Aggregate tables with Firebird

Well-performing queries in OLAP scenarios with aggregate tables

Thomas Steinmaurer

(Originally published in German in “Entwickler Magazin” Edition 05.2008)

How many units of a particular product have been sold in Europe in Q1/2008? What was the sales volume of my subsidiary on the Island of Fiji in 2007? These are typical business critical questions, which management may demand from the back office. To answer these questions, the relevant data volume can be > 100 million records, especially for a big, world-wide operating company. Processing must be – of course – fast. For an IT department head or for a database administrator, this is quite a common requirement. You want to spend >20K Euro on licenses for “well-known” commercial database management systems (DBMS) to handle this requirement? No? Then read on.

The queries mentioned in the introduction are typical for OLAP (Online Analytical Processing) scenarios. From a technical implementation point-of-view, this is usually done with a Data Warehouse [1] (DWH). In such a system, aggregates play a very important role, because an OLAP query is usually an aggregated view on existing (relational) data. In SQL, you are probably familiar with aggregate functions like: *COUNT, SUM, AVG, MIN* and *MAX*.

This article shows how to speed up aggregated queries by using pre-aggregated data. It explains the concepts behind this, and describes a not that typical application domain for data warehousing. It is an entire solution based on Open Source technology, using the Firebird [2] DBMS and Mondrian [3], an Open Source OLAP Server. Mondrian does not have its own (multi-dimensional) storage engine, but follows the relational OLAP (ROLAP) paradigm, namely accessing data in an existing relational database.

Industrial DWH

The most common application domain for DWH is still: ”Everything related to sales statistics”. This is true, but not a necessity. Industrial manufacturing is an excellent example of a different application domain, where DWH concepts and implementations can also be applied. . Large amounts of process and measurement data, which are generated during a production process, must be integrated in a DWH for further analysis tasks. Important goals for the usage of a DWH in this application domain are:

- Improving the production process to minimize the frequency of faulty parts
- Trend/prediction analysis for durability of electric devices, which are used in the field by the customer

In both cases, the required data needs to be collected and integrated in a DWH and prepared for further data analysis tasks. Take for example the following question: “Give me the measured average temperature for a particular device for the year 2008, aggregated by the quarter.” If the average temperature is close to the maximum allowed temperature according to the device specification, durability of this device might be less, compared to using the device in other temperature ranges.

Anybody in the DWH?

The question clearly involves different kinds of data, which are relevant for data analysis tasks. For example:

- Uniquely identifiable device across the entire system
- Measurement value type (e.g. temperature, current, voltage)
- Measured value (e.g. 60 degree Celsius)
- Date / Time (e.g. 11.11.2007 / 15:34:32)

In a DWH, this will be modeled as a so-called star schema, which is illustrated in Figure 1.

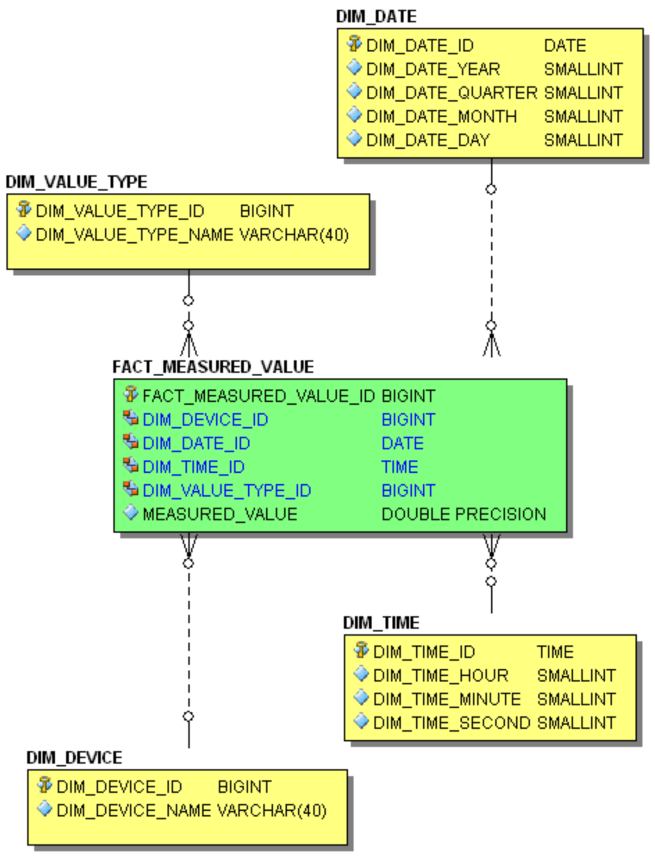


Figure 1: Star-Schema

The tables used are described in the following Table 1.

