When is it Time to Rethink the Aggregate Configuration of Your OLAP Server?

Katja Hose, Daniel Klan, Matthias Marx, Kai-Uwe Sattler
Faculty of Computer Science and Automation
TU Ilmenau
Germany
{*first.last*}@tu-ilmenau.de

ABSTRACT

OLAP servers based on relational backends typically exploit materialized aggregate tables to improve response times of complex analytical queries. One of the key problems in this context is the view selection problem: choosing the optimal set of aggregation tables (called configuration) for a given workload. In this paper, we present a system that continuously monitors the workload and raises a quantified alert, when a better configuration is available. We address the tasks of query monitoring and view selection at the OLAP level instead of the SQL level, which simplifies the containment checks as well as rewriting and in this way helps to reduce the complexity of the backend system. At the demo we plan to show how our system works, i.e., how the system reacts upon arbitrary (interactive) workloads and how the user is alerted that a better configuration is available.

1. INTRODUCTION

OLAP servers are powerful tools to answer analytical queries in data warehouses. An OLAP server represents data in a multidimensional cube and usually supports a multidimensional query language (either as a real query language such as Microsoft's MDX or as a visual query interface). A typical implementation variant is ROLAP where the cube is mapped to a set of base tables in a relational DBMS modeled as star or snowflake schema. Hence, a multidimensional OLAP query has to be translated into one or a sequence of SQL queries following the star join pattern and usually containing complex grouping and aggregation operations.

In order to speed up query processing so-called aggregation tables are used. These tables are (partial) materializations of query results. To decide which results to materialize the cost-benefit tradeoff has to be taken into account: aggregation tables improve query response time but require additional disk space and maintenance costs. The problem of identifying the optimal set of aggregation tables is called the view selection problem and due to the large number of possible aggregations known to be NP complete [7].

VLDB '08, September 24-30, 2008, Auckland, New Zealand.

Nowadays, commercial DBMS support aggregation tables in the form of materialized views (Oracle), materialized query tables (DB2), or indexed views (SQL Server). They provide sophisticated techniques for query rewriting, incremental maintenance, and – usually implemented as an external administration tool – recommending aggregation tables [1,4,12]. However, particularly the latter step is still a static solution that has to be carried out manually and repeated after changes in the data or the workload. In contrast, other aspects of physical design tuning (e.g., index tuning) are successfully moved to an online approach allowing a continuous and more autonomous tuning [3, 9, 10].

Unfortunately, determining the optimum set of aggregation tables given limited space is a hard problem and a very expensive process. Thus, such an algorithm should be run as rarely as possible. In order to determine the necessity to run such a reoptimization step and similar to the alerter approach for index tuning proposed in [2], we propose a light-weight alerter approach that indicates whether there is a configuration that would be better with respect to the current configuration. In summary, the system we present is characterized by the following points:

- The selection of candidate materializations is driven by the queries processed by the OLAP server.
- We simplify the problem of query containment checks by carrying out the view selection at the level of MDX queries instead of SQL.
- The system alerts the user if a better configuration with respect to the current workload is available and outputs one such configuration.
- The alert is quantified by the expected cost reduction with respect to the current configuration and the workload.

By taking the view selection problem as well as the query rewriting problem out of the relational backend we expect a more simple but still powerful solution as we exploit the semantics or patterns of the analytical queries. Furthermore, we argue that this helps to reduce the complexity of the query engine in the spirit of [5]. Finally, we provide the functionality of aggregation tables also for DBMS not supporting materialized views natively (e.g., open source systems). The system is implemented as part of the open source OLAP server Mondrian [6] and can run on any relational DBMS providing a cost-based optimizer.

2. THE ALERTER APPROACH

The goal of our work is to support the aggregate configuration of an OLAP server by (1) continuously monitoring information about the workload and the benefit of aggregation tables and (2) alerting the user or DBA if changes to the current configuration would be beneficial. As in the alerter approach for indexes [2] we "prove" this alert by presenting at least one new configuration that is better than the current one for the current workload situation.

In contrast to previous work on recommenders for materialized views and cubes our alerter is part of an OLAP engine (in our case the Mondrian OLAP server) and not of the relational DBMS. We have chosen this approach for the following reasons:

- Because we work at the level of MDX queries the steps of determining corresponding aggregation tables and analyzing derivability relationships between aggregation tables and queries are much easier than at the SQL level for arbitrary queries.
- OLAP servers such as Mondrian support rewriting of queries in order to exploit aggregation tables even if the underlying backend DBMS does not support materialized views. Thus, an alerter for the OLAP server supports such DBMS (which includes all open source DBMS), too.
- We are able to exploit information about the multidimensional schema such as measure attributes, aggregation levels etc., which are not available at the SQL level or would require language extensions.
- Finally, we can collect query execution costs (at least for certain queries) instead of only relying on estimations, because the queries are processed anyway by the OLAP server.

