **precomputed** and stored on the disk
- They improve **response times**
  - Aggregations and joins are precomputed
- Usually they are associated to queries with **aggregations**
- They may be used also for non aggregating queries
- Materialized views can be used as a **table** in any query

Query rewriting

- The DBMS can change the execution of a query to **optimize performance**
- Materialized views can be **automatically** used by the DBMS **without user intervention**
  - Materialized views help answering queries very similar to the query which created them

Creating materialized views

CREATE MATERIALIZED VIEW *Name*
[BUILD {IMMEDIATE|DEFERRED}]
[REFRESH {COMPLETE|FAST|FORCE|NEVER}
{ON COMMIT|ON DEMAND}]
[ENABLE QUERY REWRITE]
AS
Query

Creating materialized views

- *Name*
  - materialized view **name**
- *Query*
  - query associated to the materialized view (i.e., query that **creates** the materialized view)

Creating materialized views

- BUILD
  - IMMEDIATE
    - **creates** the materialized view and **immediately loads** the query results into the view
  - DEFERRED
    - **creates** the materialized view but does **not** immediately load the query results into the view

Creating materialized views

- REFRESH
  - COMPLETE
    - **recomputes** the query result by executing the query on **all data**
  - FAST
    - **updates** the content of the materialized view using the changes **since the last refresh**

Creating materialized views

- REFRESH
  - FORCE
    - when possible, the **FAST** refresh is performed
    - otherwise the **COMPLETE** refresh is performed
  - NEVER
    - the content of the materialized view is **not updated** using Oracle standard procedures

Materialized views options

- ON COMMIT
  - an **automatic refresh** is performed when SQL operations affect the materialized view content
- ON DEMAND
  - the refresh is performed only upon explicit **request** of the user issuing the command
    - DBMS_MVIEW.REFRESH

Materialized views options

- ENABLE QUERY REWRITE
  - enables the DBMS to automatically use the materialized view as a basic block (i.e., a table) to improve other queries performance
  - available only in the high-end versions of DBMS (e.g., not available in Oracle Express)
  - when unavailable, the query must be rewritten by the user to access the materialized view

Creation constraints

- Depending on the DBMS and the query, you can create a materialized view associated to the query if some constraints are satisfied
  - constraints on the aggregating attributes
  - constraints on the tables and the joins
  - etc.
  - you must be aware of the constraint existence!

Materialized view example

- Tables
  - SUPPLIERS(**Cod_S**, Name, SLocation)
  - ITEM(**Cod_I**, Type, Color)
  - PROJECTS(**Cod_P**, Name, PLocation)
  - FACTS(**Cod_S**, **Cod_I**, **Cod_P**, Measure)

Materialized view example

- The materialized view query is
  - SELECT Cod_S, Cod_I, SUM(Measure)
    FROM Facts
    GROUP BY Cod_S, Cod_I;
- Options
  - Immediate data loading
  - Complete refresh only upon user request
  - The DBMS can use the materialized view to optimize other queries

Materialized view example

```
CREATE MATERIALIZED VIEW Sup_Item_Sum
BUILD IMMEDIATE
REFRESH  COMPLETE ON DEMAND
ENABLE QUERY REWRITE
AS
    SELECT Cod_S, Cod_I, SUM(Measure)
    FROM Facts
    GROUP BY Cod_S, Cod_I;
```

Fast refresh

- Requires proper structures to log changes to the tables involved by the materialized view query
- MATERIALIZED VIEW LOG
  - there is a log for each table of a materialized view
  - each log is associated to a single table and some of its attributes
  - it stores changes to the materialized view table

Fast refresh

- The REFRESH FAST option can be used only if the materialized view query satisfies some constraints
  - materialized view **logs** for the tables and attributes of the query must exist
  - when the GROUP BY clause is used, in the SELECT statement an **aggregation** function must be specified (e.g., COUNT, SUM, ...)

Materialized view log example

- Create a materialized view log associated to the FACTS table, on Cod_S, Cod_I and MEASURE attributes
  - enable the options SEQUENCE and ROWID
  - enable new values handling

Materialized view log example

```
CREATE MATERIALIZED VIEW LOG
   ON Facts
WITH SEQUENCE, ROWID
   (Cod_S, Cod_I, Measure)
INCLUDING NEW VALUES;
```

Example with fast refresh option

- The materialized view query is
  - SELECT Cod_S, Cod_I, SUM(Measure)
    FROM Facts
    GROUP BY Cod_S, Cod_I;
- Options
  - Immediate data loading
  - Automatic fast refresh
  - The DBMS can use the materialized view to optimize other queries

Example with fast refresh option

```
CREATE MATERIALIZED VIEW LOG ON Facts
WITH SEQUENCE, ROWID (Cod_S, Cod_I, Measure)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW Sup_Item_Sum2
BUILD IMMEDIATE
REFRESH  FAST ON COMMIT
ENABLE QUERY REWRITE
AS
    SELECT Cod_S, Cod_I, SUM(Measure)
    FROM Facts
    GROUP BY Cod_S, Cod_I;
```

Fast refreshing materialized views

- The user or a system job can request the materialized view update by issuing the command
  - DBMS_MVIEW.REFRESH('*view*', {'C'/'F'})
    - *view*: name of the view to update
    - 'C': COMPLETE refresh
    - 'F': FAST refresh

Fast refreshing materialized views

- Example
  - COMPLETE refresh of the materialized view "Sup_Item_Sum"

```
EXECUTE DBMS_MVIEW.REFRESH('Sup_Item_Sum', 'C');
```

Changing and deleting views

- Changing
  - ALTER MATERIALIZED VIEW *name* *options*;
- Deleting
  - DROP MATERIALIZED VIEW *name;*

Analyzing materialized views

- The command DBMS_MVIEW.EXPLAIN_MVIEW allows the materialized view inspection
  - refresh type
  - operations on which the fast refresh is enabled
  - query rewrite status (enabled, allowed, disabled)
  - errors

Execution plan

- Analyzing the execution plan of frequent queries allows us to know whether materialized views are used
- Query execution plans can be shown
  - enabling the auto trace in SQLPLUS> **set autotrace on;**
  - clicking on the **Explain** link in the Oracle web interface

Results Explain Describe Saved SQL History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes
SELECT STATEMENT			5	1	14	65
HASH	GROUP BY		5	1	14	65
HASH JOIN			7,809	1	13	101,517