

Online Tuning of Aggregation Tables for OLAP

Katja Hose, Daniel Klan, Kai-Uwe Sattler

Database & Information Systems Group
Ilmenau University of Technology, Germany
 first.last@tu-ilmenau.de

Abstract—Materializing results from complex aggregation queries helps to significantly improve response times in OLAP servers. This problem is known as the view selection problem: choosing the optimal set of aggregation tables (called configuration) for a given workload. In this paper we present an online approach for adjusting the configuration dynamically to the current workload. This approach is implemented as part of an open source OLAP server and acts on the level of multidimensional MDX queries. The work presents the details of cost estimation and optimization of the system demonstrated in [10] and extends it by an online tuning strategy.

I. INTRODUCTION

OLAP servers are powerful tools for answering analytical queries in decision support applications. Usually, they represent data in a multidimensional cube and provide appropriate query facilities, either through a visual query interface or a multidimensional query language like Microsoft's MDX. Beside native, proprietary multidimensional storage schemes, relational backends based on SQL DBMS play an important role in building OLAP servers. Here, the cube is mapped to a fact table and a set of dimension tables based on the star or snowflake schema. Thus, multidimensional OLAP queries have to be translated into SQL queries (either a single one or even a sequence of queries). Though the current SQL standard defines a set of advanced features for OLAP support (OLAP functions, advanced grouping) which are already available in commercial DBMS, these queries are typically very expensive – simply due to the huge fact table, the high number of joins with dimension tables, and the grouping/aggregation operations, which are often performed using expensive external sorting.

In order to speed-up processing such queries, results from frequently asked queries can be materialized in so-called aggregation tables. Commercial DBMS implement this as materialized views (Oracle), materialized query tables (DB2), or indexed views (SQL Server). A big challenge in this context is to decide which query results should be materialized – ideally, the set of aggregation tables is used that provides the most benefit for the whole workload and that has the least costs in terms of disk space and maintenance. The problem of identifying the optimal set of aggregation tables under space constraints is called the view selection problem and known to be NP complete. To solve this combinatorial optimization problem, several greedy or randomized algorithms have been proposed in the literature. Moreover, commercial DBMS vendors provide administration tools supporting the selection of aggregation tables in the form of design advisors.

However, two problems remain to be solved:

- 1) All mentioned approaches are still static solutions that have to be carried out manually and repeated after changes in the workload or the data. This might be feasible in scenarios with static workloads known in advance. But interactive OLAP tools in particular generate queries dynamically which makes it difficult to capture a representative workload. In addition, queries may change over time, e.g., at the end of a quarter analysts could be interested in other measures than at other times. In contrast, for other aspects of physical design tuning (particularly index tuning) online approaches aiming to continuous tuning have been applied successfully.
- 2) The second problem is the complexity of the view selection problem. Besides the huge search space due to the large number of possible aggregations, a central problem is derivability of aggregations. Particularly for the general case of SQL queries, query containment checking is a difficult task. Furthermore, because materializing all possible aggregation candidates is not possible for practical reasons, cost-relevant information about these non-existing tables have to be estimated, which is also difficult for the general case.

In this paper, we address these problems by proposing an online approach for recommending aggregation tables which is characterized by the following contributions:

- (1) We present an online algorithm for the view selection problem.
- (2) By shifting the containment problem out of the relational backend into the OLAP server Mondrian¹ to the level of multidimensional queries, we are able to simplify containment and derivability checks and to exploit semantics and patterns of analytical queries.
- (3) We support efficient selection of aggregation tables by applying a cost model that exploits appropriate heuristics and estimations.

Our work is based on two assumptions:

- OLAP queries are processed by an OLAP server (by translating them into SQL) that is able to exploit materialized query results even if the underlying backend does not support query rewriting on materialized views.
- We restrict ourselves to aggregation tables and ignore indexes for two reasons: first, index tuning is a well-studied problem and the combination with aggregation

¹<http://mondrian.pentaho.org>

tables is straightforward. Second, though our current implementation is still based on traditional SQL backends like DB2 and PostgreSQL, we plan to address non-standard backends for OLAP which make indexes obsolete, e.g., MonetDB [3] or BI accelerators.

This paper extends the work presented in [10] by giving the details of the underlying cost and decision model as well as its evaluation. Furthermore, in this paper we focus on the problem of online optimization in contrast to the alterer approach described in [10].

The remainder of this paper is organized as follows. After a brief survey on related work in Section II we discuss collection and maintenance of relevant cost information in Section III. In Section IV we present an online algorithm to solve the view selection problem continuously. Section V describes the implementation of our approach as part of the open source OLAP server Mondrian². Finally, results from experimental evaluations using this implementation are reported in Section VI and Section VII concludes the paper.

II. RELATED WORK

The problem of selecting a set of aggregation tables to speed up query processing in data warehouse systems has first been discussed in [9] and has been known as the *view selection problem* ever since. Apart from the greedy algorithm described in [9], genetic algorithms [20] and randomized algorithms [13] have been proposed to solve the problem. However, view selection also plays an important role in other applications such as data placement and replication over wide area networks. In this broader context, [7] formalizes the view selection problem using conjunctive queries and provides a theoretical analysis of the optimal solution's cardinality.