The alerter is conceptually based on two main components: the aggregation lattice, which captures the aggregation nodes (materialized or not) as well as their relationships and a cost matrix to collect costs for each query / aggregation node pair. Each cell ij in this matrix (Fig. 1) represents the costs $c_{i,j}$ for processing query Q_i on the aggregation table M_j. The column M_0 represents the base relations. In addition, for each query a query counter is maintained.

		Q_1	Q_2	...	Q_n
		cnt_1	cnt_2	...	cnt_n
M_0					
M_1					
...					
M_n				...	

Figure 1: Cost matrix

In contrast to static optimization approaches, the aggregation lattice is not constructed entirely in advance. Instead, the following steps are performed:

1. Each processed MDX query is intercepted.
2. For each query the corresponding aggregation node is inserted into the lattice (if it does not already exist) by taking the derivability relationship to other nodes into account. The corresponding aggregation node can easily be extracted from the MDX query by identifying the lowest aggregation level for each queried dimension. Fig. 2 illustrates how derivability can be checked efficiently by assigning coordinates to each node in the lattice. For this purpose, hierarchy levels are assigned numbers, e.g., day-month-year corresponds to 1-2-3. In our example lattice this means that the node corresponding to the aggregation by month and city is assigned the coordinates (2,3). A node a is derivable from a node b if: $\forall i \in D : b[i] \leq a[i]$ where D is the set of dimensions and $b[i]$ denotes the coordinate of node b in dimension i.

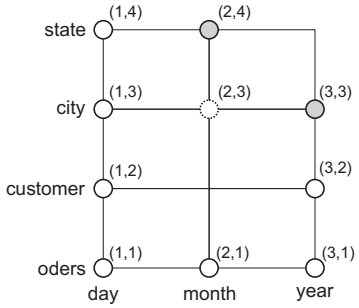


Figure 2: Aggregation lattice with coordinates

3. In order to consider also common base nodes in the lattice for which no query has been processed so far, we can again use the coordinates illustrated in Fig. 2: Using these coordinates and given two nodes a and b we can determine the "greatest" node g from which both can be derived by simply choosing the minimal coordinates: $\forall i \in D : g[i] = \min\{a[i], b[i]\}$.
4. The execution costs for the query on each aggregation node on the path(s) from the newly inserted one to the node representing the base relations are collected in the cost matrix, i.e., all nodes that the new one can be derived from are concerned. These costs are estimated as follows:
   Since queries are processed anyway, we get the exact execution costs and some statistics for each query Q_i. With this, we can estimate the costs for a table scan on an aggregation table using the cardinality of the query result set, the average tuple size, the fill factor of the page, and the page size. That is, we estimate the costs for a query Q_i on the aggregation table M_i with

$$c_{i,i} = \frac{|Q_i| \cdot tuplesize \cdot fillfactor}{pagesize}$$

   If a query Q_k is processed on an aggregation table M_i, the DBMS reads the table and computes some additional aggregation functions and projections. In general, the execution costs are dominated by the costs for the table scan. Aggregation and projection costs are not significant. Since computing query Q_j on the aggregation table M_i is a simple table scan followed by necessary aggregations and projections (joins are

eliminated by the aggregation table), we can use the following heuristic to estimate the costs for a query Q_j on an aggregation table M_i:

$$c_{j,i} = \begin{cases} c_{i,i} & Q_j \text{ is derivable from } M_i \\ c_{j,0} & \text{else} \end{cases}$$

Based on the cost matrix the benefit of an aggregation table M_i can be calculated by taking the costs of all queries into account, for which no (materialized) aggregation node $M_k \neq M_i$ with lower costs exists:

$$\begin{aligned} benefit(M_i, Q_j) &= (c_{j,0} - c_{j,i}) \cdot cnt_j \\ benefit(M_i) &= \sum_j benefit(M_i, Q_j) \text{ with } \nexists M_k : k \neq i, \\ & \quad k > 0 \wedge M_k \text{ is materialized } \wedge c_{j,k} < c_{j,i} \end{aligned}$$

The optimization goal is now to find a configuration $\mathcal{C} \subseteq \{M_0, M_1, \ldots, M_n\}$ that maximizes the benefit. For this purpose, several algorithms have been presented in the literature [7, 8, 11].

However, these are rather expensive algorithms. Thus, for an alerter we are interested in answering the question: Is there a better configuration than the current one? For this purpose, we assume an existing configuration and consider minimal changes, i.e., adding a single new aggregation node to the current configuration (and eventually drop a less beneficial one). Checking for a new configuration is triggered by the following events:

- a query counter in the cost matrix has changed,
- a new row or column was inserted into the cost matrix.

