Data warehousing in Oracle

Materialized views and SQL extensions to analyze data in Oracle data warehouses
SQL extensions for data warehouse analysis
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

```sql
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

```sql
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(\textbf{Cod\_S}, Name, SLocation)
  - ITEM(\textbf{Cod\_I}, Type, Color, Weight)
  - PROJECTS(\textbf{Cod\_P}, Name, PLocation)
  - FACTS(\textbf{Cod\_S}, \textbf{Cod\_I}, \textbf{Cod\_P}, SoldAmount)
Ranking example

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

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

<table>
<thead>
<tr>
<th>COD_I</th>
<th>SUM(SoldAmount)</th>
<th>DenseSalesRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td>300</td>
<td>1</td>
</tr>
<tr>
<td>I5</td>
<td>1100</td>
<td>2</td>
</tr>
<tr>
<td>I4</td>
<td>1300</td>
<td>3</td>
</tr>
<tr>
<td>I6</td>
<td>1300</td>
<td>3</td>
</tr>
<tr>
<td>I1</td>
<td>1900</td>
<td>5</td>
</tr>
<tr>
<td>I3</td>
<td>4500</td>
<td>6</td>
</tr>
</tbody>
</table>
Dense ranking

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

<table>
<thead>
<tr>
<th>COD_I</th>
<th>SUM(SoldAmount)</th>
<th>DenseSalesRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td>300</td>
<td>1</td>
</tr>
<tr>
<td>I5</td>
<td>1100</td>
<td>2</td>
</tr>
<tr>
<td>I4</td>
<td>1300</td>
<td>3</td>
</tr>
<tr>
<td>I6</td>
<td>1300</td>
<td>3</td>
</tr>
<tr>
<td>I1</td>
<td>1900</td>
<td>4</td>
</tr>
<tr>
<td>I3</td>
<td>4500</td>
<td>5</td>
</tr>
</tbody>
</table>
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

```sql
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

<table>
<thead>
<tr>
<th>COD_I</th>
<th>Weigh</th>
<th>SUM(SoldAmount)</th>
<th>WeightRank</th>
<th>SalesRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>12</td>
<td>1900</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>I5</td>
<td>12</td>
<td>1100</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>I4</td>
<td>14</td>
<td>1300</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>I2</td>
<td>17</td>
<td>300</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>I3</td>
<td>17</td>
<td>4500</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>I6</td>
<td>19</td>
<td>1300</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>
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
  - 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;
<table>
<thead>
<tr>
<th>Type</th>
<th>Weight</th>
<th>RowNumberWeight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bar</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>Gear</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>Screw</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>Screw</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>Screw</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>Screw</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>Screw</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>Screw</td>
<td>17</td>
<td>7</td>
</tr>
<tr>
<td>Screw</td>
<td>17</td>
<td>8</td>
</tr>
<tr>
<td>Screw</td>
<td>18</td>
<td>9</td>
</tr>
<tr>
<td>Screw</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>
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:

- \( \text{CUME_DIST}(x) = \frac{\text{number of values, which precede or have the same value of the attribute employed for the sorting}}{N} \)
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

<table>
<thead>
<tr>
<th>Type</th>
<th>Weight</th>
<th>RowNumberWeight</th>
<th>Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bar</td>
<td>12</td>
<td>1</td>
<td>(=1/1)</td>
</tr>
<tr>
<td>Gear</td>
<td>19</td>
<td>1</td>
<td>(=1/1)</td>
</tr>
<tr>
<td>Screw</td>
<td>12</td>
<td>0.1</td>
<td>(=1/10)</td>
</tr>
<tr>
<td>Screw</td>
<td>14</td>
<td>0.2</td>
<td>(=2/10)</td>
</tr>
<tr>
<td>Screw</td>
<td>16</td>
<td>0.6</td>
<td>(=6/10)</td>
</tr>
<tr>
<td>Screw</td>
<td>16</td>
<td>0.6</td>
<td>(=6/10)</td>
</tr>
<tr>
<td>Screw</td>
<td>16</td>
<td>0.6</td>
<td>(=6/10)</td>
</tr>
<tr>
<td>Screw</td>
<td>16</td>
<td>0.6</td>
<td>(=6/10)</td>
</tr>
<tr>
<td>Screw</td>
<td>17</td>
<td>0.8</td>
<td>(=8/10)</td>
</tr>
<tr>
<td>Screw</td>
<td>17</td>
<td>0.8</td>
<td>(=8/10)</td>
</tr>
<tr>
<td>Screw</td>
<td>18</td>
<td>0.9</td>
<td>(=9/10)</td>
</tr>
<tr>
<td>Screw</td>
<td>20</td>
<td>1</td>
<td>(=10/10)</td>
</tr>
</tbody>
</table>
**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

<table>
<thead>
<tr>
<th>Type</th>
<th>Weight</th>
<th>RowNumber</th>
<th>Weight</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bar</td>
<td>12</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gear</td>
<td>19</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Screw</td>
<td>12</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Screw</td>
<td>14</td>
<td>1</td>
<td></td>
<td>Subgroup 1</td>
</tr>
<tr>
<td>Screw</td>
<td>16</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Screw</td>
<td>16</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Screw</td>
<td>16</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Screw</td>
<td>16</td>
<td>2</td>
<td></td>
<td>Subgroup 2</td>
</tr>
<tr>
<td>Screw</td>
<td>17</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Screw</td>
<td>17</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Screw</td>
<td>18</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Screw</td>
<td>20</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Partition 1**

**Partition 2**

**Partition 3**

**Subgroup 1**

**Subgroup 2**

**Subgroup 3**
Materialized views
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(\textbf{Cod}_S, Name, SLocation)
  - ITEM(\textbf{Cod}_I, Type, Color)
  - PROJECTS(\textbf{Cod}_P, Name, PLocation)
  - FACTS(\textbf{Cod}_S, \textbf{Cod}_I, \textbf{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
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.