Table 1: Tables of the star-schema

Table	**Usage**
DIM_DATE	Dimension table Date. One record is a valid date with a year, quarter, month and day.
DIM_TIME	Dimension table Time. One record is a valid time with an hour, minute and second.
DIM_DEVICE	Dimension table Device. One record uniquely identifies a device in the DWH.
DIM_VALUE_TYPE	Dimension table measurement value type. One record is a type of measurement value. For example: temperature, current or voltage.
FACT_MEASURED_VALUE	Fact table. Stores the factum/measure, which needs to be analyzed. In our case this is a measured value of a particular type (DIM_VALUE_TYPE_ID) for a particular device (DIM_DEVICE_ID)

	at a specific timestamp (DIM_DATE_ID and DIM_TIME_ID).

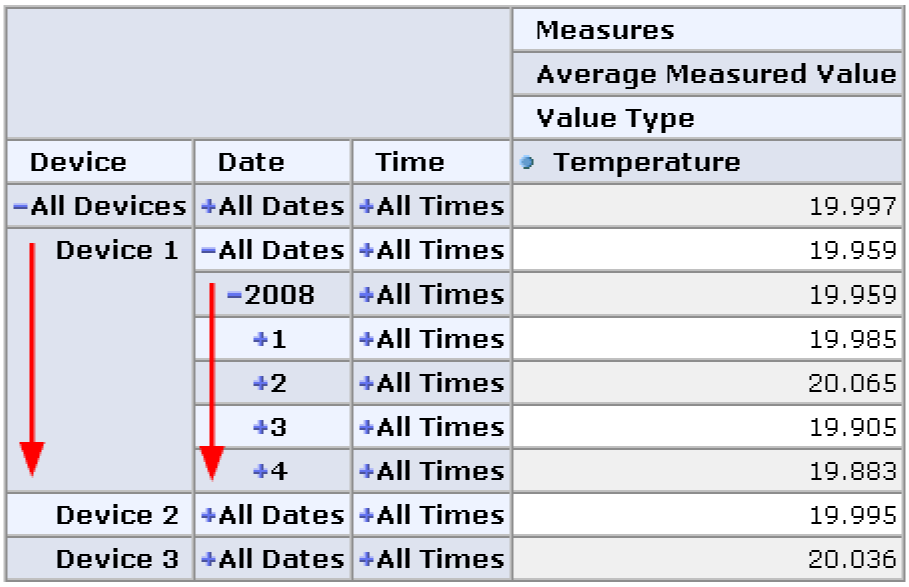
The requirements on an OLTP and OLAP database usually differ in respect to the number of concurrent connections, availability and configuration, so both systems are operated with their own separate databases. Loading the dimension tables and the fact table is done with an ETL (extraction, transformation, loading) process, which *extracts* the necessary data from the OLTP system, *transforms* the data based on defined rules for the target system und *loads* the data into the OLAP database.

In this example, I have not implemented an ETL process, but Open Source products, like Kettle [4], are available for that also. For this article, a configurable loading of the dimension tables and the fact table is done with Firebird stored procedures. To get a meaningful answer in respect to the execution time and the *Non-Indexed vs. Index-Reads* with and without aggregate tables, at least the fact table FACT_MEASURED_VALUE must be loaded with a substantial number of records. Table 2 shows the result after loading data into our OLAP database.

Table 2: Fill statistic of the tables in the star-chema

Table	**Usage**
DIM_DATE	Loading for the year 2008. => 366 records.
DIM_TIME	Loading for a day with second being the smallest unit. => 24 * 60 * 60 = 86.400 records.
DIM_DEVICE	Three devices. => 3 records.
DIM_VALUE_TYPE	Three measurement value types: voltage, current and temperature. => 3 records.
FACT_MEASURED_VALUE	For the entire year 2008, for every minute a record with a measured value will be generated for each device and each measurement value type. => 366 * 24 * 60 * 3 * 3 = 4.743.360 records.

