OLAP Queries in SQL

Posted on October 18, 2020

This article first appeared in Data Science Briefings, the DataMiningApps newsletter. Subscribe now for free if you want to be the first to receive our feature articles, or follow us @DataMiningApps. Do you also wish to contribute to Data Science Briefings? Shoot us an e-mail over at briefings@dataminingapps.com and let's get in touch!

Contributed by: Bart Baesens, Seppe vanden Broucke

The term business intelligence (BI) is often referred to as the set of activities, techniques and tools aimed at understanding patterns in past data and predicting the future. In other words, BI applications are an essential component for making better business decisions through data-driven insight. These applications can both be mission critical or occasionally used to answer a specific business question. On-Line Analytical Processing (OLAP) provides a advanced set of BI techniques to analyze your data. More specifically, OLAP allows you interactively analyze the data, summarize it and visualize it in various ways. The term on-line refers to the fact that the reports can be updated with data almost immediately after they have been designed (or with negligible delay). The goal of OLAP is to provide the business-user with a powerful tool for ad-hoc querying.

To facilitate the execution of OLAP queries and data aggregation, SQL-99 introduced three extensions to the GROUP BY statement: the **CUBE**, **ROLLUP** and **GROUPING SETS** operator.

The **CUBE** operator computes a union of GROUP BY's on every subset of the specified attribute types. Its result set represents a multidimensional cube based upon the source table. Consider the following SALESTABLE:

PRODUCT	QUARTER	REGION	SALES
A	Q1	Europe	10
A	Q1	America	20
A	Q2	Europe	20
A	Q2	America	50
A	Q3	America	20
A	Q4	Europe	10
A	Q4	America	30
B	Q1	Europe	40
B	Q1	America	60
B	Q2	Europe	20
B	Q2	America	10
B	Q3	America	20
B	Q4	Europe	10
B	Q4	America	40

We can now formulate the following SQL query:

```
SELECT QUARTER, REGION, SUM(SALES)
FROM SALESTABLE
GROUP BY CUBE (QUARTER, REGION)
```

Basically, this query computes the union of $2^2 = 4$ groupings of the SALESTABLE being: {(quarter,region), (quarter), (region), ()}, where () denotes an empty group list representing the total aggregate across the entire SALESTABLE. In other words, since quarter has 4 values and region 2 values, the resulting multiset will have 4*2+4*1+1*2+1 or 15 tuples as you can see illustrated in the figure below. NULL values have been added in the dimension columns Quarter and Region to indicate the aggregation that took place. They can be easily replaced by the more meaningful 'ALL' if desired. More specifically, we can add two CASE clauses as follows

```
SELECT CASE WHEN grouping(QUARTER) = 1
THEN 'All'
ELSE QUARTER END AS QUARTER,
CASE WHEN grouping(REGION) = 1
THEN 'All' ELSE REGION END AS REGION,
SUM(SALES)
FROM SALESTABLE
GROUP BY CUBE (QUARTER, REGION)
```

The grouping() function returns 1 in case a NULL value is generated during the aggregation and 0 otherwise. This distinguishes the generated NULLs and the possible real NULLs stemming from the data. We will not add this to the subsequent OLAP queries so as to not unnecessarily complicate them.

Also, observe the NULL value for Sales in the fifth row. This represents an attribute combination which is not present in the original SALESTABLE since apparently no products were sold in Q3 in Europe. Remark that besides SUM() also other SQL aggregator functions such as MIN(), MAX(), COUNT() and AVG() can be used in the SELECT statement.

The result looks as follows:

QUARTER	REGION	SALES
Q1	Europe	50
Q1	America	80
Q2	Europe	40
Q2	America	60
Q3	Europe	NULL
Q3	America	40
Q4	Europe	20
Q4	America	80
Q1	NULL	130
Q2	NULL	100
Q3	NULL	40
Q4	NULL	90
NULL	Europe	110
NULL	America	250
NULL	NULL	360

Next, the **ROLLUP** operator computes the union on every prefix of the list of specified attribute types, from the most detailed up to the grand total. It is especially useful to generate reports containing both subtotals and totals. The key difference between the ROLLUP and CUBE operator is that the former generates a result set showing the aggregates for a hierarchy of values of the specified attribute types, whereas the latter generates a result set showing the aggregates for all combinations of values of the selected attribute types. Hence, the order in which the attribute types are mentioned is important for the ROLLUP but not for the CUBE operator. Consider the following query:

```
SELECT QUARTER, REGION, SUM(SALES)
FROM SALESTABLE
GROUP BY ROLLUP (QUARTER, REGION)
```

This query generates the union of three groupings {(quarter,region), (quarter}, ()} where () again represents the full aggregation. The resulting multiset will thus have 4*2+4+1 or 13 rows and is displayed in the figure below. You can see that the region dimension is first rolled up followed by the quarter dimension. Note the two rows which have been left out when compared to the result of the CUBE operator:

