Dynamic Construction and Administration of the Workload Graph for Materialized Views Selection

Andreas Lübcke, Ingolf Geist, Ronny Bubke
Otto-von-Guericke-University Magdeburg
School of Computer Science
Institute for Technical and Business Information Systems
D-39016 Magdeburg, Germany
P.O. Box 4120
{andreas.luebcke, ingolf.geist}@ovgu.de

To offer the best performance to each application (e.g., decision support systems) administration and optimization of database management systems is needed. In order to reduce administration costs and to provide continuous adaptation to changing workload patterns, self-management techniques have been in the focus of researchers and DBMS vendors for recent years. One important topic in this area is the selection of materialized views, i.e., complex queries of OLAP systems benefit from pre-computed results. In this work, we present an automatic selection of materialized views using statistics of a workload which is represented in the query graph model. We propose merging of queries of a workload into a workload graph and the computation of merged views. Hence, we introduce a cost model to select the views to be materialized. The proposed techniques are evaluated using the TPC-H benchmark and compared to an industrial design advisor. Furthermore, our approach can be used in static as well as in on-line self-tuning.

Decision support systems and OLAP (On-Line Analytical Processing) queries put high performance demands on the underlying database management systems because of huge data sets and the typically complex queries. Besides optimizing physical design and query processing, materialized views (MV) provide large performance advantages by storing and using pre-computed partial results of complex query operations. But, MVs also create costs in two ways: `(i)` increased storage space because of storing of redundant pre-computed results, and `(ii)` increased update costs because modifications of the base tables trigger re-computations of the MVs. That leads to the problem of the selection of the appropriate set of MVs for a given workload and space constraints. Complex and changing workloads in an OLAP scenario require the automation of MV selection, which is one problem of the self-tuning techniques developed by database system vendors (Chaudhuri & Narasayya, 2007).

We contribute a new approach for automatic and dynamic selection of MVs for changing workloads. In contrast to prior work, we do not consider individual queries, but entire workloads consisting of multiple queries. We therefore extend the Query Graph Model (QGM) and use it in a new way. In order to support online statistic collection with low overhead, we present fast workload graph modifications and restructuring operations. The operations ensure a sufficient and compact representation of the workload. The statistics collected online in the workload graph are used in a cost model for selecting MVs using a greedy algorithm. Using these techniques, the approach is able to select a near optimal set of MVs and to adapt to changing workloads.

Related Work

Self-tuning techniques has been in the focus of database community for recent years. Chaudhuri and Narasayya (2007) show the progress of the last ten years as well as the need of further tasks.

The WATCHMAN approach (Scheuermann et al., 1996) considers intelligent caching strategies for decision support systems. Therefore, Scheuermann et al. (1996) do not restrict to MV or indexes rather they aim to a more general solution. We adopt these caching strategies to administrate MVs for a given workload.

Zilio et al. (2004) illustrate the necessity of joint consideration of MV and indexes. They show the capabilities of an online advisor using this joint consideration. We assume a separate estimation of indexes if the indexes do not belong to MVs. In our work, we propose a stepwise estimation because our approach is designed as MV recommendation which should estimate only indexes for existing MVs.

Agrawal et al. (2000) describe the need of merging MVs for online recommendation and administration of MVs. Agrawal et al. (2000) propose an algorithm based on merging of complete MV. This approach aims at a general solution for MV selection in a commercial DBMS. Hence, the merging of MV is bound to several conditions, e.g., the MV for merging need to have the same join predicate. These conditions restrict the optimization of administration and (cache) size constraints. Our approach aims to a finer granularity for merging because our algorithm has fewer restrictions. We assume a benefit by merging parts of MVs.

Background

The QGM was presented by Zaharioudakis et al. (2000)[1]. It represents a single query. For further considerations, an instance of the QGM is called QGM graph (QG).

