Automated Selection of Materialized Views and Indexes for SQL Databases

Sanjay Agrawal
Microsoft Research
sagrawal@microsoft.com

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

Abstract

Automatically selecting an appropriate set of materialized views and indexes for SQL databases is a non-trivial task. A judicious choice must be cost-driven and influenced by the workload experienced by the system. Although there has been work in materialized view selection in the context of multidimensional (OLAP) databases, no past work has looked at the problem of building an industry-strength tool for automated selection of materialized views *and* indexes for SQL workloads. In this paper, we present an end-to-end solution to the problem of selecting materialized views and indexes. We describe results of extensive experimental evaluation that demonstrate the effectiveness of our techniques. Our solution is implemented as part of a tuning wizard that ships with Microsoft SQL Server 2000.

1. Introduction

In addition to indexes, today's commercial SQL database systems also support creation and use of materialized views. The presence of the right materialized views can significantly improve performance, particularly for decision support applications. However, to realize this potential, a judicious selection of materialized views is crucial.

Conceptually, both indexes and materialized views are physical structures that can significantly accelerate performance. An effective physical database design tool must therefore take into account the interaction between indexes and materialized views by considering them together to optimize the physical design for the workload on the system. Ignoring this interaction can significantly compromise the quality of recommendations. Despite a large number of recent papers in this area, most of the prior work considers the problems of index selection and materialized view selection in isolation.

Although indexes and materialized views are similar, a materialized view is much richer in structure than an index since a materialized view may be defined over multiple tables, and can have selections and GROUP BY over multiple columns. In fact, an index can logically be considered as a special case of a single-table, projection only materialized view. This richness of structure of materialized views makes the problem of selecting materialized views significantly more complex than that of index selection. We therefore need innovative techniques for dealing with the large space of potentially interesting materialized views that are possible for a given set of SQL queries and updates over a large schema. Previous papers on materialized view selection typically ignore this problem. Rather, they focus only on the "search" problem of picking an attractive set of materialized views from a given set. Thus, they implicitly assume that the given set is the set of all potentially interesting materialized views for the workload. Such an approach is simply not scalable in the context of SQL workloads. Finally, to be an effective solution, it is important to ensure that the solution to this problem is robust and takes into account the complexities of full SQL as a query language, as well as pragmatic issues such as the fact that in today's commercial database systems, it is often the case that the language of materialized views is a restricted subset of the language of queries. For example, a materialized view may not be allowed to contain nested sub-queries.

In this paper, we present an architecture and novel algorithms for addressing each of the above problems. Our work leverages previous work we did in building an index selection tool for Microsoft SQL Server [4,5], but requires several significant innovations. We establish that in order to pick a physical design consisting of indexes and materialized views, it is critical to search over the combined space of indexes and materialized views (Section 5). We quantify the impact on quality of *not* enumerating this space together, particularly in the presence of storage constraints or updates. Second, we present a principled way to identify a much smaller set of *candidate* materialized views such that searching over the reduced space of candidate materialized views preserves most of the gains of searching the entire space of possible

Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000

materialized views, at a fraction of the enumeration cost (Section 4). We introduce two key techniques that form the basis of a scalable approach for candidate materialized view selection. First, we show how to identify *interesting sets of tables* such that we need to consider materialized views only over such sets of tables. Next, we present a *view merging* technique that identifies candidate materialized views that while not optimal for any single query, can be beneficial to multiple queries in the workload. The techniques presented in this paper are designed to be robust for handling the generality of SQL as well as other pragmatic issues arising in index and materialized view selection. These techniques have enabled us to build an industry-strength physical database design tool that can determine an appropriate set of indexes, materialized views (and indexes on materialized views) for a given database and *workload* consisting of SQL queries and updates. This tool is now part of Microsoft SQL Server 2000's upcoming release. The extensive experimental results in this paper (Section 6) demonstrate the value of our proposed techniques. This work was done as part of the AutoAdmin [1] research project at Microsoft, which explores novel techniques to make databases self-tuning.

2. Architecture for Index and Materialized View Selection

An architectural overview of our approach to index and materialized view selection is shown in Figure 1. We assume that we are given a representative workload for which we need to recommend indexes and materialized views. One way to obtain such a workload is to use the logging capability of modern database systems to capture a trace of queries and updates faced by the system. Alternatively, customer or organization specific benchmarks may be used. As in our previous work on index selection [4], the key components of the architecture are: *syntactic structure selection*, *candidate selection*, *configuration enumeration*, and *configuration simulation and cost estimation.*

Given a workload, the first step is to identify *syntactically relevant* indexes, materialized views and indexes on materialized views that can potentially be used to answer the query. For example, consider a query **Q**: SELECT Sum(Sales) FROM Sales_Data WHERE City = 'Seattle'. For the query **Q**, the following materialized views (among others) are syntactically relevant: $\mathbf{v_1}$: SELECT Sum(Sales) FROM Sales_Data WHERE City = 'Seattle'. $\mathbf{v_2}$: SELECT City, Sum(Sales) FROM Sales_Data GROUP BY City. $\mathbf{v_3}$: SELECT City, Product, Sum(Sales) FROM Sales_Data GROUP BY City, Product. Optionally, we can consider additional indexes on the columns of the materialized view. Like indexes on base tables, indexes on materialized views can be single-column or multi-column, clustered or non-clustered, with the restriction that a given materialized view can have at most one clustered index on it. In this paper, we focus on the class of single-block materialized views consisting of selection, join, grouping and aggregation. The workload however, may consist of arbitrary SQL statements. In this paper, we do not consider materialized views that can be exploited using back-joins by the optimizer.

As mentioned in the introduction, searching the space of all syntactically relevant indexes and materialized views for a workload is infeasible in practice, particularly when the workload is large or complex. Therefore, it is crucial to eliminate spurious indexes and materialized views from consideration early, thereby focusing the search on a smaller, and interesting subset. The *candidate selection* module is responsible for identifying a set of traditional indexes, materialized views and indexes on materialized views for the given workload that are worthy of further exploration. Efficient selection of candidate materialized views is a key contribution of our work. For the purposes of this paper, we assume that candidate indexes have already been picked. For details on how candidate indexes may be chosen for a workload, we refer the reader to [4].