[9] introduces a lattice framework that expresses the computational dependencies of views – each of its nodes corresponds to an aggregation table. The lattice considers two kinds of dependencies: the dependencies within a dimension caused by attribute hierarchies and those originating from the interaction of different dimensions with each other. Having augmented the nodes in the lattice with the additional information of estimated space consumption, a greedy algorithm is proposed to solve the optimization problem with the two goals of (i) minimizing the average time necessary to evaluate the set of queries that are identical to the views and (ii) dealing with only a limited amount of space to store the aggregation tables. The key concept is to maximize the overall benefit and successively choose those candidate views for materialization that provide the highest increase in the overall benefit. The benefit of materializing a view v is defined as the sum over the benefits of all dependent views in the lattice including itself. The benefit for each of these dependent views is defined as the difference of the costs to compute the views with and without the materialization of v . It has been shown that this greedy algorithm performs well [12] even though the problem of view selection is known to be NP-complete [9]. However,

²<http://mondrian.pentaho.org>

although this approach considers the dependencies between views, it does not pay any attention to the current workload but assumes that the access frequencies of all views in the lattice are the same. Thus, there could actually exist a better set of aggregation tables with respect to the current workload.

This in turn is the strength of the approach published in [17], which is closely related to the work we present in this paper. WATCHMAN [17] deals with caching a set of query results that might be beneficial in the future. It also uses a greedy algorithm and consists of two parts: a strategy for cache replacement and a strategy for cache admission. The key concept is to consider more aspects than only the past reference pattern of a query to approximate the future pattern. WATCHMAN uses a profit metric to measure the benefit of caching the results of a particular query. In addition to the reference pattern, this metric takes query execution costs and result size into account. The profit is used for both parts of WATCHMAN: those cached query results that provide the least profit are candidates for eviction and a new result set is added to the cache only if it improves the overall profit. However, this approach considers queries in separate and does not pay attention to the relationships between them – a query might still benefit from cached query results even though the queries are not the same. Consider for instance a query Q_2 originating from a roll-up operation on the result of query Q_1 . In that case the result of Q_2 can be derived from the cached results of Q_1 and thus benefit from it.

The major commercial DBMS also provide support for the view selection problem as part of so-called design advisors. The first incarnations of these tools were limited to index selection [6] but current versions also take aggregation tables as well as their interdependencies with other physical design features (partitioning, clustering, indexes) based on an analysis of the given workload into account and exploit multiquery optimization [1], [21]. Some of these tools work also at the OLAP level, e.g., the IBM DB2 Cube View Optimization Advisor. Even Mondrian offers an Aggregation Designer, but this tool is not workload-aware. All these approaches are still more or less static design tools requiring intervention by the DBA. Thus, several online approaches have recently been proposed for the related problem of index selection [5], [14], [16] as well as vertical partitioning [15]. In [18] an approach for materialized view selection is described that models the problem as the shortest path problem in a DAG and proposes a greedy solution. However, the authors focus on the optimization part and do not address the problem of obtaining the needed cost information.

III. AGGREGATION LATTICE AND COST MODEL

In the following we assume that the OLAP server supports MDX queries. An MDX query consists of axis specifications describing the dimensions and their projections onto axes, a cube specification, and a slicer to filter the data. Facts (measures) are treated as special dimensions. Conceptually, each MDX query codirectly corresponds to an aggregation table or an *aggregation node*, respectively. An aggregation

node is described by a set of dimension levels and measures specifying at which level in each dimension the measure is to be aggregated.

Example 3.1: Assume we have two measures and three dimensions derived from the TPC-H³ schema and their hierarchies (also denoted as dimension/aggregation levels):

- 1) *measures* (price - quantity).
- 2) *time* (day - month - year),
- 3) *order* (date - customer - nation - region), and
- 4) *part* (name - brand - manufacturer - type - supplier - nation - region).

Based on these dimensions and levels, an MDX query asking for the quantity of sold items, e.g., for all part types ordered in the USA in May 2000, would be defined as follows:

```
SELECT {[Measures].[Quantity]} ON AXIS(0),
[Part].[Region].[Nation].[Supplier].[Type] ON AXIS(1),
[Order].[Region].[USA] ON AXIS(2),
[Time].[2000].[May] ON AXIS(3)
FROM [TPCH];
```

Thus, for each MDX query the corresponding aggregation node can be derived by processing the following steps:

- 1) extract the dimension levels,
- 2) generalize the dimension levels, and
- 3) merge dimension levels.

This means (1) all referenced dimensions and dimension levels are extracted from the query, (2) strings referring to specific attribute values are removed and replaced by references to the corresponding dimension levels, and (3) if the query refers to multiple levels of the same dimension, we merge them so that we only keep the lowest level. Based on this, for the query of Example 3.1 we derive the aggregation node corresponding to the following levels:

- Time.[Year].[Month]
- Order.[Region].[Nation]
- Part.[Region].[Nation].[Supplier].[Type]
- Measures.[Quantity]

Note that all queries with the same set of dimension levels share a common node.