The QG is a directed acyclic graph with a root node ($QG = (V, E)$). The data flow is represented by edges (E (QG)) from child nodes to the parent node. The nodes (V (QG)) are called `boxes`. A box is defined by its type, its predicates, and the input- and output attributes. The QGM comprises two types of boxes, SELECT and GROUP BY boxes. The output attributes and predicates are computed based on the input attributes. A parent boxes adopts the output attributes from its children. SELECT boxes represent selections, projections, joins, and all scalar expressions. GROUP BY boxes contain all grouping and aggregation functions. The GROUP BY boxes can have any number of parent boxes, but only one child box is allowed.

Now we illustrate an example SQL query decomposed in the QGM.

```
SELECT  faid, state, year(date) as year,
count(*) as cnt
FROM    Trans, Loc
WHERE   flid = lid
AND     country = 'USA'
GROUP BY faid, state, year(date)
HAVING  count(*)>10;
```

Listing 1 Query Q1 - Selection of country drawn transaction data.

Figure 1 shows the QGM representation of Q1 (the query graph) in **Listing 1**. The bottom SELECT box implements the join between the tables. The output attributes of the bottom box conform to the projection of query Q1. The parent box is a GROUP BY box that represents the grouping clause. Accordingly, the parent box (the root) retains the output attributes and enforces the selection of the HAVING clause `cnt>10`. Finally, the output columns represent the SELECT clause of query Q1.

In most cases, we do not have exact matches between subgraphs. Zaharioudakis et al. (2000) derive matching patterns according to non-exact matches for QGM. In general, the following properties have to be fulfilled. A QGM graph `G(E,R)` can be constructed that `G(E,R)` contains the sub-graph `G(R)` (root is R), and `G(E,R)` is (semantically) equivalent to the sub-graph `G(E)` (root is E). `G(E)` and `G(E,R)` always have the same result. The `composition` is a set of operations performed on `G(R)` to obtain the same result as `G(E)` (see **Figure 2**).

Dynamic Creation and Administration of Materialized Views

In this section, we present our approach. We use the previous work on the QGM (Zaharioudakis et al., 2000), but consider not individual queries but entire workloads. First,

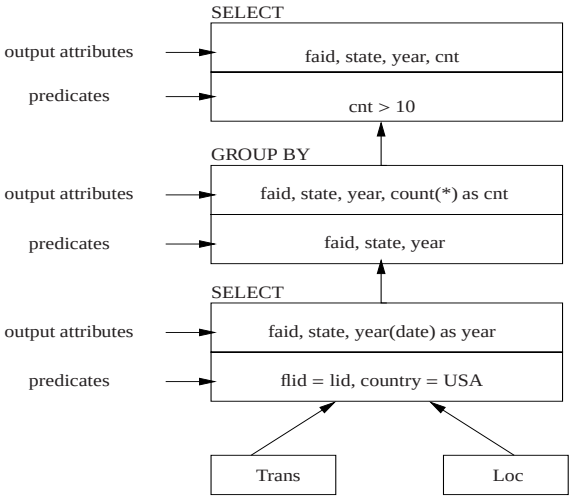

Figure 1. Query graph of Query Q1

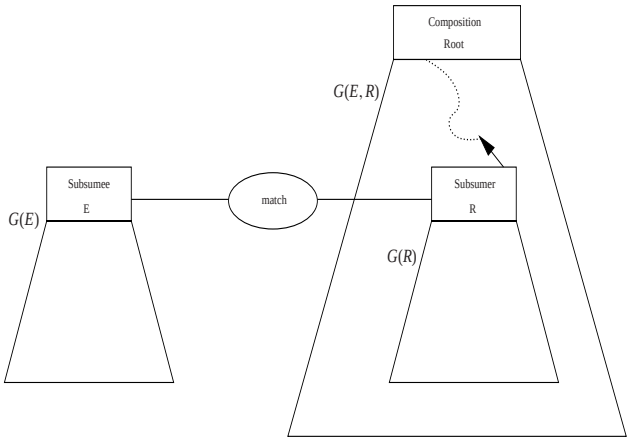

Figure 2. Matching between G(E) and G(R)

.

we describe the extension of the QGM to capture the relevant statistics which are required for MV selection. Second, we develop a profit function to select MV for materialisation deduced from the WATCHMAN approach (Scheuermann et al., 1996). Therefore, we use the statistics obtained by our extensions to the QGM. Finally, we describe our algorithms to reduce administration costs and discuss the costs of them.