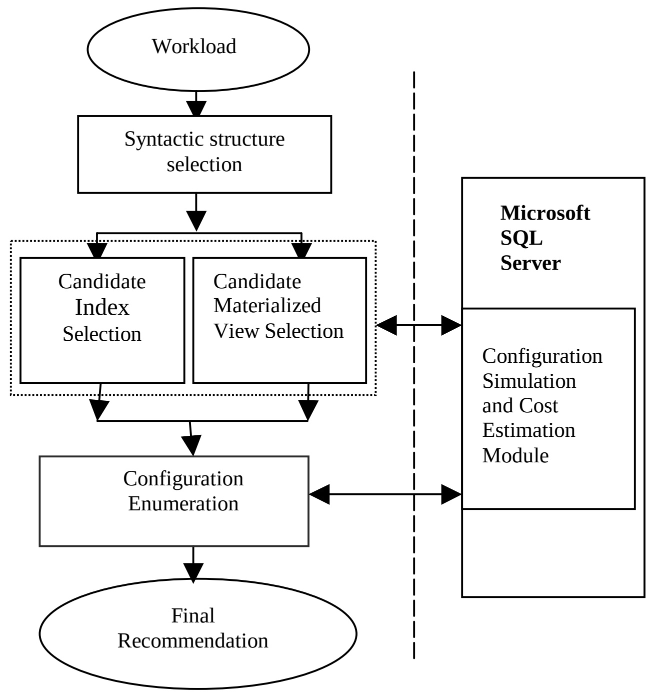

Figure 1. Architecture of Index and Materialized View Selection Tool

Once we have chosen a set of candidate indexes and candidate materialized views, we need to search among these structures to determine the ideal physical design, henceforth called a **configuration**. In our context, a configuration will consist of a set of traditional indexes, materialized views and indexes on materialized views. In this paper we will not discuss issues related to selection of indexes on materialized views due to lack of space. Despite the remarkable pruning achieved by the *candidate selection* module, searching through this space in a naïve fashion by enumerating all subsets of structures is infeasible. We adopt the same greedy algorithm for *configuration enumeration* as was used in [4]: Greedy(m,k). This algorithm returns a configuration consisting of a total of k indexes and materialized views. It first picks an optimal configuration of size up to m ($\leq$ k) by exhaustively enumerating all configurations of size up to m. It then picks the remaining (k-m) structures greedily. As will be shown in Section 6.2.4, this algorithm works well even when the set of candidates contains materialized views in addition to indexes. An important characteristic of our approach is that *configuration enumeration* is over the *joint* space of indexes and materialized views.

The configurations considered by the *configuration enumeration* module are compared for quality by taking into account the expected impact of the proposed configurations on the sum of the cost of queries in the workload. The *configuration simulation and cost estimation* module is responsible for providing this support. We have extended Microsoft SQL server to simulate the presence of indexes and materialized views that do not exist (referred to as "what-if" indexes and materialized views) to the query optimizer, and have also extended the optimizer costing module, so that given a query Q and a configuration C, the cost of Q when the physical design is the configuration C, may be computed. A detailed discussion of simulation of what-if structures is beyond the scope of this paper (see [5]). Finally, we note that index and materialized view maintenance costs are accounted for in our approach by the inclusion of updates/inserts/deletes statements in the workload.

3. Related Work

Recently, there have been several papers on selection of materialized views in the OLAP/Data Cube context [9,10,11,12,18]. These papers assume that the set of candidate materialized views is identical to the set of syntactically relevant materialized views for the workload[1]. As argued earlier, such a technique is not scalable for reasonably large SQL workloads since the space of syntactically relevant materialized views is very large. The focus of the above papers is almost exclusively on the *configuration enumeration* problem. In principle, their proposed enumeration schemes may be adopted in our architecture by simply substituting Greedy(m,k). Thus, we view the work presented in the above papers to be complementary to the work presented in this paper. Although one of the papers [11] studies the interaction of materialized views with indexes on materialized views, none of the papers consider interaction among selection of indexes on base tables and selection of materialized views. Thus, they implicitly assume that indexes are either already picked, or will be picked after selection of materialized views. As will be shown in this paper, both these alternatives severely impact quality of the solution.

The work by Baralis et al. [3] is also set in the context of OLAP/Data Cube and does not consider traditional indexes on base tables. For a given workload, they consider materialized views that exactly match queries in the workload, as well as a set of additional views that can leverage commonality among queries in the workload. Our technique for exploiting commonality among queries in the workload for candidate materialized view selection (Section 4.3) is different. Further, our techniques can also deal with arbitrary SQL workloads and materialized views with selection.

In the context of SQL databases and workloads, the work by [22] picks materialized views by examining the plan information of queries. However, since the plan is an artifact of the existing physical design, such an approach can lead to sub-optimal recommendations. The paper also suggests an alternative of examining all possible query plans of a query. However, the latter technique is not scalable for even moderately sized workloads.

There is a substantial body of work in the area of index selection that describes how to pick a good set of indexes for a given workload [4,8,16]. More recently, other commercial systems have also added support for automatically picking indexes [14,20]. The architecture adopted in our scheme is in the spirit of [4]. However, as noted above, the candidate materialized view selection as well as the comparison of alternative strategies to pick indexes on base tables along with materialized views, constitute novel and important contributions of this paper. Rozen [15] presents a framework for choosing a physical design consisting of various "feature sets" including indexes and materialized views. The space of materialized views considered in Rozen's thesis is restricted to single-table aggregation views with GROUP BY, whereas we allow materialized views to consist of join, selection, grouping and aggregation operators,

Some commercial systems (e.g., Redbrick/Informix [16] and Oracle 8i [14]) provide tools to tune the selection of materialized views for a workload. As with the body of the work referenced above, these tools exclusively recommend materialized views. In contrast, we present an integrated tool that can recommend indexes on base tables as well as materialized views (and indexes on them) by weighing in the impact of both on the performance of the workload. Finally, our paper is concerned with selection of materialized views but not with techniques to rewrite queries in the presence of materialized views.

4. Candidate Materialized View Selection

Considering all syntactically relevant materialized views for a workload in the *configuration enumeration* phase (see Figure 1) is not scalable since it would explode the space of configurations that must be searched. The space of syntactically relevant materialized views for a query (and hence a workload) is very large, since in principle, a materialized view can be proposed on any subset of tables in the query. Furthermore, even for a given *table-subset* (a table-subset is a subset of tables referenced in a query in the workload.), there is an explosion in the space of materialized views arising from selection conditions and group by columns in the query. If there are *m* selection conditions in the query on a table-subset T, then materialized views containing any subset of these selection conditions are syntactically relevant. Therefore, the goal of *candidate materialized view selection* is to quickly eliminate materialized views that are syntactically relevant for one or more queries in the workload but are never used in answering any query from entering the *configuration enumeration* phase.