QUARTER	REGION	SALES
Q1	Europe	50
Q1	America	80
Q2	Europe	40
Q2	America	60
Q3	Europe	NULL
Q3	America	40
Q4	Europe	20
Q4	America	80
Q1	NULL	130
Q2	NULL	100
Q3	NULL	40
Q4	NULL	90
NULL	NULL	360

Whereas the previous example applied the GROUP BY ROLLUP construct to two completely independent dimensions, it can also be applied to attribute types that represent different aggregation levels (and hence different levels of detail) along the same dimension. For example, suppose the SALESTABLE tuples represented more detailed sales data at the individual city level and that the table contained three location related columns: City, Country and Region. We could then formulate the following ROLLUP query, yielding sales totals respectively per city, per country, per region and the grand total:

```
SELECT REGION, COUNTRY, CITY, SUM(SALES)
FROM SALESTABLE
GROUP BY ROLLUP (REGION, COUNTRY, CITY)
```

Note that in that case the SALESTABLE would include the attribute types City, Country and Region in a single table. Since the three attribute types represent different levels of detail in the same dimension, they are transitively dependent on one another, illustrating the fact that these data warehouse data are indeed denormalized.

The **GROUPING SETS** operator generates a result set equivalent to that generated by a UNION ALL of multiple simple GROUP BY clauses. Consider the following example:

```
SELECT QUARTER, REGION, SUM(SALES)
FROM SALESTABLE
GROUP BY GROUPING SETS ((QUARTER), (REGION))
```

This query is equivalent to:

```
SELECT QUARTER, NULL, SUM(SALES)
FROM SALESTABLE
GROUP BY QUARTER
UNION ALL
SELECT NULL, REGION, SUM(SALES)
FROM SALESTABLE
GROUP BY REGION
```

The result is given below.

QUARTER	REGION	SALES
Q1	NULL	130
Q2	NULL	100
Q3	NULL	40
Q4	NULL	90
NULL	Europe	110
NULL	America	250

Multiple CUBE, ROLLUP and GROUPING SETS statements can be used in a single SQL query. Different combinations of CUBE, ROLLUP and GROUPING SETS can generate equivalent result sets. Consider the following query

```
SELECT QUARTER, REGION, SUM(SALES)
FROM SALESTABLE
GROUP BY CUBE (QUARTER, REGION)
```

This query is equivalent to:

```
SELECT QUARTER, REGION, SUM(SALES)
FROM SALESTABLE
GROUP BY GROUPING SETS ((QUARTER, REGION), (QUARTER), (REGION), ())
```

Likewise, the following query

```
SELECT QUARTER, REGION, SUM(SALES)
FROM SALESTABLE
GROUP BY ROLLUP (QUARTER, REGION)
```

is identical to:

```
SELECT QUARTER, REGION, SUM(SALES)
FROM SALESTABLE
GROUP BY GROUPING SETS ((QUARTER, REGION), (QUARTER),())
```

SQL2003 introduced additional analytical support for two types of frequently encountered OLAP activities: ranking and windowing. **Ranking** should always be done in combination with an SQL ORDER BY clause. Assume we have the following table:

PRODUCT	SALES
A	50
B	20
C	10
D	45
E	40
F	30
G	60
H	20
I	15
J	25

Various ranking measures can now be calculated by using the following SQL query:

```
SELECT PRODUCT, SALES,
RANK() OVER (ORDER BY SALES ASC) as RANK_SALES,
DENSE_RANK() OVER (ORDER BY SALES ASC) as DENSE_RANK_SALES,
PERCENT_RANK() OVER (ORDER BY SALES ASC) as PERC_RANK_SALES,
CUM_DIST() OVER (ORDER BY SALES ASC) as CUM_DIST_SALES,
FROM SALES
ORDER BY RANK_SALES ASC
```

The result of this query is depicted below. The RANK() function assigns a rank based upon the ordered sales value whereby similar sales values are assigned the same rank. Contrary to the RANK() function, the DENSE_RANK() function does not leave gaps between the ranks. The PERCENT_RANK() function calculates the percentage of values less than the current value, excluding the highest value. It is calculated as (RANK() – 1) / (Number of Rows – 1). The CUM_DIST() function calculates the cumulative distribution or the percentage of values less than or equal to the current value.