Workload Representation

In our approach, the basic idea is to map the whole workload into one representation. So far, queries are represented corresponding QG, but in this way, we cannot calculate an optimal representation for a workload. In our approach, the workload is represented by merging different QGs to the `workload graph` (WG) that represents the complete workload. In the following sections, we describe the needed algorithms and extensions for our approach to represent the workload and its corresponding statistics.

[1] Note the differences to STARBURST (Pirahesh et al., 1992)

Gathering Statistiscs for MV Selection

We need to collect statistics about the workload and the base relations to calculate the benefit of a MV. The number of tuples of a box is called $TpCnt$. The feature tuple size $TpSz$ characterizes the sum of the attribute sizes. The operation process cost OPC of a box derives from its number of tuples and its child boxes. Furthermore, the reference counter $RefCnt$ indicates the access frequency. If a query matches to a box, the $RefCnt$ will be incremented.

Furthermore, we define common properties for all types of boxes. Every output attribute has two features: `size` and `cardinality`. The `size` describes the maximum size of the attribute values which is used for the tuple size calculation. The `cardinality` is the number of distinct values of an attribute.

Therefore, we introduce a third box type `TABLE` that represents relations of the system. This extension is necessary to describe statistics of base relations which are necessary for our profit calculation. A `TABLE box` has all attributes of the base table as output attributes and the only predicate is the base table name (no inputs are allowed).

Profit calculation for Selection of MVs

To calculate the benefit of MVs, we need a profit function. In this section, we introduce our profit function, and above all, the processing of MV selection is described according to the profit function.

We have to estimate the benefit of each box in the WG is estimated using a profit function. The profit function calculates a profit value for each box from its statistics. The parameters operation costs OPC and reference counter $RefCnt$ increase the profit. The tuple count $TpCnt$ and the tuple size $TpSz$ reduce the profit. The relation size of the box is needed because we have a limited space. We get for box b:

$$profit(b) = \frac{OPC(b)}{TpCnt(b) * TpSz(b)} * RefCnt(b) \quad (1)$$

Another important criterion for profit calculation is the update frequency of the base tables. Base tables with a higher update frequency should have a lower profit then other tables. The update behaviour $UpdBhr$ provides the costs for updating of the dependent base tables. A higher update frequency decreases the profit because of increasing number of MV updates. We obtain the alternative profit function:

$$profit_{update}(b) = \frac{OPC(b) * RefCnt(b)}{TpCnt(b) * TpSz(b) * UpdBhr(b)} \quad (2)$$

For now, we use the profit function (1) without the update behaviour for our approach. But, the function can be transparently replaced.

The MV selection algorithm uses our profit function and is executed periodically. The profit function processes all WG boxes except the TABLE boxes. The result is a list of box profit value pairs which is sorted in descending order (greedy selection) by the profit values. The box sizes will be successively downwards summarized and subtracted from cache size. If a box size is bigger than the remaining cache size it will not be subtracted from the cache size but will be deleted from the list. At the end, the list only consists of boxes which fit completely into the cache. Afterwards, all existing MVs will be deleted which are not in the update list and these MVs are materialized which are not in the cache.

The used algorithm works in a greedy manner which does not necessarily find the optimal solution of a problem. To find an optimal solution we have to permute and estimate all MVs. This problem is well known as a variant of the Knapsack Problem (Kellerer, Pferschy, & Pisinger, 2004).

Aging Function

A changing workload will not be optimal represented in the WG. Often occurring queries will more often referenced which leads to a higher profit in the profit calculation. In general, this is a good approach for static workloads but we will get a nearly static set of MVs for a changing workload. The system reacts very slowly on changing workloads, and there will be relics in the WG. The alternation of purposed MVs and MVs in the WG would be the worst case. Therefore, we need a suitable aging function (Scheuermann et al., 1996).