In order to check derivability of queries (aggregation tables) from other tables, an aggregation lattice is constructed. Such a lattice is a directed acyclic graph whose nodes represent aggregation tables and whose edges represent derivability relationships between nodes. An edge from v_i to v_j represents the relationship $v_i \preceq v_j$ meaning that v_i can be computed based on data from v_j without accessing the base tables.

For each pair of a query q_i (which is a synonym for v_i) and an aggregation table v_j supporting q_i (i.e., $q_i \preceq v_j$) we need the cost value c_{ij} that represents the costs for answering q_i based on v_j . For each node v_i we collect the cost c_{ii} for processing query q_i on v_i (which is in fact only a table scan on v_i) as well as a query counter cnt_i . These cost values are collected in a cost matrix where cell (i, j) represents the costs c_{ij} for processing query q_i on the aggregation table M_j .

³<http://www.tpc.org/tpch/>

The challenges are now (1) to maintain the lattice and (2) to collect necessary cost information for the cost matrix efficiently. In most cases, constructing the complete lattice in advance is not a feasible solution. The number of nodes for a schema with d dimensions, m measures, and l_i levels in dimension i are:

$$\#nodes = m \cdot \prod_{i=1}^d l_i$$

Therefore, we add only relevant nodes to the lattice on-demand. A node v_i is relevant if

- 1) a corresponding query q_i has already been issued or
- 2) node v_i is a common base node for two nodes of the lattice, i.e., $\exists v_j, v_k$ with $v_j \preceq v_i \wedge v_k \preceq v_i$.

Thus, for each issued query we check if the lattice already contains a corresponding node. If not, we need to insert a new node into the lattice whose position is identified by taking the derivability relationship into account. Instead of checking derivability for all nodes, we use an efficient coordinate-based approach, where coordinates are assigned to each node in the lattice. For this purpose, dimension levels are represented by numbers, e.g., day-month-year by 1-2-3. This is illustrated in Fig. 1 showing a grey highlighted node for the aggregation by month (time) and nation (order) with the coordinates (2,3). A query that can be computed on this node can also be computed on all nodes that provide more detailed information (dashed lines), e.g., nodes (1,2) and (1,1) which provide more details on both dimensions (time and order). Furthermore, all queries referring to less detailed information, e.g., nodes (1,4), (2,4), and (3,3) can be answered using the aggregation table corresponding to node (2,3). Of course, future queries referring to node (3,4) can also be derived from (2,3). For sake of clarity, Fig. 1 focuses on direct derivability relationships of nodes within the lattice (solid lines) and shows indirect derivability relationships only for node (2,3). Indirect derivability in this context means that within the lattice another directed path between two nodes exists that involves others as intermediate nodes.

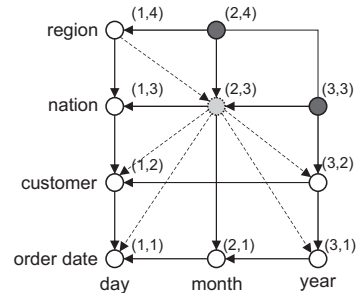


Fig. 1. Aggregation Lattice with Coordinates – representing a subcube referring to the two dimensions time and order, direct derivability relationships are represented by solid lines and indirect relationships by dashed lines.

Formally, we say that a node v_2 is derivable from a node v_1 if

$$\forall i \in D : v_1[i] \leq v_2[i],$$

where D is the set of dimensions and $v[i]$ denotes the coordinate of node v in dimension i .

A relevant new node v (relevant either because it is a common base node or because it represents a node that no previous queries referred to) with the coordinates (i, j) is inserted into the lattice at position (i, j) such that

- 1) v has an edge from all existing nodes derivable from v ,
- 2) v has an edge to all existing nodes it can be derived from, and
- 3) there is no edge between nodes that cannot be derived from each other.

As we construct the lattice on demand, it is possible that after the insertion of a node corresponding to an issued query, two nodes exist in the lattice but not their common base node for which no query has been processed so far and from which the first two nodes are derivable. In order to avoid this, we again use the coordinates: we can determine the greatest node from which both nodes can be derived by choosing the minimal coordinates so that the following equation identifies the coordinates of the greatest common base node v of two nodes a and b .

$$\forall i \in D : v[i] = \min\{a[i], b[i]\}$$

Assume, for example, we have already processed queries referring to nodes (2,4) and (3,3) but no query referring to node (2,3) in the lattice of Fig. 1. Then, after having computed queries corresponding to nodes (2,4) and (3,3), node (2,3) is inserted into the lattice as it represents the greatest common base node and should therefore be considered by the view selection algorithm.