We observe that the obvious approach of selecting one candidate materialized view per query that exactly matches each query in the workload does not work since in many database systems the language of materialized views may not match the language of queries. For example, nested sub-queries can appear in the query but may not be part of the materialized view language.

[1] Typically, these are aggregation views over subsets of dimensions. For each subset of dimensions, multiple aggregate views are possible in the presence of dimension hierarchy.

Moreover, in storage-constrained environments, ignoring commonality across queries in the workload can result in sub-optimal quality. This problem is even more severe in large workloads. The following simplified example of Q_1 from the TPC-H benchmark illustrates this point:

Example 1. Consider a workload consisting of 1000 queries of the form: SELECT l_returnflag, l_linestatus, SUM(l_quantity) FROM lineitem WHERE l_shipdate BETWEEN <Date1> and <Date2> GROUP BY l_returnflag, l_linestatus. Assume that each of the 1000 queries has different constants for <Date1> and <Date2>. Then, rather than recommending 1000 materialized views, the following materialized view that can service all 1000 queries may be more attractive for the entire workload: SELECT l_shipdate, l_returnflag, l_linestatus, SUM(l_quantity) FROM lineitem GROUP BY l_shipdate, l_returnflag, l_linestatus.

A second observation that influences our approach to candidate materialized view selection is that there are certain table-subsets such that, even if we were to propose materialized views on those subsets it would only lead to a small reduction in cost for the entire workload. This can happen either because the table-subsets occur infrequently in the workload or they occur only in inexpensive queries.

Example 2. Consider a workload of 100 queries whose total cost is 10,000 units. Let T be a table-subset that occurs in 25 queries whose combined cost is 50 units. Then even if we considered all syntactically relevant materialized views on T, the maximum possible benefit of those materialized views for the workload is 0.5%.

Furthermore, even among table-subsets that occur frequently or occur in expensive queries, not all table-subsets are likely to be equally useful.

Example 3. Consider the TPC-H 1GB database and the workload specified in the benchmark. There are several queries in which the tables, lineitem, orders, nation, and region co-occur. However, it is likely that materialized views proposed on the table-subset {*lineitem*, *orders*} are more useful than materialized views proposed on {*nation*, *region*}. This is because the tables *lineitem* and *orders* have 6 million and 1.5 million rows respectively, but tables *nation* and *region* are very small (25 and 5 rows respectively). Hence, the benefit of pre-computing the portion of the queries involving {*nation*, *region*} is insignificant compared to the benefit of pre-computing the portion of the query involving {*lineitem*, *orders*}.

Based on these observations, we approach the task of *candidate materialized view selection* using three steps: (1) From the large space of all possible table-subsets for the workload, we arrive at a smaller set of interesting table-subsets (Section 4.1). (2) Based on these interesting table-subsets, we propose a set of materialized views for each query in the workload, and from this set we select a configuration that is best for that query. This step uses a cost-based analysis for selecting the best configuration for a query (Section 4.2). (3) Starting with the views selected in (2), we generate an additional set of "merged" materialized views in a controlled manner such that the merged materialized views can service multiple queries in the workload (Section 4.3). The new set of merged materialized views, along with the materialized views selected in (2) is the set of candidate materialized views that enters *configuration enumeration*. We now present the details of each of these steps.

4.1. Finding Interesting Table-Subsets

Our goal is to find "interesting" table-subsets from among all possible table-subsets for the workload, and restrict the space of materialized views considered to only those table-subsets. Intuitively, a table-subset T is interesting if materializing one or more views on T has the potential to reduce the cost of the workload significantly, i.e., above a given threshold. Thus, the first step is to define a metric that captures the relative importance of a table-subset.

Consider the following metric: *TS-Cost(T)* = total cost[2] of all queries in the workload (for the current database) where table-subset T occurs. The above metric, while simple, is not a good measure of relative importance of a table-subset. For example, in the context of Example 3, if all queries in the workload referenced the tables *lineitem*, *orders*, *nation*, and *region* together, then using the *TS-Cost(T)* metric, the table-subsets T_1 = {*lineitem*, *orders*} would have the same importance as the table-subset T_2 = {*nation*, *region*} even though a materialized view on T_1 is likely to be much more useful than a materialized view on T_2. Therefore, we propose the following metric that better captures the relative importance of a table-subset: *TS-Weight(T)* = $\sum_i$ *Cost*(Q_i)*(sum of sizes of tables in T)/ (sum of sizes of all tables referenced in Q_i)), where the summation is only over queries in the workload where T occurs. Observe that *TS-Weight* is a simple function that can discriminate between table-subsets even if they occur in exactly the same queries in the workload. A complete evaluation of this and alternative functions, and their relationship to cost estimation by the query optimizer is part of our ongoing work.

1. Let S_1 = {T | T is a table-subset of size 1 satisfying *TS-Cost*(T) $\geq$ **C**}; i = 1
2. **While** i < MAX-TABLES and $|S_i| > 0$
3.   i = i + 1; S_i = {}
4.   Let G ={T | T is a table-subset of size i, and $\exists$ s $\in$ S_{i-1} such that s $\subset$ T}
5.   **For** each T $\in$ G
       If *TS-Cost* (T) $\geq$ **C Then** $S_i = S_i \cup$ {T}
6.   **End For**
7. **End While**
8. S = $S_1 \cup S_2 \cup \ldots S_{\text{MAX-TABLES}}$
9. R = {T | T $\in$ S and *TS-Weight*(T) $\geq$ **C**}
10. **Return** R

Figure 2. Algorithm for finding interesting table-subsets in the workload.