We define an upper limit of reference counter values to solve this problem. If any box of the WG reaches this limit we will divide the reference counter $ref(b)$ with $b \in WG$ of all graph boxes by an integer value F (normalization). Hence, we achieve:

$$ref_{new}(b) = \frac{ref_{cur}(b)}{F} \quad (3)$$

If the reference counter of a box b falls below a lower limit doing the normalization then the box b will be deleted, and the size of the WG is reduced. This is a capable solution for regular changing workloads with a suitable chosen reference limit and divisor. The limits can be computed automatically from heuristics or determined manually. Finally, we have to recalculate the profits after normalization of our reference counter.

Constructing the Workload Graph

So how is a WG created? We incrementally construct it from all queries. We start with the first query, then combine it in some form with the next, then continue with the third box etc.

Therefore, a matching algorithm is needed to compare a QG with the WG. To navigate in the WG, we can traverse the WG in any direction because the child boxes and the parent boxes reference each other. Hence, a QG and the WG will be compared (pair wise) in a bottom up way. So, the matching of QG and WG starts with TABLE boxes.

In the first steps, TABLE boxes of a QG are compared to TABLE boxes of the WG. Equally, the SELECT boxes and the GROUP BY boxes are compared in each case to each other. If there is an exact match then we do not insert a box. So, we go up to the parent nodes and iterate with it again.

If the matching algorithm does not find a match between a TABLE box of QG and a box in the WG, we have to insert the TABLE box of QG into the WG. Furthermore, the complete sub-graph of QG is inserted into the WG because the QG is disjoint from the WG until a join predicate is reached. A box which contains a join predicate is called `join box`. A join box can only be inserted if all child boxes are matched positively. Additionally, SELECT boxes and GROUP BY boxes of QG are inserted into the WG if there is no match between them. For example, `Box-Sel3` will be added to `Box-Table-WG` as a parent node (see **Figure 3**). We mind that a GROUP BY box can only have one child. If an insertion contradicts the condition for GROUP BY boxes, we have to add a complete sub-graph, too.

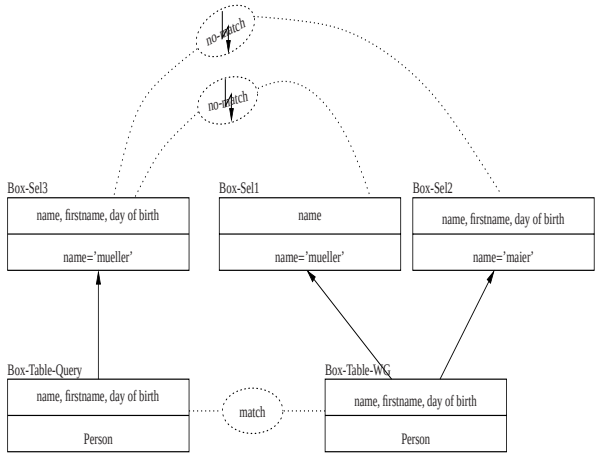

Figure 3. Insertion into the Workload Graph

We continue the matching until the root is reached or a join box is encountered. If a join predicate is found, matching stops and the next TABLE box of QG is chosen.

Ambiguities and irregular profit calculations in the WG can occur if different predicates will be inserted individually which appear jointly at a later point of time. This can be avoided by sorting the attribute names in lexicographic order before matching the boxes.

If a TABLE box (or child box) has to be deleted then their parent boxes will be deleted, too, because the bidirectional reference implicates the need of deletion in the next step. These boxes could not be reached because the WG will always be traversed starting with the TABLE box. Hence, we delete these boxes like boxes without parent nodes.

Altogether, our WG represents the workload by adding and deleting boxes. Due to the benefit of our approach, we have to minimize the maintenance costs. Therefore, we develop two algorithms (restructuring and merging) which are described in the following two sections.