Cost Model

Each time a new node v_i is added to the lattice, the costs values c_{ii} and c_{ij} (measured in #pages) for all nodes v_j with $v_j \preceq v_i$ have to be determined. Because materializing the node v_i and querying the catalog for the table size or asking the query optimizer for cost estimations is not an option, we use the following heuristics:

- 1) c_{ii} represents the costs of a simple table scan on v_i . Assuming we know the cardinality $|Q_i|$ of v_i , then c_{ii} is estimated by:

$$c_{ii} = \frac{|Q_i| \cdot \text{tuplesize} \cdot \text{fillfactor}}{\text{pagesize}}$$

$|Q_i|$ is estimated either by heuristic 3 (see below) or collected from execution as the result from a processed query – note that the execution of query q_i has triggered the insertion of v_i .

- 2) To estimate c_{ij} ($i \neq j$), we observe that all queries are compiled into similar execution plans. A typical example of such a plan generated by PostgreSQL is:

```
GroupAggregate (cost=...)
-> Sort (cost=...)
    SortKey: o_orderdate, s_name, ...
    -> Seq Scan on table
```

Other DBMS produce similar plans. In this plan the total costs are dominated by the sort and the scan and therefore mainly depend on the cardinality of the input relation,

which is v_i (or q_i). Thus, we can use the DBMS's internal cost function to get the same estimations as the query optimizer.

- 3) The critical step is estimating the cardinality of aggregation tables. Cardinality estimation is a general problem in cost-based query optimization, especially if many joins are involved as in our case. However, queries representing aggregation tables belong to the special class of grouping/aggregation queries. Therefore, we can restrict ourselves to the case of estimating the number of distinct value combinations of the grouping attributes. For this purpose, we have adopted the COLSCARD approach presented in [19]. This technique requires only frequency vectors f_1, f_2, \dots, f_k (exact or approximated by histograms) for all involved attributes A_1, \dots, A_k and derives an estimation E based on the Kronecker product of all frequency vectors: $f_v = f_1 \times f_2 \times \dots \times f_k$ by:

$$E = M - \sum_{v \in V} (1 - f_v)^N,$$

where V denotes the set of all value combinations, N the cardinality of the table, and M the number of possible value combinations.

These steps of analyzing an MDX query, identifying the corresponding aggregation nodes, and collecting or updating relevant cost values and counting information are performed for each incoming query.

IV. FINDING THE OPTIMAL CONFIGURATION

As stated in Section I, the objective of our approach is to maintain an optimal configuration, which we understand as the set of aggregation nodes that minimizes the costs for executing a given workload under given size constraints. For this purpose, we exploit the information collected in the aggregation lattice and the cost matrix. In order to achieve this goal, we can choose between several options:

- (a) **static:** If the workload is fixed or known in advance, we can determine the optimal configuration once.
- (b) **alert:** Under the assumption of changes or drifts in the workload, we continuously or periodically check if the current configuration is still optimal.
- (c) **online:** We try to adjust the configuration to the current workload continuously.

For variants (b) and (c) we therefore assume that the workload changes (slightly) over time and that the future behavior is not known in advance – otherwise, a static solution like [1] or a sequence-based approach [2] could be chosen. Moreover, we consider (b) as a special case of (c) because if there is a need to adjust the current configuration, it means that the current one is not optimal anymore.

This class of problems can be solved by online algorithms which process their input piece-by-piece without knowing the entire input (workload). Such algorithms are often illustrated by the ski rental problem or the ice cream vendor problem [8]. For our purpose, the specific class of task systems can be applied, which can be described as follows: Let $W =$

$\langle q_1, \dots, q_n \rangle$ be a sequence or workload of queries, M_i a configuration as a set of materialized aggregation nodes, and $\mathcal{S} = \langle M_1, \dots, M_k \rangle$ a configuration schedule where q_i is processed on M_i . Given an initial configuration M_0 , the goal is to find an optimal schedule \mathcal{S}^* that minimizes the costs, i.e.,

$$\mathcal{S}^* = \arg \min_{\mathcal{S}} \text{cost}(W, \mathcal{S})$$

where $\text{cost}(W, \mathcal{S})$ comprises the execution costs for W in \mathcal{S} as well as the transition costs between two configurations:

$$\text{cost}(W, \mathcal{S}) = \sum_{i=1}^n \text{cost}(q_i) + \text{transition}(M_{i-1}, M_i)$$

Here, the transition $M_{i-1} \rightarrow M_i$ basically means to materialize all aggregation nodes that are not already contained in M_i , i.e., $M_i \setminus M_{i-1}$, and to drop the nodes that are not needed anymore, i.e., $M_{i-1} \setminus M_i$. In the literature, several online algorithms have been proposed to solve this problem and their competitiveness has been proven e.g., for any metrical task system with n states (configurations) there is a $(2n - 1)$ competitive deterministic online algorithm [4]. However, these competitiveness properties of algorithms for task systems are valid only under special assumptions such as symmetrical transition costs forming a metric as in metrical task systems and for a limited number of states. Therefore, instead of randomized algorithms as discussed for example in [11], online approaches for database tuning tasks rely more or less on heuristics.