To answer the question described above, a database developer can formulate an appropriate SQL query against the OLAP database or one can use an OLAP client application, which allows the query to be defined in a visual way through user interaction. The user is able to drill-down through the device and date dimensions to “navigate” to the expected result. This can be accomplished, for example, with a JPivot-based web application as shown in Figure 2.



			Measures
			Average Measured Value
			Value Type
Device	Date	Time	Temperature
-All Devices	+All Dates	+All Times	19.997
Device 1	-All Dates	+All Times	19.959
	-2008	+All Times	19.959
	+1	+All Times	19.985
	+2	+All Times	20.065
	+3	+All Times	19.905
	+4	+All Times	19.883
Device 2	+All Dates	+All Times	19.995
Device 3	+All Dates	+All Times	20.036

Figure 2: JPivot-based web application

The red arrows show the possible navigation path in both dimensions. By using the activated statement tracing in Mondrian, the executed SQL statements can be identified pretty quickly. Without using any aggregate tables, the SQL statement looks like:

```
select
    "DIM_VALUE_TYPE"."DIM_VALUE_TYPE_NAME" as "c0",
    "DIM_DATE"."DIM_DATE_YEAR" as "c1",
    "DIM_DATE"."DIM_DATE_QUARTER" as "c2",
    "DIM_DEVICE"."DIM_DEVICE_NAME" as "c3",
    avg("FACT_MEASURED_VALUE"."MEASURED_VALUE") as "m0"
from
    "DIM_VALUE_TYPE" "DIM_VALUE_TYPE",
    "FACT_MEASURED_VALUE" "FACT_MEASURED_VALUE",
    "DIM_DATE" "DIM_DATE",
    "DIM_DEVICE" "DIM_DEVICE"
where
    "FACT_MEASURED_VALUE"."DIM_VALUE_TYPE_ID" = "DIM_VALUE_TYPE"."DIM_VALUE_TYPE_ID"
and
    "DIM_VALUE_TYPE"."DIM_VALUE_TYPE_NAME" = 'Temperature' and
    "FACT_MEASURED_VALUE"."DIM_DATE_ID" = "DIM_DATE"."DIM_DATE_ID" and
    "DIM_DATE"."DIM_DATE_YEAR" = 2008 and
    "FACT_MEASURED_VALUE"."DIM_DEVICE_ID" = "DIM_DEVICE"."DIM_DEVICE_ID" and
    "DIM_DEVICE"."DIM_DEVICE_NAME" = 'Device 1'
group by
    "DIM_VALUE_TYPE"."DIM_VALUE_TYPE_NAME",
    "DIM_DATE"."DIM_DATE_YEAR",
    "DIM_DATE"."DIM_DATE_QUARTER",
    "DIM_DEVICE"."DIM_DEVICE_NAME"
```

Listing 1: OLAP SQL query without aggregate table usage

If I execute the SQL statement in a tool with my Desktop-PC, then I get the result back in approx. 1 minute and 17 seconds. The indexed vs. non-indexed reads for this SQL query in Figure 3 show an interesting result, namely a lot of indexed-reads on different tables. The DBMS has quite some work to do to return the expected result set.

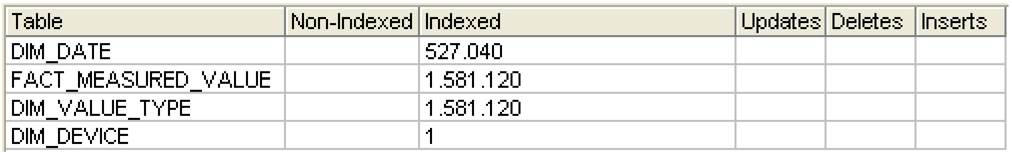


Table	Non-Indexed	Indexed	Updates	Deletes	Inserts
DIM_DATE		527.040			
FACT_MEASURED_VALUE		1.581.120			
DIM_VALUE_TYPE		1.581.120			
DIM_DEVICE		1			

Figure 3: Indexed vs. Non-Indexed reads without aggregate table

Aggregate tables as afterburner

The concept of an aggregate table is not a new development in the area of database technologies. “Enterprise-capable” DBMSs like Oracle, DB2 or Microsoft SQL Server support the persistence of a result set from a query, including different update strategies, in the event that the data in the base-table(s) changes. DBMS vendors often refer to this as *Materialized Views* or *Indexed Views*. In this article, I will use the term “aggregate table”, unless I am referring to a DBMS-specific implementation.