Product	Sales	RANK_SALES	DENSE_RANK_SALES	PERC_RANK_SALES	CUM_DIST_SALES
C	10	1	1	0	0.1
I	15	2	2	1/9=0.11	0.2
B	20	3	3	2/9=0,22	0.4
H	20	3	3	2/9=0,22	0.4
J	25	5	4	4/9=0,44	0.5
F	30	6	5	5/9=0,55	0.6
E	40	7	6	6/9=0,66	0.7
D	45	8	7	7/9=0,77	0.8
A	50	9	8	8/9=0,88	0.9
G	60	10	9	9/9=1	1

All these measures can also be computed for selected partitions of the data. The measures depicted can also be computed for each region separately. Assuming the source table SALES now also includes a REGION attribute type, the query would then become

```
SELECT PRODUCT, SALES,
RANK() OVER (PARTITION BY REGION ORDER BY SALES ASC) as RANK_SALES,
DENSE_RANK() OVER (PARTITION BY REGION ORDER BY SALES ASC) as DENSE_RAN
PERCENT_RANK() OVER (PARTITION BY REGION ORDER BY SALES ASC) as PERC_RA
CUM_DIST OVER (PARTITION BY REGION ORDER BY SALES ASC) as CUM_DIST_SALE
FROM SALES
ORDER BY RANK_SALES ASC
```

Windowing allows calculating cumulative totals or running averages based on a specified window of values. In other words, windowing allows getting access to more than one row of a table without requiring a self-join. Consider the table depicted below.

QUARTER	REGION	SALES
1	America	10
2	America	20
3	America	10
4	America	30
1	Europe	10
2	Europe	20
3	Europe	10
4	Europe	20

The following query calculates the average sales for each region and quarter on the basis of the current, previous and next quarter.

```
SELECT QUARTER, REGION, SALES,
AVG(SALES) OVER
  (PARTITION BY REGION ORDER BY QUARTER ROWS
   BETWEEN 1 PRECEDING AND 1 FOLLOWING)
AS SALES_AVG
FROM SALES
ORDER BY REGION, QUARTER, SALES_AVG
```

The result is as such:

QUARTER	REGION	SALES	SALES_AVG
1	America	10	15
2	America	20	13,33
3	America	10	20
4	America	30	20
1	Europe	10	15
2	Europe	20	13,33
3	Europe	10	16,67
4	Europe	20	15

The **PARTITION BY REGION** statement subdivides the rows into partitions, similar to a GROUP BY clause. It enforces that the windows do not reach across partition boundaries. In other words, the SALES_AVG values will always be computed within a particular region. As an example, the SALES_AVG value for quarter 2 in America will be calculated as (10+20+10)/3=13,33 whereas for quarter 4 in America it is calculated as: (10+30)/2=20.

These are just a few examples of ranking and windowing facilities available in SQL. It is highly recommended to check the manual of the RDBMS vendor for more information. Furthermore, note that not all RDBMS vendors support these extensions. The ones that do support them usually also provide a user-friendly and graphical environment to construct OLAP reports using point-and-click which are then automatically translated by the tool into the corresponding SQL statements.

Given the amount of data to be aggregated and retrieved, OLAP SQL queries may get very time consuming. One way to speed up performance is by turning some of these OLAP queries into materialized views. For example, an SQL query with a CUBE operator can be used to precompute aggregations on a selection of dimensions of which the results can then be stored as a materialized view. A disadvantage of view materialization is that extra efforts are needed to regularly refresh these materialized views, although it can be noted that usually companies are fine with a close to current version of the data such that the synchronization can be done overnight or at fixed time intervals.

‹ Web Picks (week of 28 September 2020)   Web Picks (week of 5 October 2020) ›

Recent Posts

- Web Picks (week of 29 May 2023)
- What's Been Happening with Large Language Models
- Web Picks (week of 1 May 2023)
- Key Challenges in Analytics: Interview with Véronique Van Vlasselaer
- Web Picks (week of 17 April 2023)

Archives

- June 2023
- May 2023
- March 2023
- February 2023
- January 2023
- December 2022
- November 2022
- October 2022
- September 2022
- November 2021
- July 2021
- March 2021
- February 2021
- October 2020
- August 2020
- June 2020
- May 2020
- April 2020
- March 2020
- February 2020
- January 2020
- December 2019
- November 2019
- October 2019
- September 2019
- August 2019
- June 2019
- May 2019
- April 2019
- March 2019
- December 2018
- November 2018
- October 2018
- September 2018
- August 2018
- July 2018
- June 2018
- May 2018
- April 2018
- March 2018
- February 2018
- January 2018
- December 2017
- November 2017
- October 2017
- September 2017
- August 2017
- July 2017
- June 2017
- May 2017
- April 2017
- March 2017
- February 2017
- January 2017
- December 2016
- November 2016
- October 2016
- September 2016
- August 2016
- July 2016
- June 2016
- May 2016
- April 2016
- March 2016
- February 2016
- January 2016
- December 2015
- November 2015
- October 2015
- September 2015
- August 2015
- July 2015
- June 2015
- May 2015

KU Leuven, Department of Decision Sciences and Information Management
Naamsestraat 69, 3000 Leuven, Belgium
DataMiningApps on Twitter, Facebook, YouTube
info@dataminingapps.com