In our approach, we have also chosen a heuristic-based solution, which was inspired by the strategy proposed in [15]. This online strategy is sketched in Algorithm 1, which is executed for each incoming query. Each query q_i can trigger a transition to a new configuration M_i . We decide about a transition based on the cumulative penalty $\text{penalty}(M_{i-1})$ of remaining in the old configuration M_{i-1} instead of transitioning to M_i . The penalty Δ_i for q_i of remaining in M_{i-1} is determined by

$$\Delta_i = \text{cost}(q_i, M_{i-1}) - \text{cost}(q_i, M_i)$$

and aggregated to the cumulative penalty:

$$\text{penalty}(M_{i-1}) = \text{penalty}(M_{i-1}) + \Delta_i$$

M_i is the configuration with the maximum benefit for q_i , i.e., the configuration minimizing its computation. Obviously, this configuration can be constructed by adding the direct node v_i for answering q_i to M_{i-1} . However, M_i is not necessarily the optimal configuration for the current workload. Therefore, we use the penalty only as an indicator for a necessary transition. If a transition is necessary, we perform a more comprehensive search: if the penalty exceeds a given threshold thr , we look for the configuration M_j that minimizes the query costs regarding the previous k queries $q_i, q_{i-1}, \dots, q_{i-k+1}$ (function *best-neighbor-config* in Algorithm 1). Considering k queries instead of only the current one gives enough benefit to exceed the transition threshold and helps to find a configuration that does not favor the current query. For this purpose, we construct a sub-lattice consisting of all nodes $v_i, v_{i-1}, \dots, v_{i-k+1}$ as well as their common base nodes (Sect. III) and perform an exhaustive or greedy search (depending on the number of

nodes) on all possible configurations starting from the current configuration M_{i-1} by taking the values c_{ij} from the cost matrix as well as the transition costs into account. Note that the number of possible configurations is low for small k .

If such an M_j exists, we materialize this configuration, reset the penalty, and adjust the threshold thr . Its initial value is determined as follows: we simply assume an initial configuration M_0 (which could be constructed by materializing the nodes corresponding to the first issued queries as long as space is available to materialize further aggregation nodes) and compute the threshold as

$$\text{thr} = \text{transition}(M_0, M_0)$$

where M_0 is the empty configuration. As transition costs $\text{transition}(M_{i-1}, M_i)$ in all steps we use the materialization costs of new aggregation nodes in M_i that are not contained in M_{i-1} .

Input: incoming query q_j
current configuration M_{i-1}

- 1 $M_i = M_{i-1} \cup \{v_j\}$;
- 2 $\Delta_i = \text{cost}(q_j, M_{i-1}) - \text{cost}(q_j, M_i)$;
- 3 $\text{penalty}(M_{i-1}) = \text{penalty}(M_{i-1}) + \Delta_i$;
- 4 **if** $\text{penalty}(M_{i-1}) > \text{thr}$ **then**
- 5 $M_j = \text{best-neighbor-config}(M_{i-1})$;
- 6 $\text{materialize}(M_j)$;
- 7 $\text{penalty}(M_j) = 0$;
- 8 $\text{thr} = \text{transition}(M_{i-1}, M_j)$

Algorithm 1: Online Algorithm

Space constraints for aggregation tables are considered during the search for the best configuration (*best-neighbor-config*). Here, a fixed amount of disk space for aggregation tables is defined and new nodes are added to the configuration as long as the available space is not exceeded. If an already materialized node has to be removed, the penalty of dropping the table is added to the transition costs. This penalty can be again obtained by aggregating the costs c_{ij} regarding the last k queries.

A more flexible solution is described in [10] where we apply a principle from economics: Gossen's First Law or the concept of diminishing marginal utility.

Another important issue in any workload-aware tuning approach is the consideration of update costs. One possible solution is to treat updates as queries with negative benefits, a second strategy is to treat the workload as a sequence and try to find the optimal ordering of create/drop operations. However, using an online strategy without knowing the entire workload in advance, the negative benefits are cumulated over time until the expensive aggregation tables are dropped one after another. Obviously, this is not appropriate for the bulk loading step in an OLAP/data warehousing scenario. In this case, either the aggregation tables should be dropped before updating the base tables and recreated on-demand by the online optimizer or all previously selected aggregation tables should be refreshed automatically after the loading step.

V. ARCHITECTURE AND IMPLEMENTATION

The main components of the system architecture are illustrated in Figure 2. As we are using Mondrian as OLAP server, the query received as input to our system is not an SQL query but an MDX query, which has to be parsed and executed. As the system uses PostgreSQL as relational backend, the MDX query has to be rewritten into SQL referring to relations. Then, the resulting SQL statement is evaluated and the result can be output to the user. As PostgreSQL does not support aggregation tables natively, the aggregate manager and the schema/dimensional manager provide all information necessary to rewrite an MDX query in consideration of all aggregation tables that might already exist.

In addition to the components concerning query execution, our system also contains components for monitoring the current query load: the aggregate monitor. Whenever a query is executed, the monitor adds corresponding nodes to the aggregation lattice, as we have already discussed in Section III. It also monitors the number of times the query has been issued in the current query load. Furthermore, it maintains the cost matrix, which is a crucial component to determine a configuration's benefit.