The main task of an aggregate table, based on an access pattern, is to store the result set in a physical table with far fewer records than the base table. In our case, the access pattern is the aggregation of the fact table through the date dimension down to the quarter level. Furthermore, perhaps an additional requirement exists from the QA department that queries down to the month level should run with good performance as well. The aggregate schema required for this is shown in Figure 4.

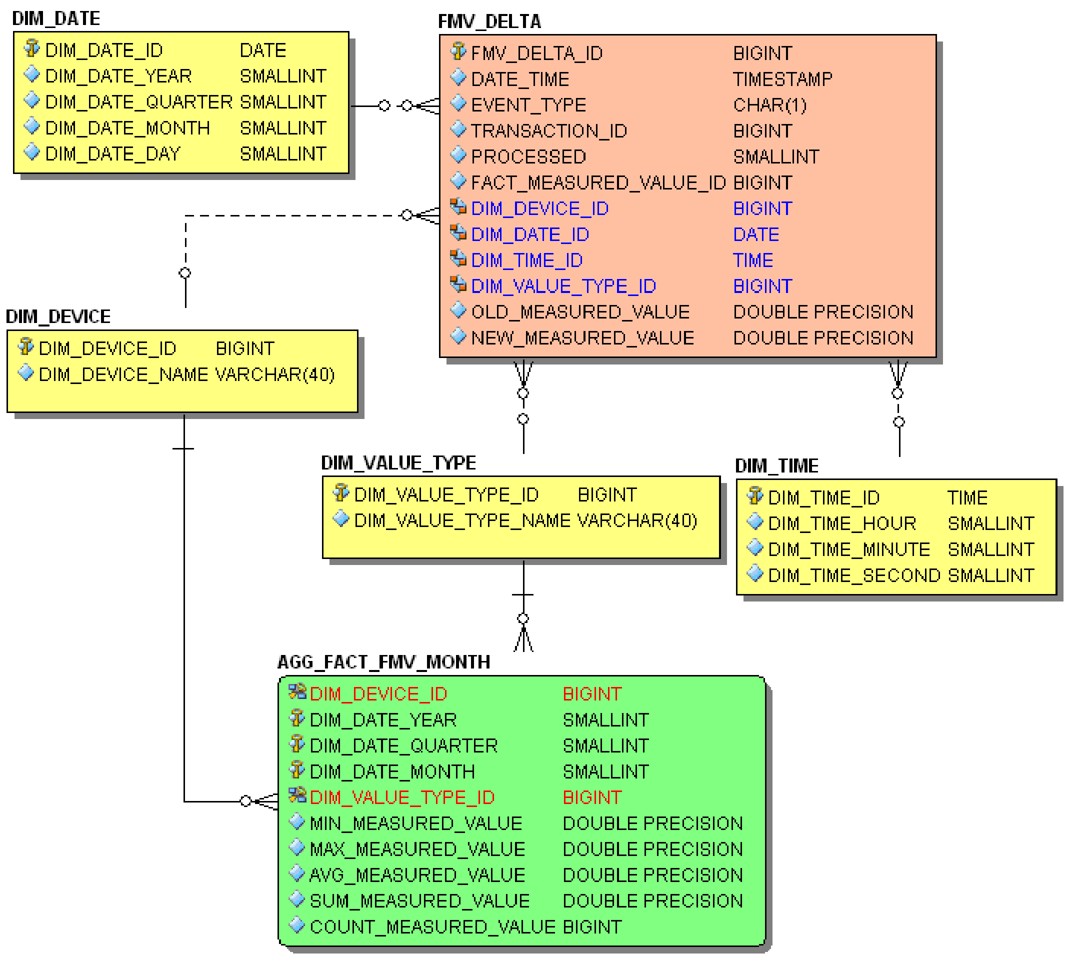


Figure 4: Aggregate Schema

The aggregate table AGG_FACT_FMV_MONTH is a slightly changed version of your fact table from the star-schema. The date dimension down to the month level is collapsed into the aggregate table and is not referenced by a foreign key constraint to the dimension table DIM_DATE anymore. Additionally, the pre-calculated aggregates *MIN, MAX, AVG, SUM* and the number of relevant records are stored in the aggregate table as well. The table FMV_DELTA is dealt with later, when different loading and update strategies of the aggregate table are discussed. For the time being, we simply assume a correctly loaded aggregate table. The performance when using the aggregate table to answer the same question now looks much more promising. The following listing shows the executed SQL statement.

```
select
    "DIM_VALUE_TYPE"."DIM_VALUE_TYPE_NAME" as "c0",
    "AGG_FACT_FMV_MONTH"."DIM_DATE_YEAR" as "c1",
    "AGG_FACT_FMV_MONTH"."DIM_DATE_QUARTER" as "c2",
    "DIM_DEVICE"."DIM_DEVICE_NAME" as "c3",
    sum("AGG_FACT_FMV_MONTH"."AVG_MEASURED_VALUE" *
"AGG_FACT_FMV_MONTH"."COUNT_MEASURED_VALUE") /
sum("AGG_FACT_FMV_MONTH"."COUNT_MEASURED_VALUE") as "m0"
from
    "DIM_VALUE_TYPE" "DIM_VALUE_TYPE",
    "AGG_FACT_FMV_MONTH" "AGG_FACT_FMV_MONTH",
    "DIM_DEVICE" "DIM_DEVICE"
where
    "AGG_FACT_FMV_MONTH"."DIM_VALUE_TYPE_ID" = "DIM_VALUE_TYPE"."DIM_VALUE_TYPE_ID"
and
    "DIM_VALUE_TYPE"."DIM_VALUE_TYPE_NAME" = 'Temperature' and
    "AGG_FACT_FMV_MONTH"."DIM_DATE_YEAR" = 2008 and
    "AGG_FACT_FMV_MONTH"."DIM_DEVICE_ID" = "DIM_DEVICE"."DIM_DEVICE_ID" and