In both cases the benefits of the affected aggregation nodes are recalculated. If all affected rows represent already materialized aggregation tables, there is no need to change the configuration. When triggered, the alerter evaluates all possible new configurations $C_1, C_2, \ldots$ which can be constructed by adding one of the affected aggregation nodes to the current configuration C_0. For this purpose, the benefit deltas are calculated and the configuration C_i with the highest improvement of the overall benefit is returned to the user. This allows him to decide if a reconfiguration is necessary.

In our prototype several strategies for dealing with materialization costs as well as workload drifts are implemented and can be activated in the alerter. First, the materialization costs for new aggregation tables can be taken into account as negative benefit. Second, a simple LRU-k strategy for aging the query counters can be applied to reduce the impact of older queries.

To avoid missing a better configuration simply due to a restrictive limit for the configuration size, we apply a principle from economics: Gossen's First Law or the concept of diminishing marginal utility. It basically says that although the total utility of a good increases as more of it is consumed, the additional satisfaction (the marginal utility) usually decreases with additional consumptions if a certain threshold of satisfaction is reached. For the problem of view selection, disk space is the good and the benefit of views represents the utility. That means, we allow to add a new aggregation table even if the total size (i.e., the required disk space) of the configuration is exceeded as long as the slope of the line connecting two benefit values in the function of benefit over the size is above a certain threshold.

3. ARCHITECTURE

In this section, we illustrate the implementation of our alerter approach as part of the Mondrian OLAP server. The Mondrian server is running on a DB2 database backend. Mondrian itself uses a browser-based frontend for user interaction (Fig. 4(a)). For this purpose, the OLAP query result is rendered by the JPivot[1] library.

We extended Mondrian by an alerter, an admin console, and a configuration recommender. The alerter is based on an aggregation monitor, which observes the workload and propagates changes to the aggregation lattice and the cost matrix. The alerter component monitors the cost matrix and notifies the admin console if there exists a better configuration than the current one. In that case, the choice is upon the user: keep the current configuration or change it. On the one hand, there is the fast but not optimal configuration proposed by the alerter as it considers only changes detected by the alerter. On the other hand, there is the configuration recommender, which uses a greedy-based optimization strategy to determine the best configuration for the current workload. The configuration recommender may find the optimal configuration but it is very expensive. The architecture of the overall system is shown in Fig. 3.

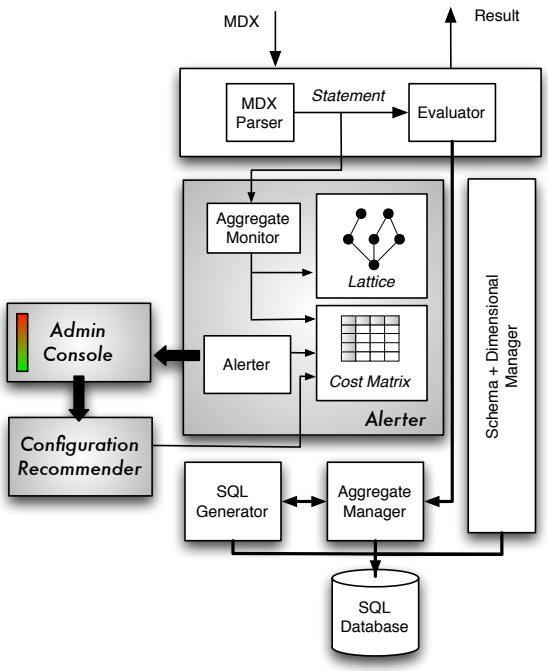
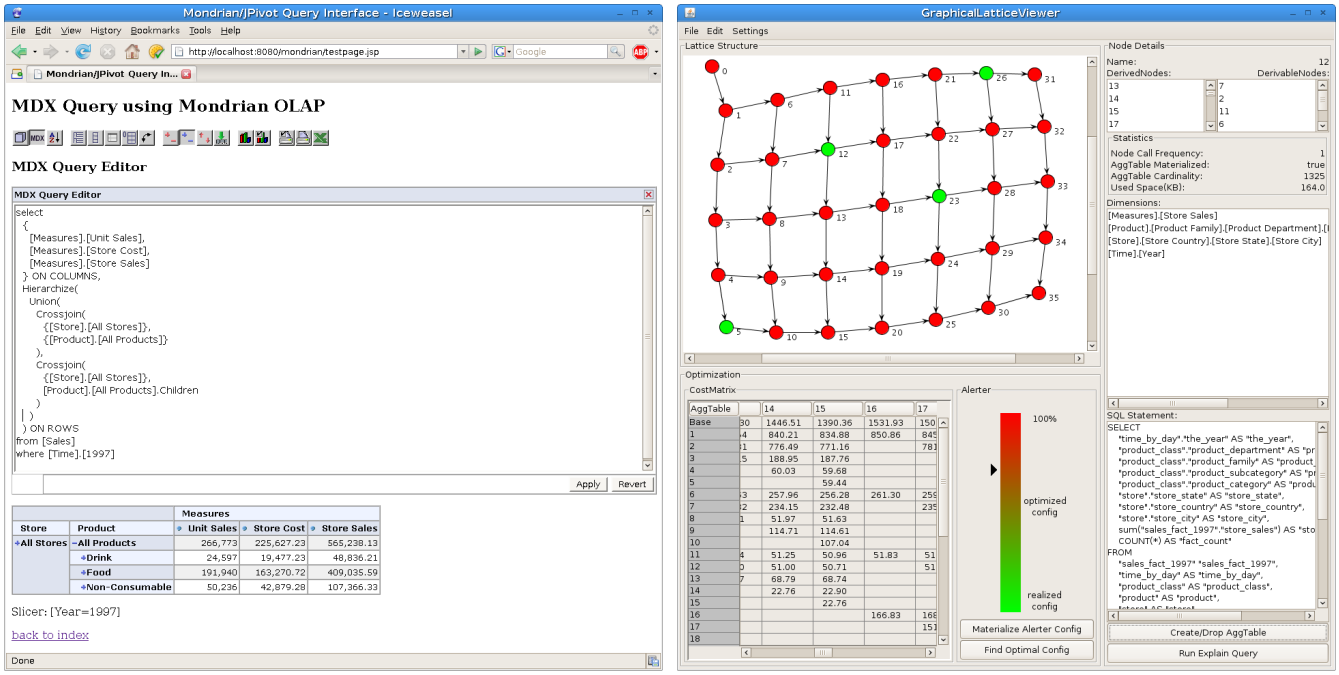