Although *TS-Weight(T)* is a reasonable metric for relative importance of a table-subset, there does not appear to be an obvious efficient algorithm for finding all table subsets whose *TS-Weight* exceeds a given threshold. In contrast, the *TS-Cost* metric has the property of "monotonicity" since for table subsets T_1, T_2, $T_1 \subseteq T_2 \Rightarrow$ *TS-Cost*(T_1) $\geq$ *TS-Cost*(T_2). This is because in all queries where T_2 occurs, T_1 (and likewise all other subsets of T_2)

[2] The cost of a query (or update statement) Q, denoted by *Cost*(Q) can be obtained from the *configuration simulation and cost estimation* shown in Figure 1.

also occur. This monotonicity property of *TS-Cost* allows us to leverage efficient algorithms proposed for identifying frequent itemsets, e.g., as in [2], to identify all table-subsets whose *TS-Cost* exceeds the specified threshold. Fortunately, it is also the case that if *TS-Weight(T)* $\geq$ **C** (for any threshold **C**), then *TS-Cost(T)* $\geq$ **C**. Therefore, our algorithm (shown in Figure 2) for identifying interesting table-subsets by the *TS-Weight* metric has two steps: (a) Prune table-subsets not satisfying the given threshold using the *TS-Cost* metric. (b) Prune the table-subsets retained in (a) that do not satisfy the given threshold using the *TS-Weight* metric. We note that the efficiency gained by the algorithm is due to reduced CPU and memory costs by not having to enumerate all table-subsets.

In Figure 2, we define the *size* of a table-subset T to be the number of tables in T. MAX-TABLES is the maximum number of tables referenced in any query in the workload. A lower threshold **C** leads to a larger space being considered and vice versa. Based on experiments on various databases and workloads, we found that using **C** = 10% of the total workload cost had a negligible negative impact on the solution compared to the case when there is no cut off (**C** = 0), but was significantly faster (see Section 6.2.1 for details).

4.2. Exploiting the Query Optimizer to Prune Syntactically Relevant Materialized Views

The algorithm for identifying interesting table-subsets presented in Section 4.1 significantly reduces the number of syntactically relevant materialized views that must be considered for a workload. Nonetheless, many of these views may still not be useful for answering any query in the workload. This is because the decision of whether or not a materialized view is useful in answering a query is made by the *query optimizer* using cost estimation. Therefore, our goal is to prevent syntactically relevant materialized views that are not used in answering any query from being considered during *configuration enumeration*. We achieve this goal using the algorithm shown in Figure 3, which is based on the intuition that if a materialized view is not part of the best solution for even a single query in the workload, then it is unlikely to be part of the best solution for the entire workload. This approach is similar to the one used in [4] for selecting candidate indexes. For a given query Q, and a set S of materialized views (and indexes on them) proposed for Q, Step 4 of our algorithm assumes the existence of the function *Find-Best-Configuration*(Q, S) that returns the best configuration for Q from S. *Find-Best-Configuration* has the property that the choice of the best configuration for a query is cost based, i.e., it is the configuration that the optimizer estimates as having the lowest cost for Q. Any suitable search method can be used in this function, e.g., the Greedy(m,k) algorithm described in Section 2. We also see that in the presence of updates or storage constraints, we may need to pick more than one configuration for a query (e.g., the **n** best configurations) in Step 4 to maintain quality, at the expense of increased running time during *configuration enumeration* [4].

Next we discuss the issue of which syntactically relevant materialized views should proposed for a query Q_i in Step 3. Observe that among the interesting table-subsets that occur in Q_i, it is not sufficient to propose materialized views only on the table-subset that exactly matches the tables referenced in Q_i. One reason for this is that the language of views may not match the language of queries, e.g., the query may contain a nested sub-query whereas the view cannot. Also, for complex queries the query optimizer performs algebraic transformations of the query to find a better execution plan. In such cases, determining which of the interesting table-subsets to consider requires analysis of the structure of the query as well as knowledge of the transformations considered by the query optimizer. We also note that due to the pruning of table-subsets in previous step (Section 4.1), the table-subset that exactly matches the tables referenced in the query may not even be deemed interesting. In such cases, it again becomes important to consider smaller interesting table-subsets that occur in Q_i. Fortunately, due to the effective pruning achieved by the algorithm for finding interesting table-subsets (Figure 2), we are able to take the simple approach of proposing syntactically relevant materialized views for a query Q_i on *all* interesting table subsets that occur in Q_i.

1. M = {} /* M is the set of materialized views that is useful for at least one query in the workload W*/
2. **For** i = 1 to |W|
3. Let S_i = Set of materialized views proposed for query Q_i.
4. C = *Find-Best-Configuration* (Q_i, S_i)
5. M = M $\cup$ C;
6. **End For**
7. **Return** M

Figure 3. Cost-based pruning of syntactically relevant materialized views.

For each such interesting table-subset T, we propose (in Step 3): (1) A "pure-join"[3] materialized view on T containing join and selection conditions in Q_i on tables in T. (2) If Q_i has grouping columns, then a materialized view similar to (1) but also containing GROUP BY columns and aggregate expression from Q_i on tables in T. It is also possible to propose additional materialized views on a table-subset that include only a *subset* of the selection conditions in the query on tables in T, since such views may also apply to other queries in the workload. However, in our approach, this aspect of exploiting commonality across queries in the workload is handled via view merging (Section 4.3). For each materialized view proposed, we also propose a set of clustered and non-clustered indexes on the materialized view. We omit the details of this discussion due to lack of space. Our experiments (see Section 6.2.3) show that the above algorithm is not only efficient, but it dramatically reduces the number of materialized views that need to be considered in *configuration enumeration* (Figure 1).

4.3. View Merging

We observe that if the materialized views that enter *configuration enumeration* are limited to the ones selected

[3] In principle, such a "pure-join" materialized view can also be generated via view merging (Section 4.3). We omit this discussion due to lack of space.

by the algorithm presented in Section 4.2, then we can get sub-optimal recommendations for the workload when storage is constrained (see Example 1). This observation suggests that we need to consider the space of materialized views that although are not optimal for any individual query, are useful for multiple queries, and therefore may be optimal for the workload. However, proposing such a set of syntactically relevant materialized views by analyzing multiple queries at once could lead to an explosion in the number of merged materialized views proposed. Instead, our approach is based on the observation that M, the set of materialized views returned by the algorithm in Figure 2 (Section 4.2), contains materialized views selected on a cost-basis and are therefore sure (or very likely) to be used by the query optimizer. This set M is therefore a good starting point for generating additional "merged" materialized views that are derived by exploiting commonality among views in M. The newly generated set of merged views, along with M, are our candidate materialized views. Our approach is significantly more scalable than the alternative of generated merged views starting from all syntactically relevant materialized views.