Restructuring the Workload Graph. A restructuring algorithm is needed because we need to use the same sub-graph of the QG for as many queries as possible. Thereby, we minimize the overhead, and we materialize further parts of the workload according to a designated space limit. We have to check if parts of the WG meet the subset condition after insertion, i.e., the resulting WG (after restructuring) has no parent box which is a subset of another parent box starting from a certain box.

The restructuring algorithm for the WG starts from the TABLE boxes and ends at a leaf node. Afterwards, another TABLE box is chosen until each box visited (at most) once. Thereby, all parent boxes match with each other. In case of success, a help function attaches the parent box of the `subset box` to the matched box or the occurred composition. Afterwards, we have to recalculate the operation costs of the restructured boxes. The algorithm is listed in **Listing 2**.

Merging in the Workload Graph. To reduce the size of the WG and minimize redundancy, we propose the following merging algorithm, i.e., we minimize the maintenance overhead of the WG. In most cases, a workload does not comprise exact matches between the boxes of the WG. So, many boxes will be inserted into the WG although there are overlaps between boxes. We extend the approach (Zaharioudakis et al., 2000) by a merging functionality, i.e., we merge different boxes of the WG with common parts to one box represented in the WG. We use a `composition` to replace the corresponding parts semantically correct by a new box. We iterate this procedure in a bottom up way, i.e., we merge subgraphs of the WG, too, if they contain common parts. Therefore, the composition can contain any number of SELECT and GROUP BY boxes. If the composition has been empty, we find an exact match within the WG. To match the boxes, we use the presented matching algorithm using the match patterns derived by Zaharioudakis et al. (2000). To compute the composition correctly, we investigate different merge patterns for boxes in the WG. We distinguish four merge patterns: two SELECT boxes, two GROUP BY boxes, two boxes with interval predicates, and disjunctive predicates. For following considerations, `G(E)` is called `subsumee` and `G(R)` is called `subsumer`.

First, we have two SELECT boxes with one common predicate. The `merge box` becomes subsumer. The subsumer contains all common predicates of the subsumees (the two SELECT boxes) and all derivable predicates from the child boxes. The child boxes of the subsumees become child boxes of the subsumer (the merge box) if the predicates can be derived from its predicates. The subsumees will be the parent boxes of the subsumer. The new reference rate is computed by the sum of references of the merged boxes. Hence, the subsumees reference to their children and their common predicates will be deleted. The subsumees obtain the subsumer as child. This example is shown in **Figure 4** and **5**.

The second case considers merging of two GROUP BY boxes (subsumees) whereby their children are no GROUP BY boxes. The merge box (subsumer) obtains all predicates and attributes of the subsumees. Hence, the common child of the subsumees becomes child of the merge box (subsumer). The subsumees are the new parents of the subsumer. The reference rate is again the sum of references. Afterwards, the subsumees derive their predicates from the subsumer predicates according to the rules from Zaharioudakis et al. (2000).

```
function restructureMapGraph (childBox)
        boxPairs = getAllBoxPairs(childBox)
        foreach (pair in boxPairs)
                lBox = pair.left
                rBox = pair.right
                if (MatchFunction.match(lBox, rBox, null))
                        compensation = MatchFunction.getCompensation()
                        manageBoxRestructering (lBox, rBox, compensation)
                else
                        if (MatchFunction.match(rBox, lBox, null))
                                compensation = MatchFunction.getCompensation()
                                manageBoxRestructering (rBox, lBox, compensation)

function manageBoxRestructering(boxToRemove, mGraphBox, compensation)
        if (compensation == null)
                mGraphBox.addParentBoxes(boxToRemove.getParentBoxes())
        else
                mGraphBox.addParentBox(compensation)
                compensation.addParentBoxes(boxToRemove.getParentBoxes())
        boxToRemove.removeAllParentBoxes()
        boxToRemove.removeAllChildBoxes()
```