Figure 3: Architecture of the demonstration system

4. DEMONSTRATION

For our demonstration we use two different kinds of workloads on a data cube: (i) a batch of MDX queries and (ii) an interactive session of OLAP queries using Mondrian's

[1] http://jpivot.sourceforge.net/



(a) Mondrian's browser-based frontend

(b) Admin console

Figure 4: Screenshots

browser-based frontend. Using these workloads we demonstrate the online collection and evaluation of information about the workload and the benefit of (potential) aggregation tables. Both workloads are designed to contain shifts requiring an adaptation of the aggregation configuration. The need for a reconfiguration is observed by the alerter that intercepts all incoming queries.

At the demonstration, we provide a console application that visualizes the aggregation lattice and the cost matrix (Fig. 4(b)). Red nodes in the lattice represent non-materialized aggregation tables and green nodes represent materialized aggregation tables, i.e., the summary of all green nodes corresponds to the running configuration. On the right hand side of Fig. 4(b) some node statistics (cardinality, space consumption) and the corresponding MDX and SQL queries are displayed. The cost matrix shows the cost and benefit information of all nodes.

The alerter state is visualized by a gauge showing the benefit of a new configuration (if any exists). The gauge visualizes the differences between the current (running) configuration and the best configuration recommended by the alerter scaled to a user defined presetting.

The user is now able to extract the configuration recommended by the alerter or initiate an exhaustive optimization. We can prove the validity of the alerter's recommendation by re-running the captured workload on the optimized configuration.

5. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB '00*, pages 496–505, 2000.

[2] N. Bruno and S. Chaudhuri. To Tune or not to Tune? A Lightweight Physical Design Alerter. In *VLDB 06*, pages 499–510, 2006.

[3] N. Bruno and S. Chaudhuri. An Online Approach to Physical Design Tuning. In *ICDE '07*, pages 826–835, 2007.

[4] S. Chaudhuri and V. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB '97*, pages 146–155, 1997.

[5] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *VLDB '00*, pages 1–10, 2000.

[6] P. Corporation. Pentaho Analysis Services: Mondrian Project. http://mondrian.pentaho.org, 2007.

[7] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing Data Cubes Efficiently. *SIGMOD '96*, pages 205–216, 1996.

[8] M. Lee and J. Hammer. Speeding Up Materialized View Selection in Data Warehouses Using a Randomized Algorithm. *Int. J. Cooperative Inf. Syst.*, 10(3):327–353, 2001.

[9] M. Lühring, K. Sattler, K. Schmidt, and E. Schallehn. Autonomous Tuning with Soft Indexes. In *SMDB '07*, pages 450–458, 2007.

[10] K. Sattler, I. Geist, and E. Schallehn. QUIET: Continuous Query-driven Index Tuning. In *VLDB '03*, pages 1129–1132, 2003.

[11] C. Zhang and J. Yang. Genetic Algorithm for Materialized View Selection in Data Warehouse Environments. In *DaWaK '99*, pages 116–125, 1999.

[12] D. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB '04*, pages 1087–1097, 2004.