An important issue in view merging is characterizing the space of merged views to be explored. In our approach, we have decided to explore this space using a sequence of pair-wise merges. Thus, the two key issues that must be addressed are: (1) determining the criteria that govern when and how a given pair of views is merged (Section 4.3.1), and (2) enumerating the space of possible merged views (Section 4.3.2). Architecturally, our approach for view merging is similar to the one adopted in our prior work on *index merging* [7]. However, the algorithm for merging a pair of views needs to recognize the fact that views (unlike indexes) are multi-table structures that may contain selections, grouping and aggregation. These differences significantly influence the way in which a given pair of views is merged.

4.3.1. Merging a Pair of Views

Our goal when merging a given pair of views, referred to as the *parent* views, is to generate a new view, called the *merged* view, which has the following two properties. First, all queries that can be answered using either of the parent views should be answerable using the merged view. Second, the cost of answering these queries using the merged view should not be significantly higher than the cost of answering the queries using views in M (the set obtained using algorithm in Figure 2). Our algorithm for merging a pair of views, called *MergeViewPair*, is shown in Figure 4. Intuitively, the algorithm achieves the first property by structurally modifying the parent views as little as possible when generating the merged view, i.e., by retaining the common aspects of the parent views and generalizing only their differences. For simplicity, we present the algorithm for SPJ views with grouping and aggregation, where the selection conditions are conjunctions of simple predicates. The algorithm can be generalized to handle complex selection conditions as well as account for differences in constants between conditions on the same column.

Note that a merged view v may be derived starting from views in M through a sequence of pair-wise merges. We define Parent-Closure(v) as the set of views in M from which v is derived. The goal of Step 4 in the *MergeViewPair* algorithm is to achieve the second property mentioned above by preventing a merged view from being generated if it is much larger than the views in Parent-Closure(v). Precisely characterizing the factors that determine the value of the size increase threshold (**x**) requires further work. In our implementation on Microsoft SQL Server, we have found that setting **x** between 1 and 2 works well over a variety of databases and workloads.

1. Let v_1 and v_2 be a pair of materialized views that reference the same tables and the same join conditions.
2. Let $s_{11}, \ldots s_{1m}$ be the selection conditions that occur in v_1 but not in v_2. Let $s_{21}, \ldots s_{2n}$ be the selection conditions that occur in v_2 but not in v_1.
3. Let v_{12} be the view obtained by **(a)** taking the union of the projection columns of v_1 and v_2 **(b)** taking the union of the GROUP BY columns of v_1 and v_2 **(c)** pushing the columns $s_{11}, \ldots s_{1m}$ and $s_{21}, \ldots s_{2n}$ into the GROUP BY clause of v_{12} and **(d)** including selection conditions common to v_1 and v_2.
4. **If** $((|v_{12}| >$ Min Size (Parent-Closure $(v_1) \cup$ Parent-Closure $(v_2)) *$ **x**) **Then Return** Null.
5. **Return** v_{12}.

Figure 4. MergeViewPair algorithm

We note that in Step 4, *MergeViewPair* requires estimating the size of a materialized view. One way to achieve this is to obtain an estimate of the view size from the query optimizer. The accuracy of such estimation depends on the availability of an appropriate set of statistics for query optimization [6]. Alternatively, less expensive heuristic techniques have been proposed in [19] for more restricted multidimensional scenarios.

1. R = M
2. **While** (|R| > 1)
3. Let M' = The set of merged views obtained by calling *MergeViewPair* on each pair of views in R.
4. **If** M' = {} **Return** (R–M)
5. R = R $\cup$ M'
6. For each view v $\in$ M', remove both parents of v from R
7. **End While**
8. **Return** (R–M).

Figure 5. Algorithm for generating a set of merged views from a given set of views M

4.3.2. Algorithm for generating merged views

Our algorithm for generating a set of merged views from a given set of views is shown in Figure 5. As mentioned earlier, we invoke this algorithm with the set of materialized views M (obtained using the algorithm in Figure 3). We comment on several properties of the algorithm. First, note that it is possible for a merged materialized view generated in Step 3 to be merged again in a subsequent iteration of the outer loop (Steps 2-7). This allows more than two views in M to be combined into one merged view even though the merging is done pair-wise. Second, although the number of new merged views explored by this algorithm can be exponential in the size of M in the worst case, we observe that much fewer merged materialized views are explored in practice (see Section 6.2.3) because of the checks built into Step 4 of *MergeViewPair* (Figure 4). Third, the set of merged views returned by the algorithm does not depend on the exact sequence in which views are merged (we omit the proof due to lack of space). Furthermore, the algorithm is guaranteed to explore all merged views that can be generated using *any* sequence of merges using *MergeViewPair* starting with the views in M. The algorithm in Figure 5 has been presented in its current form for simplicity of exposition. Note however, that if views v_1 and v_2 *cannot* be merged to form v_{12}, then no other merged view derived from v_1 *and* v_2 is possible, e.g., v_{123} is not possible. We can leverage this observation to increase efficiency by using techniques for finding frequent itemsets, e.g., as in [2]. Finally, we can ensure that merged views generated by this algorithm are actually useful in answering queries in the workload, by performing a cost-based pruning using the query optimizer (similar to the algorithm in Figure 3).

5. Trading Choices of Indexes and Materialized Views

Previous work in physical database design has considered the problems of index selection and materialized view selection in isolation. However, both indexes and materialized views are fundamentally similar – both are redundant structures that speed up query execution, compete for the same resource – storage, and incur maintenance overhead in the presence of updates. Not surprisingly, indexes and materialized views can interact with one another, i.e., the presence of an index can make a materialized view more attractive and vice versa. Therefore, as described in Section 2, our approach is to consider joint enumeration of the space of candidate indexes and materialized views. In this section, we compare our approach to alternative approaches and quantify the benefit of joint enumeration.

There are two alternatives to our approach of jointly enumerating the space of indexes and materialized views. One alternative is to pick materialized views first, and then select indexes for the workload given the materialized views picked earlier (we denote this alternative by **MVFIRST**). The second alternative reverses the above order and picks indexes first, followed by materialized views (**INDFIRST**). We have implemented these alternatives on Microsoft SQL Server 2000, and conducted extensive experiments to compare their quality and efficiency. These experiments (see Section 6.2.5) support the hypothesis that joint enumeration results in significantly better quality solutions than the two alternatives, and also shows the scalability of our approach.