Listing 2 Restructuring algorithm

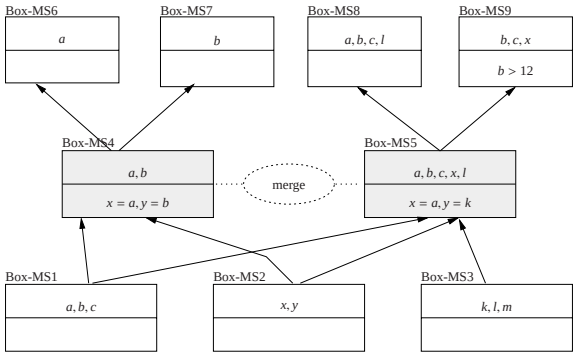

Figure 4. WG before Merging of the SELECT Boxes.

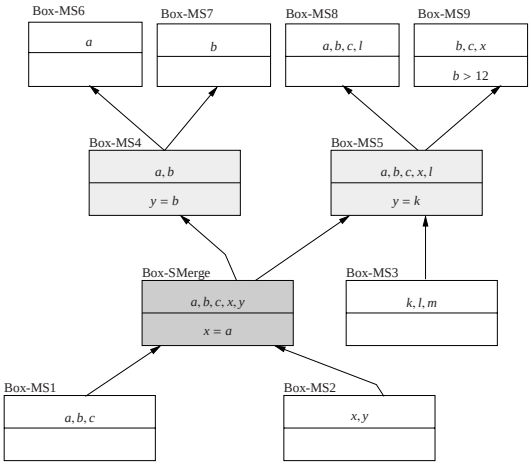

Figure 5. WG after Merging of the SELECT boxes.

The references of the subsumees to the child box will be deleted and the subsumer becomes the child box of the subsumees.

The third pattern deals with interval predicates. We do not require the complete subset condition as for insertion and restructuring to merge boxes with interval predicates. We only demand for a minimal overlap, i.e., the subsumees have to have an intersection relation. At this point, the execution of merging is dependent of profit calculation, i.e., the merging will be executed if the profit of the merge box is higher than the profit of the subsumees.

The fourth pattern considers conjunctions and disjunctions. The previous patterns only deal with conjunctive box predicates, but, we have to consider about disjunctive predicates, too. Each query containing disjunctive predicates is transformed to disjunctive normal form (DNF). Hence, we do not have to principally change the match function because the predicates of subsumee and subsumer are transformed into DNF. Afterwards, each conjunctive predicate of the subsumee will be matched to each conjunctive predicate of the subsumer. If all conjunctions of the subsumee are positive matched then the predicate match is positive, too.

The usage of super-group functions, e.g., roll-up, cube are not relevant to our extensions for now. Our approach can be easily extended with this functionality.

All in all, we introduced our approach using the WG with administration functionality (insertion and deletion). To reduce the overhead of our approach, we presented our restructuring and merging algorithm that reduce redundancy by insertion and find common parts for merging in the WG. Finally, we put all introduced algorithms together and obtain our prototype (`Workload Analyzer`). Hence, we developed

an analyzing working like an optimizer in a DBMS.

Cost Evaluation

We have to discuss the additional cost of our approach. An approach is not useful if the costs are higher than the obtained profit.

The costs of matching and optimization are proportionally to the size of WG. The size of the WG depends on the similarity of queries because similar queries will be represented by the same boxes. The navigation costs depend on the number of boxes visited for matching.

Traversing starts always from a TABLE box, i.e., we can estimate the WG as a tree starting from one TABLE box. The traversing cost is $log_{m_{WG}}(n_{WG})$ with order of the tree m_{WG} and number of boxes in the WG n_{QG}. But, any TABLE box of a WG can be a starting point. So, we have to consider the number of TABLE boxes as $\#_{TB}(WG)$, the costs of order are $O(\#_{TB}(WG) \times log_{m_{WG}}(n_{WG}))$.