The online optimizer implements the strategy described in Section IV. If an aggregation node is chosen for materialization, the SQL generator is invoked to create and populate the table in the backend and to register it in Mondrian's aggregate manager.

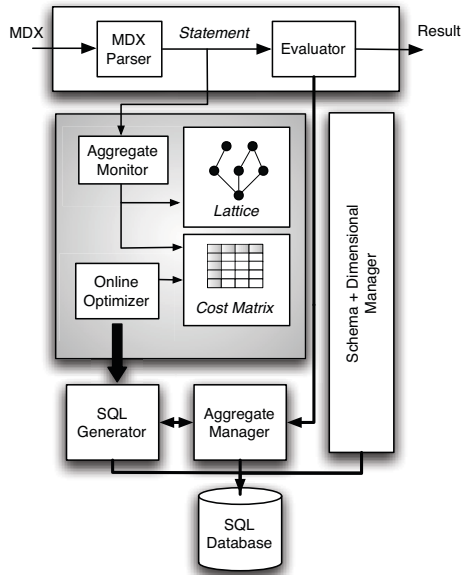


Fig. 2. System Architecture

VI. EVALUATION

In our experimental evaluation we were interested in the following important questions: (1) because the cost estimations are essential for finding the optimal configuration – how is the quality of the heuristics for obtaining correct estimations? (2)

How does the online approach perform compared to alternative approaches?

For our evaluation we used the TPC-H data set (SF = 1 GB) as well as the Mondrian server together with PostgreSQL 8.3 as backend on a Linux Debian box.

In the first set of experiments we have evaluated the heuristics introduced in Sect. III. For this purpose we compared the actual execution costs of a set of 1000 randomly chosen queries/nodes from the lattice with the estimated costs of our heuristics. Due to space constraints we omit the results for heuristic 1 – the c_{ii} costs – which are too simple, and the results for heuristic 3 because they have been presented by the original authors in [19]. However, we should note that in our experiments we obtained results that are worse than presented in [19], particularly for skewed and correlated data. Thus, in future work we will investigate other techniques (e.g. sampling) for estimating the number of distinct value combinations.

In Fig. 3 the results for the c_{ij} costs (heuristic 2) are shown. The diagram shows the frequency of the relative estimation errors in percent. Most of the estimations are in the range of 10% which shows that the rather simple heuristics already produce good estimations without stressing the query optimizer of the underlying DBMS.

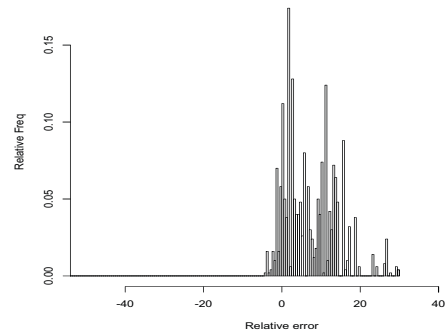


Fig. 3. Relative Estimation Error

In the second set of experiments, we evaluated the performance of our approach (online) in comparison to several alternative approaches:

- **no optimization** – there are no materialized views at all,
- **greedy** – a greedy approach [10] selects the set of materialized views that promises the least costs for all queries issued so far, and
- **exhaustive** – all possible configurations with respect to the nodes in the lattice are considered. The one promising the least costs for all queries issued so far is chosen.

After a query has been processed, the exhaustive optimizer tries to find the best configuration with respect to the query workload, which consists of all queries processed so far. If a better configuration than the one currently in use is found, the configuration is adjusted. Because of the huge overhead in execution costs, the exhaustive optimizer is started only periodically (after 5, 10, 15, ... queries have been issued). In

contrast, the online optimizer and the greedy optimizer might adapt the configuration after each issued query. The online optimizer used a cache size of four, which means the four most recently issued queries were considered for optimization once the threshold was exceeded. Each configuration was allowed to occupy 1.46 GB – the space required to materialize four times the table corresponding to the join of all base tables, i.e., the aggregation node with (0,0,0) as coordinates.

Fig. 4 and Fig. 5 show our results using the TPC-H data set, the quantity measure, and all aggregation levels introduced in Example 3.1. We used four different setups, each using a workload of 50 queries:

- **random:** queries were referring randomly to any possible aggregation table,
- **repetition:** 10 queries referring randomly to any possible aggregation table were chosen, each issued query was chosen randomly from this set,
- **random shift:** two sets of queries (each consisting of 10 different queries referring randomly to any possible aggregation table) were generated, the first 25 queries were chosen randomly from the first set, the remaining 25 queries were chosen randomly from the second set,
- **dimensional shift:** two sets of queries (each consisting of 10 different queries) were generated, each set referred only to a disjoint subset of dimensional levels, the first 25 issued queries were chosen randomly from the first set, the remaining 25 issued queries were chosen randomly from the second set.

We measured the cumulative execution costs of the query load, i.e., the costs for issuing a query with respect to the current configuration were accumulated. Furthermore, whenever a configuration was adjusted, we measured the transition costs – again in a cumulative manner.