5.1. Selecting one feature set followed by the other

In both MVFIRST and INDFIRST, if the global storage bound is S, then we need to determine a fraction **f** (0≤ **f** ≤1), such that a storage constraint of **f***S is applied to the selection of the first feature set. After selecting the first feature set, all the remaining storage can be used when picking the second feature set. This raises the issue of how to determine the fraction **f** of the total storage bound to be allocated to the first feature set? In practice, the "optimal" fraction **f** depends on several attributes of the workload including amount of updates, complexity of queries; as well as the absolute value of the total storage bound. In our empirical evaluation of MVFIRST and INDFIRST we found that the optimal value of **f** changes from one workload to the next. Furthermore, even at this optimal data point, the quality of the solution is inferior in most cases compared to our approach (JOINTSEL). Another problem relevant to both INDFIRST and MVFIRST is redundant recommendations if the feature selected second is better for a query than the feature selected first. This happens since the feature set selected first is fixed and cannot be back-tracked subsequently.

A further drawback of MVFIRST is that selecting materialized views first can adversely affect the quality of candidate indexes picked. This is because for a given query, the best materialized view is likely to be more beneficial than the best index since a materialized view can pre-compute (parts of) the query (via aggregations, grouping, joins etc.). Therefore, when materialized views are chosen first, they are likely to preclude selection of potentially useful candidate indexes for the workload.

5.2. Joint Enumeration

The two attractions of joint enumeration of candidate indexes and materialized views are: (a) A graceful adjustment to storage bounds, and (b) Considering interactions between candidate indexes and candidate materialized views that are not possible in the other approaches. For example, consider a query Q for which indexes I_1, I_2 and materialized view v are candidates. Assume that I_1 alone reduces the cost of Q by 25 units and I_2 reduces the cost by 30 units, but I_1 and v together reduce the cost by 100 units. Then, using INDFIRST, I_2 would eliminate I_1 when indexes are picked, and we would not be able to get the optimal recommendation {I_1, v}. We use the Greedy(m,k) algorithm for enumeration, which allows us to treat indexes, materialized views and indexes on materialized views on the same footing. We demonstrate the quality and scalability of this algorithm for index and materialized view selection in Section 6.2.4.

6. Experiments

We have implemented the algorithms presented in this paper on Microsoft SQL Server 2000. In the first set of experiments, we evaluate the quality and running time of our algorithm for selecting *candidate materialized views* (Section 4). We demonstrate that: (1) Our algorithm for identifying interesting table-subsets for a workload (Section 4.1) does not eliminate useful materialized views, while substantially reducing the number of materialized views that need to be proposed. (2) The application of our view merging algorithm (Section 4.3) significantly improves quality of the recommendation specially when storage is at a premium.

Our second set of experiments is related to the architectural issues in this paper. We show that: (1) Our *candidate selection* module (Figure 1) significantly reduces the running time compared to an exhaustive scheme that does not use this module, while maintaining high quality recommendations. (2) Our *configuration enumeration* module Greedy(m,k) gives results comparable to an exhaustive algorithm that enumerates over all subsets of candidates, and runs significantly faster. (3) Our approach for joint enumeration over the space of indexes and materialized views (JOINTSEL) gives significantly better solutions than MVFIRST or INDFIRST.

6.1. Experimental Setup

The experiments were run on two Dell Precision 610 machines with 550 Mhz CPU and 256 MB RAM. The databases used for our tests were stored on an internal 16.9 GB hard drive.

Databases: The algorithms presented in this paper have been extensively tested on several real and synthetic databases as part of the shipping process of the tuning wizard for Microsoft SQL Server 2000. However, due to lack of space and the intrinsic difficulty of comparing our algorithms with "optimal" algorithms on large workloads, we limit our experiments to relatively small workloads on the TPC-H [20] 1GB database as well as one real-world database used within Microsoft to track the sales of products by the company. Therefore, the experiments presented should be interpreted as illustrative rather than exhaustive empirical validation.

Name	#queries	Remarks
TPCH-22	22	TPC-H benchmark
TCPH-UPD25, TCPH-UPD75	25 25	25 % update statements 75% update statements
WKLD-4-TBL, WKLD-8-TBL	100 100	Max 4-table queries Max 8-table queries
WKLD-VM	50	Real-world workload
WKLD-SCALE (n)	n = 25, 50, 75, 100, 125	Workloads of increasing size

Table 1. Summary of workloads used in experiments.

Workloads: The workloads used in our experiments are summarized in Table 1. We created the synthetic workloads using a program that can generate Select, Insert, Delete and Update statements. The queries generated by this program are limited to Select, Project, Join queries with Group By and Aggregation. Nested sub-queries connected via an EXISTS clause can also be generated. In all experiments we use the cost of the workload for the recommended configuration as a measure of the *quality* of that configuration.

6.2. Experimental Results

6.2.1. Evaluation of algorithm for identifying interesting table-subsets

In this experiment, we evaluate the reduction in number of syntactically relevant materialized views proposed by our algorithm (see Section 4.1) and its impact on quality compared to an approach that exhaustively proposes all syntactically relevant materialized views. We carry out this comparison for three workloads: TPCH-22 (the original benchmark), WKLD-4TBL, and WKLD-8TBL (see Table 1). We used a threshold of **C**=10%. Figure 6 shows that across all three workloads, our algorithm achieves significant pruning of the space of syntactically relevant materialized views. Furthermore, as seen in Figure 7, we see a small drop in quality. This experiment shows that our pruning is effective and yet does not miss out on important table-subsets.

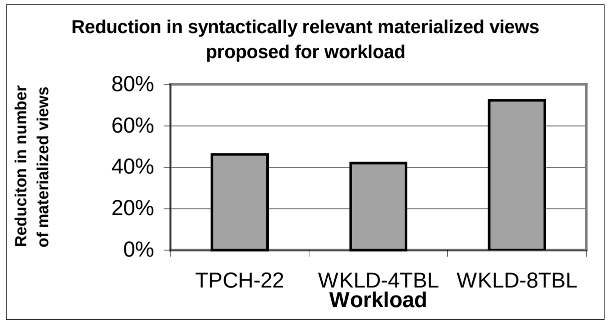

Figure 6. Reduction in syntactically relevant materialized views proposed compared to Exhaustive

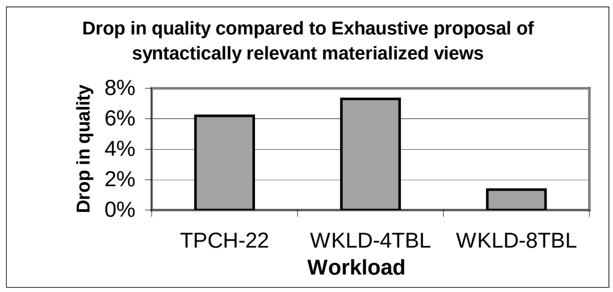

Figure 7. Comparison of quality of our algorithm to Exhaustive.