We generate pairs of parent boxes of a box for the operations insertion, restructuring, and merging by traversing the query and the WG because the matching is needed for each of these algorithms. The cost is at most quadratic $\left(O\left(n^2\right)\right)$ with the maximum number of parent boxes *n*.

Experimental Evaluation

We evaluate our approach with a JAVA application whereby the database functionality is implemented. We simulate those parts of DBMS which are needed for the evaluation (table manager, WG administration, query processing and workload emulator). We compare our results with the application IBM DB2 Design Advisor (version 9.1). The used SQL compiler in our simulation is currently implemented only with restricted number of expressions of the SQL standard (Eisenberg & Melton, 1999), e.g., sub-selects, scalar functions, like or between conditions are not supported. The setup is the following: we submit *x* queries to both systems (DB2 and our prototype) and measure the execution time. Our prototype remembers all queries and integrates them into the WG. After recommendation of a MV set, the prototype transfer the create MV statements to the DBMS. So, both recommendations are processed in DB2.

Environment

The TCP-H benchmark is used for evaluation. Of its 22 queries we select eight appropriate queries (Q1, Q2, Q3, Q5, Q6, Q7, Q8, and Q10) ("TPC-H Benchmark", 2008). These eight queries are chosen because the other queries of the TCP-H benchmark test scalar and algebraic functions. These functions would require more complex simulation to examine the MV selection and we expect similar results.

Moreover, different point and range queries are interesting for applications, i.e., we have to extend the workload with such queries. So, we derive five queries from `Q10` of the eight queries to obtain different point and range queries. These five queries are modified by replacing and changing predicates but they still share common predicates, because applications often use the same common parts, e.g., the date column (predicate) in decision support systems.

All experiments were performed on an AMD Athlon 1700XP with 1GB RAM. The TCP-H execution time was 437 seconds without any MVs (scale factor 1GB) both systems are roughly equally fast. The IBM DB2 Design Advisor estimated six MVs for the given workload. We created these six MVs, and set the `CURRENT REFRESH AGE register` to `ANY`. This reduced execution time to 67 seconds. Our prototype (`Workload Analyzer`) estimated eight MVs for this workload and the execution time was about 5 seconds after creation. An overview of the results is presented in **Table 1**.

Discussion

The improvement of the execution time by using our prototype seems to be significant but the reason is trivial because the estimation of our system is much more specific for definition of MVs by same used cache size.

In the case of many point queries with groups of similar queries containing predicate intersection such general definition of MVs are more suitable. Many point queries can use the general definition whereby the MV can improve its profit. Otherwise, if we have hardly any groups of similar queries (with predicate intersection) then it will be advantageously for query processing at all. But, it will not satisfy the requirements of specific queries, i.e., it is necessary to calculate a composition to satisfy these queries because predicates and groupings are missing in a general MV. This slows the query processing down and the size of such general definitions can be a disadvantage, too.

In our test environment, the Design Advisor uses only join predicates for the estimation of MVs which is a suitable instrument but will not explore all possible optimizations. This fact leads the attention to execution of TCP-H query 10. The Advisor abandons completely the use of the date domain. The queries 10-2, 10-3, 10-4 and 10-5 match as subsets of TCP-H query 10 whereby our prototype estimates this MV. Additionally, the used cache by our concept application is smaller than by the Design Advisor because of the special estimation and use of interval predicates.

Conclusion

The presented work addressed the problem of automatic selection and administration of materialized views in a new way. The workload is represented in a graph that contains all queries and their statistics using the query graph model (QGM). The graph is continuously updated during query processing. Specialized algorithms allow the merging of queries. Therefore, common and specific views will be evaluated without using general assumptions like predicate reduction. The predicates were classified and referenced after their usage to cluster of boxes. Thereby, the estimation is not only possible on the query level but also on a subset of predicates. That gives us the possibility to map combined regions of a workload to a single materialization. Every cluster of predicates is estimated and materialized based on its profit computed from the reference value, calculation costs, and the