Note that there is no guarantee that the online or the greedy approach find the optimal configuration – in contrast to the exhaustive optimizer. Fig. 4(a) shows our results with respect to the randomly chosen set of queries. This is the worst case scenario for any optimization algorithm because although a set of recently issued queries is known, it is impossible to determine an optimal configuration for future queries as there is no correlation between them. Thus, our results with respect to query execution costs in the random scenario (Fig. 4(a)) illustrate that all optimizers produce configurations that are only little better than using no materialized views at all. All approaches often change their configurations so that transition costs are high (Fig. 5(a)). The online approach changes its configuration the most because it only considers the last four issued queries to find an optimal configuration. Thus, it produces the highest transition costs.

Fig. 4(b) and 5(b) show our results with respect to the repetition scenario. For this scenario, all optimizers result in similar query execution costs. The exhaustive optimizer performs a little worse because it was only allowed to adapt the configuration periodically so that optimization could not take place for the queries issued in between. For this test series the online approach results in relatively low cumulative transition

costs. The reason is that with a workload consisting of only 10 different queries, optimizing the configuration with respect to four of them results in almost ideal configurations so that it takes some time until the transition threshold is exceeded. The greedy optimizer continuously adapts the configuration so that it causes the highest transition costs.

For both shift scenarios we obtained similar results (Fig. 4(c), 4(d), 5(c), and 5(d)). Due to the periodic optimization, the exhaustive optimizer cannot clearly outperform the other two optimizers. For both experiments we have a sudden increase of execution costs around the 25th issued query because the first 25 queries are chosen from a different query pool than the remaining 25 queries.

VII. CONCLUSION

In this paper, we have investigated the well-studied problem of view selection as an online tuning task. In contrast to the approach chosen by the major DBMS vendors, which is based on special tables (materialized views) and query rewriting strategies, our work does not require special DBMS extensions and therefore, seamlessly integrates with other implementations of aggregation tables outside the SQL backend, e.g., implementations exploiting main memory and compression techniques. Another contribution of our work are heuristics for obtaining the necessary cost information without expensive optimizer calls.

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] S. Agrawal, E. Chu, and V. R. Narasayya. Automatic Physical Design Tuning: Workload as a Sequence. In *SIGMOD 2006*, pages 683–694, 2006.
- [3] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR 2005*, pages 225–237, 2005.
- [4] A. Borodin, N. Linial, and M. Saks. An Optimal Online Algorithm for Metrical Task System. *Journal of the ACM*, 39(4):745–763, 1992.
- [5] N. Bruno and S. Chaudhuri. An Online Approach to Physical Design Tuning. In *ICDE '07*, pages 826–835, 2007.
- [6] S. Chaudhuri and V. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB '97*, pages 146–155, 1997.
- [7] R. Chirkova, A. Halevy, and D. Suciu. A Formal Perspective on the View Selection Problem. In *VLDB '01*, pages 59–68, 2001.
- [8] M. Chrobak and L. Larmore. Metrical Task Systems, the Server Problem, and the Work Function Algorithm. In A. Fiat and G. Woeginger, editors, *Online Algorithms: The State of the Art*, pages 74–94. Springer, 1998.
- [9] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing Data Cubes Efficiently. *SIGMOD '96*, pages 205–216, 1996.
- [10] K. Hose, D. Klan, M. Marx, and K. Sattler. When is it Time to Rethink the Aggregate Configuration of Your OLAP Server. In *VLDB 08 Demonstration*, 2008.
- [11] S. Irani and S. Seiden. Randomized Algorithms for Metrical Task Systems. *Theoretical Computer Science*, 194(1-2):163–182, 1998.
- [12] H. Karloff and M. Mihail. On the complexity of the view-selection problem. In *PODS '99*, pages 167–173, 1999.
- [13] 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.
- [14] M. Lühring, K. Sattler, K. Schmidt, and E. Schallehn. Autonomous Tuning with Soft Indexes. In *SMDB '07*, pages 450–458, 2007.
- [15] T. Malik, X. Wang, R. Burns, D. Dash, and A. Ailamaki. Automated Physical Design in Database Caches. In *SMDB '08*, pages 27–34, 2008.