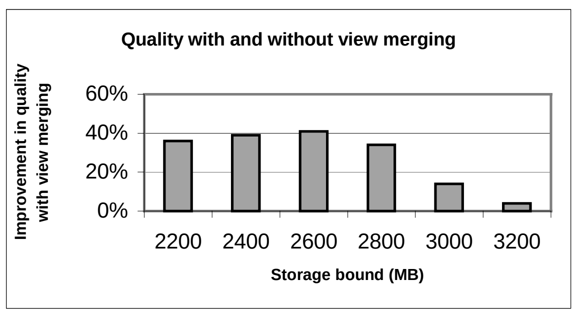

Figure 8. Quality vs. storage bound with and without view merging.

6.2.2. Evaluation of view merging algorithm

Next, we illustrate the importance of view merging (Section 4.3) using workload WKLD-VM (see Table 1), which consists of 50 real-world queries (SPJ with grouping and aggregation). We compare two versions of our algorithm – with and without our view merging module. Figure 8 shows the improvement in quality of the solution as the total storage bound is varied from 2.2GB to 3.2 GB. We see that at low storage constraints the version with view merging significantly outperforms the version without view merging. As expected, when the storage bound is increased, the two versions converge to the same solution. For the above workload the number of additional merged views proposed was about 19%, and the increase in running time due to view merging was about 9%. Finally, we note that yet another positive aspect of view merging is that it produces more compact recommendations (i.e., having fewer materialized views).

Workload	**Ratio of running time**	**% improv. in quality *Without***	**% improv. in quality *With***
TPC-H queries Q_1, Q_2, Q_3	64	98.1%	97.6%
TPC-H queries Q_4, Q_5	13	93.6%	93.6%
TPC-H queries Q_6, Q_7, Q_8	31	73.4%	73.4%
TPC-H queries Q_9,Q_{10},Q_{11}	14	66.6%	60.1%

Table 2. Comparison of schemes *with* and *without* the candidate selection module.

6.2.3. Evaluation of Candidate Selection

Table 2 compares the running time and quality of our approach to an exhaustive approach in which the *candidate selection* step (Section 4) is omitted, i.e. all syntactically relevant materialized views and indexes are considered in the *configuration enumeration*. In both cases, we use Greedy(m,k) as the algorithm for *configuration enumeration*. Due to the large running time of the version without *candidate selection*, we restrict each workload to a small subset of the TPC-H workload. The table shows that *candidate selection* not only reduces the running time by several orders of magnitude, but the drop in quality resulting from this pruning is very small. This experiment emphasizes the importance of restricting enumeration to a set of candidates rather than all syntactically relevant indexes and materialized views. In Figure 9 we evaluate the scalability of our candidate materialized view selection technique (Section 4) as the workload size (using workloads WKLD-SCALE(n)) is increased from 25 to 125. We see that the number of candidate materialized views grows approximately linearly with the workload size.

6.2.4. Evaluation of Enumeration algorithm

In this experiment, we show that the Greedy(m,k) algorithm for *configuration enumeration* over the space of candidate indexes and materialized views: (a) performs well with respect to quality of recommendation compared to an exhaustive algorithm that enumerates over all subsets of candidates and (b) is significantly faster than the exhaustive approach. Table 3 shows that the Greedy(m,k) algorithm (with m=2) gives a solution that is comparable in quality to exhaustive enumeration, while it runs about an order of magnitude faster on both workloads.

6.2.5. JOINTSEL vs. MVFIRST vs. INDFIRST

We first compare the quality and running time of our architecture for selecting indexes and materialized views, JOINTSEL (Section 5.2), with the two alternative architectures MVFIRST and INDFIRST (Section 5.1). We study the quality of these alternatives when they are not subject to any storage constraint (i.e., storage = ∞). Table 4 shows that even with no storage constraint the quality of solution using MVFIRST is significantly worse than the quality of JOINTSEL, particularly in the presence of updates in the workload. This confirms our intuition that picking materialized views first adversely affects the subsequent selection of indexes (see Section 5.1) even in a query only workload (TPCH-22). In the presence of updates, the solution of MVFIRST degenerates rapidly (TPCH-UPD25) compared to JOINTSEL. We therefore drop this alternative from further experiments. We note that the quality of INDFIRST is comparable to JOINTSEL on TPCH-22 when storage is not an issue. In the presence of updates (TPCH-UPD25) however, the INDFIRST recommendations are inferior compared to JOINTSEL.

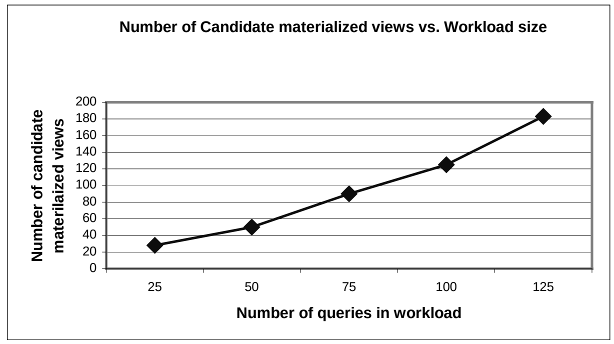

Figure 9. Scalability of candidate materialized view selection with workload size

Workload	**Ratio of Running Time: Exhaustive to Greedy (m, k)**	**% improv in quality with Exhaustive**	**% improv in quality *withGreedy (m, k)***
TPCH-22	11	83%	81%
TPCH-UPD25	9	79%	77%

Table 3. Comparison of Greedy(m,k) and exhaustive enumeration algorithms.