Query	without MVs (in sec)	DB2 Design Advisor (in sec)	Workload Analyzer (in sec)
TCP-H Query 1	62	0.044	0.018
TCP-H Query 2	2	0.374	0.022
TCP-H Query 3	39	≈ 10	0.029
TCP-H Query 5	39	0.018	0.017
TCP-H Query 6	32	0.019	0.017
TCP-H Query 7	33	0.017	0.017
TCP-H Query 8	58	0.907	0.022
TCP-H Query 10	42	≈ 10	1.125 (0.981 for groupingop.)
TCP-H Query 10-1	24	≈ 9	0.541
TCP-H Query 10-2	33	≈ 9	1.068
TCP-H Query 10-3	34	≈ 9	0.749
TCP-H Query 10-4	22	≈ 9	0.417
TCP-H Query 10-5	17	≈ 10	0.476

Table 1
Execution time of the compared systems.

size of relation. In that way, new MV definitions can be created, instead of using only actual queries as MV definitions. The WG is used as input for MV selection algorithms and caching strategies. The evaluation showed similar and better results than a state-of-art self-tuning advisor. Furthermore, following future research problems arise:

- Index selection: Indexes can be derived perfectly from the WG. The parent boxes of a materialized box contain all predicates which are necessary to create indexes for the view.
- Static parameters: In our test setup, we use a static upper and bottom reference level and a static factor to decrement. For dynamic workloads, the number of boxes is changing permanently. Hence, the estimations of materialized views will not be optimal.
- Temporal behaviour: Workloads often contains periodical sequences. Our approach has to be extended to retain such workloads.
- Thrashing: We propose a threshold that defines a minimum difference between the profit of old MV set and the new estimated set of MVs. So, we can avoid cache thrashing.
- Further optimization: All parts of the algorithm can be optimized and should be investigated in detail, in particular the merge pattern, the profit function, merging of boxes with no common predicate intersection.
- Cost function: Replacing the cost function by DBMS optimizer functionality could be a suitable way to improve cost estimations and adaptability.

References

Agrawal, S., Chaudhuri, S., & Narasayya, V. (2000). Automated selection of materialized views and indexes in sql databases. In *VLDB '00: Proceedings of 26th international conference on very large data bases, september 10-14, cairo, egypt* (p. 496-505).

Chaudhuri, S., & Narasayya, V. (2007). Self-tuning database systems: A decade of progress. In *VLDB '07: Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27* (p. 3).

Eisenberg, A., & Melton, J. (1999). SQL: Formerly known as SQL 3. *SIGMOD Record*, *28*(1), 131.

Kellerer, H., Pferschy, U., & Pisinger, D. (2004). *Knapsack problems*. Berlin, Heidelberg: Springer-Verlag.

Pirahesh, H., Hellerstein, J. M., & Hasan, W. (1992). Extensible/rule based query rewrite optimization in starburst. In *SIGMOD '00: Proceedings of the 26th ACM SIGMOD International Conference on Management of Data, June 2-5, San Diego, California, USA* (p. 39).

Scheuermann, P., Shim, J., & Vingralek, R. (1996). Watchman : A data warehouse intelligent cache manager. In *VLDB '96: Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India* (p. 51).

TPC-H Benchmark [Computer software manual]. (2008, September). Available from `http://www.tpc.org/tpch/spec/tpch2.8.0.pdf` (Standard Specification Revision 2.8.0)

Zaharioudakis, M., Cochrane, R., Lapis, G., Pirahesh, H., & Urata, M. (2000). Answering complex SQL queries using automatic summary tables. In *SIGMOD '00: Proceedings of the 26th ACM SIGMOD International Conference on Management of Data, May 16-18, Dallas, Texas, USA* (p. 105).

Zilio, D. C., Zuzarte, C., Lightstone, S., Ma, W., Lohman, G. M., Cochrane, R., et al. (2004). Recommending materialized views and indexes with IBM DB2 Design Advisor. In *1st International Conference on Autonomic Computing (ICAC), 17-19 May, New York, NY, USA* (p. 180).