    "DIM_DEVICE"."DIM_DEVICE_NAME" = 'Device 1'
group by
    "DIM_VALUE_TYPE"."DIM_VALUE_TYPE_NAME",
    "AGG_FACT_FMV_MONTH"."DIM_DATE_YEAR",
    "AGG_FACT_FMV_MONTH"."DIM_DATE_QUARTER",
    "DIM_DEVICE"."DIM_DEVICE_NAME"
```

Listing 2: OLAP SQL query with aggregate table usage

As you can see, the aggregate table AGG_FACT_FMV_MONTH and not the fact table FACT_MEASURED_VALUE is now used to get the expected result set, for the same drill-down user interaction. This SQL statement executed with a tool shows that the result set is returned in 30 milliseconds. The dramatically decreased number of reads is shown in Figure 5.

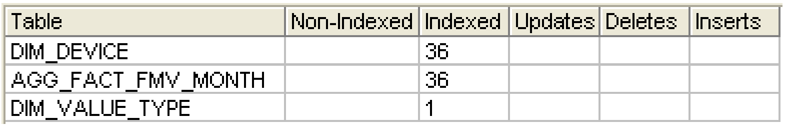


Table	Non-Indexed	Indexed	Updates	Deletes	Inserts
DIM_DEVICE		36			
AGG_FACT_FMV_MONTH		36			
DIM_VALUE_TYPE		1			

Figure 5: Indexed vs. Non-Indexed reads with aggregate table

Not a big surprise though, because the aggregate table holds only 12 * 3 * 3 = 108 records compared to 4.743.360 records in the fact table. This is a very beneficial optimization for this particular OLAP scenario.

Query Rewriting

Mondrian needs to know in the OLAP cube definition file that an aggregate table exists for the fact table. If this definition has been done properly, the OLAP server is able to transform the initial query so that the aggregate table and not the fact table will be queried. This mechanism is called *Query Rewriting*, which is an important component when using aggregate tables, because it ensures transparency for the user when using an OLAP client. The user should not need to know that there is an aggregate table. He/she simply fires off an OLAP query via a user-friendly OLAP client application and the OLAP server takes care of choosing the appropriate aggregate or fact table(s) for processing the query. If this component is clever, it can use an aggregate table even if there is no 1:1 mapping between the aggregated query and an existing aggregate table. For example, Mondrian is able to use our month-based aggregate table to carry out a quarter-based OLAP query.