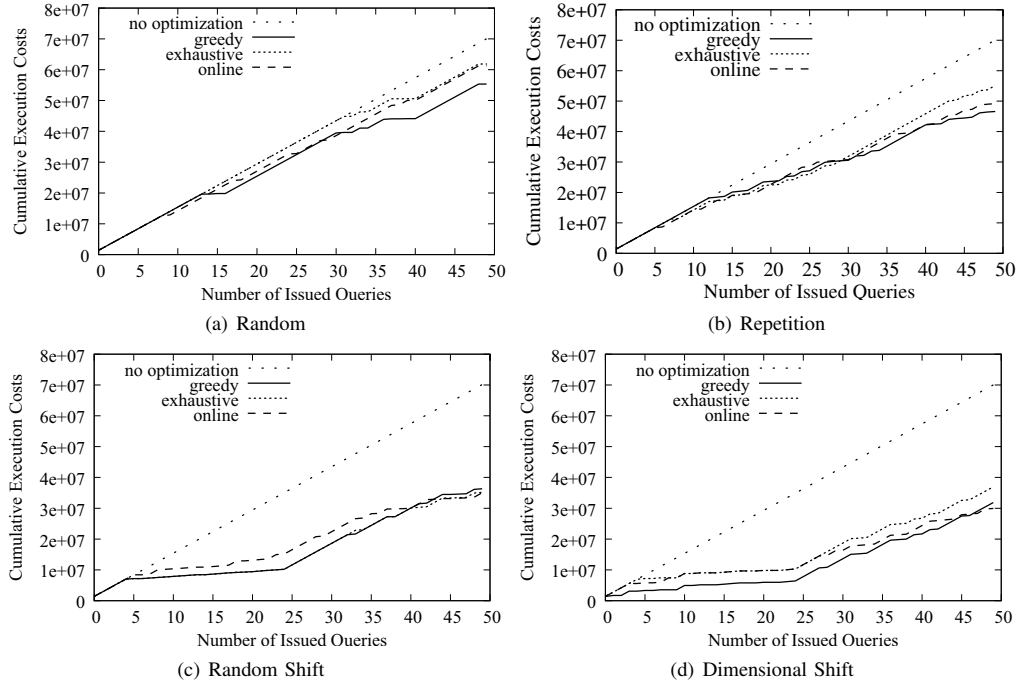


Fig. 4. Evaluation Results – Execution Costs

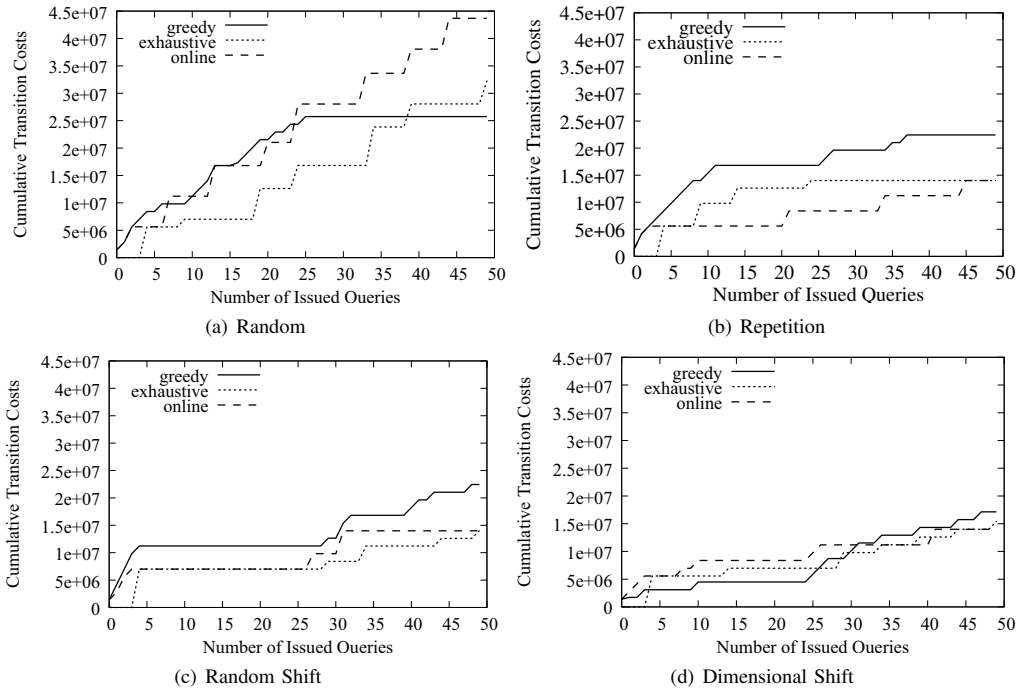


Fig. 5. Evaluation Results – Transition Costs

- [16] K. Sattler, I. Geist, and E. Schallehn. QUIET: Continuous Query-driven Index Tuning. In *VLDB '03*, pages 1129–1132, 2003.
- [17] P. Scheuermann, J. Shim, and R. Vingralek. WATCHMAN : A Data Warehouse Intelligent Cache Manager. In *VLDB '96*, pages 51–62, 1996.
- [18] W. Xu, D. Theodoratos, C. Zuzarte, X. Wu, and V. Oria. A Dynamic View Materialization Scheme for Sequences of Query and Update Statements. In *DaWaK 2007, LNCS 4654*, pages 55–65, 2007.
- [19] X. Yu, C. Zuzarte, and K. Sevcik. Towards Estimating the Number of Distinct Value Combinations for a Set of Attributes. In *CIKM'05*, pages 656–663, 2005.
- [20] C. Zhang and J. Yang. Genetic Algorithm for Materialized View Selection in Data Warehouse Environments. In *DaWaK '99*, pages 116–125, 1999.
- [21] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB '04*, pages 1087–1097, 2004.