Workload	**Drop in quality of MVFIRST compared to JOINTSEL**	**Drop in quality of INDFIRST compared to JOINTSEL**
TPCH-22	8%	0%
TPCH-UPD25	67%	11%

Table 4. Comparison of alternative schemes without storage bound (i.e., storage = ∞)

Next, we compare the quality of JOINTSEL and INDFIRST with varying storage. INDFIRST (**f**) denotes that fraction **f** of the total additional storage space is available for indexes. (with **f** = 0.25, 0.50, 0.75). We vary the additional storage allowed (**s**) between 25% of the current database size to 100% of the current database size. Figure 10 shows that JOINTSEL consistently outperforms INDFIRST for the TPCH-22 workload. In addition, we observe that for **s**=1, **f**=0.75 is the optimal partitioning fraction whereas for **s**=0.5, **f**=0.50 is the right fraction. *For a given database and a workload, the optimal storage partitioning varies with the storage constraint.* Finally, we study the behavior of INDFIRST vs. JOINTSEL for three workloads and a fixed total storage, as the fraction of storage allotted to indexes (**f**) is varied. Figure 11 shows that the best allocation fraction is different for each workload, e.g., **f** = 0.25 is best for TPC-H and TPCH-UPD25 but **f** = 0.50 is optimal for TPCH-UPD75. *For a given database and a storage space, the "right" partition varies with the workload.* In contrast, we see the consistently high quality of JOINTSEL across various workloads. We also note that the running time of JOINTSEL and INDFIRST are comparable to one another (within approximately 10% of each other for the workloads we experimented with). For example, for the data point where additional storage allowed = 100%, for the TPCH-22 workload, JOINTSEL is slightly faster than INDFIRST (**f**=0.50) by about 4% whereas for the TPCH-UPD25 workload, INDFIRST is faster by about 6%.

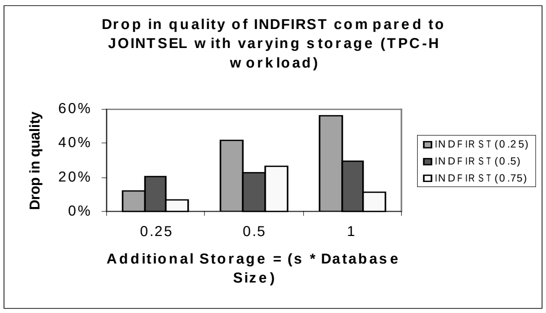

Figure 10. Quality of INDFIRST vs. JOINTSEL with varying storage bound.

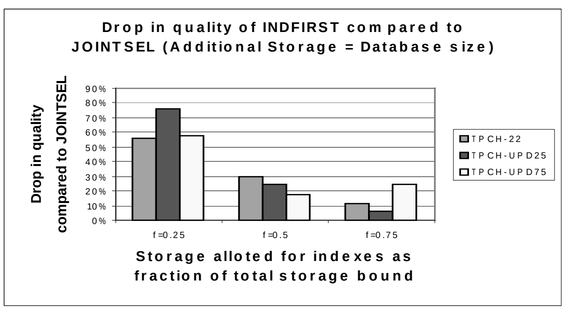

Figure 11. Quality of INDFIRST vs. JOINTSEL with varying storage partitioning (f).

7. Conclusion

The architecture and novel algorithms presented in this paper are the foundation of a robust physical database design tool for Microsoft SQL Server 2000 that can recommend both indexes and materialized views. In a recent paper, Kotidis et al.[13] present a technique for OLAP databases to dynamically determine which materialized views should be maintained. Extending this paradigm to SQL workloads is a significantly more complex problem, but is worth exploring. Another challenging task is developing a theoretical framework and appropriate abstractions for physical database design that is able to capture complexities of the physical design problem, and thus enables us to compare properties of alternative algorithms. Finally, note that indexes and materialized views are only a part of the physical design space. In the context of the AutoAdmin project [1], we continue to pursue our long-term goal of a complete physical design tool for SQL databases.

8. Acknowledgments

We thank Gautam Das for his help in analyzing the main algorithms presented in this paper. We thank the Microsoft SQL Server team with their help in providing the necessary server-side support for our implementation.

9. References

1. AutoAdmin project, Microsoft Research. http://www.research.microsoft.com/dmx/AutoAdmin
2. Agrawal R., Ramakrishnan, S. Fast Algorithms for Mining Association Rules in Large Databases, VLDB 1994.
3. Baralis E., Paraboschi S., Teniente E., Materialized View Selection in a Multidimensional Database, VLDB 1997.
4. Chaudhuri S., Narasayya V., An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. VLDB 1997.
5. Chaudhuri S., Narasayya V., AutoAdmin "What-If" Index Analysis Utility. ACM SIGMOD 1998.
6. Chaudhuri S., Narasayya V., Automating Statistics Management for Query Optimizers. ICDE 2000.
7. Chaudhuri S., Narasayya V. Index Merging. ICDE 1999.
8. Finkelstein S, Schkolnick M, Tiberio P. Physical Database Design for Relational Databases, ACM TODS, Mar. 1988.
9. Gupta H., Selection of Views to Materialize in a Data Warehouse. ICDT, 1997.
10. Gupta H., Mumick I.S. Selection of Views to Materialize Under a Maintenance-Time Constraint. ICDT 1999.
11. Gupta H., Harinarayan V., Rajaramana A., Ullman J.D., Index Selection for OLAP, ICDE 1997.
12. Harinarayan V., Rajaramana A., Ullman J.D., Implementing Data Cubes Efficiently, ACM SIGMOD 1996.
13. Kotidis Y., Roussopoulos N. DynaMat: A Dynamic View Management System for Data Warehouses. ACM SIGMOD 1999.
14. http://www.oracle.com/
15. Rozen S. Automating Physical Database Design: An Extensible Approach, Ph.D. Dissertation. New York Univeristy, 1993.
16. http://www.informix.com/informix/solutions/dw/redbrick/vista/
17. Rozen S., Shasha D. A Framework for Automating Physical Database Design, VLDB 1991.
18. Shukla A., Deshpande P.M., Naughton J.F., Materialized View Selection for Multidimensional Datasets. VLDB 1998.
19. Shukla A., Deshpande P.M., Naughton J.F., Ramaswamy K., Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies. VLDB 1996.
20. TPC Benchmark H (Decision Support) Revision 1.1.0. http://www.tpc.org/
21. Valentin G., Zuliani M., Zilio D., Lohman G., Skelley A. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. ICDE 2000.
22. Yang J., Karlapalem K., Li Q., Algorithms For Materialized View Design in Data Warehousing Environment. VLDB 1997.