If query rewriting is not supported by the OLAP server, then the usage of an aggregate table is not transparent to the user, because the user needs to know which aggregate tables exist in order to formulate the correct query. Be aware that many DBMS vendors support query rewriting in their high-priced “Enterprise-capable” editions only, even if they support materialized or indexed views in their less expensive editions!

Firebird as a DBMS does not support query rewriting at all. In our DWH architecture, another component is responsible for that, namely Mondrian. The user does not need to know about the existence of aggregate tables. Mondrian handles that behind the scenes.

Update strategies

With a well-designed aggregate schema, you can achieve a high performance gain, but, some compromise is necessary, because it doesn’t make sense to create an aggregate table for each possible query in your DWH environment. For example, you might choose to concentrate on the most used OLAP queries and improve their performance with aggregate tables first. Simply create your aggregate schema driven by real-use requirements.

With the introduction of aggregate tables, one is confronted with one topic pretty quickly, namely *redundancy*. In the case of aggregate tables, possibly the most important factor is whether pre-aggregated data is as up-to-date as the on-the-fly aggregate calculation of a query against the fact table. If data in the fact table gets changed, pre-aggregated data in an aggregate table is out-dated automatically. As a result, querying an aggregate table might produce a different result set than querying the fact table. Primarily, there are three different update strategies discussed in the literature and used in real-life DWH environments: *Snapshot*, *Eager* and *Lazy*.

When using the *snapshot* strategy, data in the aggregate table gets fully rebuilt by deleting and re-inserting records with up-to-date aggregations. The implementation of this strategy is very simple, but server utilization during a snapshot load increases with the number of records in the fact table. A snapshot load is usually done periodically, for example after loading the fact table with an ETL process. Starting a snapshot load could be the last action in an ETL process.

For the *eager* strategy, there is a delete/insert/update trigger on the fact table for each dependent aggregate table, which basically has some logic in place to re-calculate aggregates incrementally. The aggregate table does not need to be re-built from scratch every time, but the trigger simply updates the existing pre-aggregated values with the new values from the fact table accordingly. The implementation of this strategy is more complex, but still possible. A disadvantage of this approach is that the execution time of the ETL process is slower, because with each *COMMIT* the trigger on the fact table gets fired. The main advantage is that pre-aggregated data is always in-sync with the fact table.

The *lazy* approach has one trigger on the fact table, which logs any data changes on the fact table into a separate table (see table FMV_DELTA in Figure 4). Periodically, a stored procedure processes the log table and for each non-processed record, the eager mechanism gets executed. With this approach, there are again additional write operations necessary, namely into the log table, but the incremental update of pre-aggregated data can be done at a later, possibly better point of time. This approach is a mixture of the other two in respect to up-to-date data in the aggregate table(s) and server utilization.

Conclusion

Aggregate tables are a very interesting possibility to dramatically reduce the response time of aggregate queries in OLAP scenarios. DBMSs like Oracle, DB2 and Microsoft SQL Server have implementations for aggregate tables called Materialized or Indexed Views. Firebird currently (Version 2.1) does not have a comparable feature, but with support for triggers and stored procedures, Firebird can serve as a DBMS for a hand-made aggregate table implementation, including all three update strategies mentioned above. One gets an aggregate schema which has data redundancies, but they are all controllable. In combination with an OLAP server like Mondrian with support for query rewriting, a low-cost DWH system can be built entirely on top of Open Source software, which is suitable even for larger DWH projects.

Thomas Steinmaurer works as an Indusrial Researcher at the Software Competence Center Hagenberg (SCCH; Austria) [5] in the area of data management and data warehousing in the industrial application domain. Furthermore, he is responsible for the LogManager Series at Upscene Productions [6]. He is a co-founder of the Firebird Foundation [7]. The author can be contacted via one of the following email addresses: thomas.steinmaurer@scch.at or *t.steinmaurer@upscene.com.*

Links & Literature

[1] http://en.wikipedia.org/wiki/Data_warehouse
[2] http://www.firebirdsql.org
[3] http://mondrian.pentaho.org
[4] http://kettle.pentaho.org
[5] http://www.scch.at
[6] http://www.upscene.com
[7] http://www.firebirdsql.org/index.php?op=ffoundation