PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 19/06

Automated Selection of Materialized Views

José Maria Monteiro
Sérgio Lifschitz
Ângelo Brayner

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 19/06 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena July, 2006

Automated Selection of Materialized Views *

José Maria Monteiro, Sérgio Lifschitz, Ângelo Brayner[1]

[1]Mestrado em Informática Aplicada – Universidade de Fortaleza (UNIFOR)

monteiro@inf.puc-rio.br, sergio@inf.puc-rio.br, brayner@unifor.br

Abstract. Materialized views can provide massive improvements in query processing time, especially for aggregation queries over large tables. Due to the space constraint and maintenance cost constraint, the materialization of all views is not possible. Therefore, a subset of views needs to be selected to be materialized. The problem is NP-hard, therefore, exhaustive search is unfeasible. Further, a judicious choice must be cost-driven and influenced by the workload experienced by the system. For this reason, some past papers have looked at the problem of automated selection of materialized views for SQL workloads. Though, no prior work considers the problem of dynamic and automated creation (self-maintenance) of materialized views. This paper presents an architecture for completely automatic selection and creation (maintenance) of materialized views.

Keywords: Self-Tuning, Materialized Views, Automated Tuning, Heuristics, SQL Workload.

Resumo. A utilização de visões materializadas pode proporcionar expressivos ganhos de performance no processamento de consultas, especialmente em consultas agregadas sobre tabelas com grande volume de dados. Devido às restrições de espaço e custo de manutenção, a materialização de todas as visões torna-se inviável. Desta forma, um sub-conjunto das visões deve ser selecionado, a fim de que somente estas visões sejam materializadas. Entretanto, este é um problema NP-Difícil. Logo, a busca exaustiva torna-se uma solução inviável. Assim, uma escolha criteriosa deve basear-se no custo de execução das consultas e na carga de trabalho submetida ao sistema. Por esta razão, alguns trabalhos anteriores têm endereçado o problema da seleção automática de visões materializadas para uma determinada carga de trabalho. Porém, nenhum trabalho anterior considera o problema da seleção, criação e remoção de visões materializadas, de forma dinâmica e automática. Este trabalho apresenta uma arquitetura para automatizar completamente a seleção, criação e remoção de visões materializadas.

Palavras-chave: Ajustes Automáticos, Visões Materializadas, Heurísticas, Carga de Trabalho, SQL.

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

1 Introduction

Materialized views are derived relations, which are stored as relations in the database. Materialized views can be used for reducing the query response time. Using materialized views to speed up query processing is an old idea [9] but only in the last few years has the idea been adopted in commercial database systems. Recent TCP-R benchmark results and actual customer experiences show that query processing time can be improved by orders of magnitude through judicious use of materialized views. To realize the potential of materialized views, efficient solutions to three issues are required [5]:

- **View design**: determining what views to materialize, including how to store and index them.
- **View maintenance**: efficient updating of materialized views when base tables are updated.
- **View exploration**: making efficient use of materialized views to speed up query processing.

This paper deals with view design. As already mentioned, choosing an appropriate set of views to materialize in the database is crucial in order to obtain performance benefits. In this paper, we present an architecture and novel algorithms for dynamic and automated selection and creation (maintenance) of materialized views that can significantly improve query processing performance.

The rest of the paper is organized as follows. In section 2 an architecture for automated materialized view creation will be presented. Section 3 describes the strategy used for the candidate materialized view selection. In section 4 we discuss the algorithms and heuristics used for the final materialized view selection and creation. Section 5 concludes this work and outlines future works.

2 Architecture for Automated Materialized View Creation

Most prior work considers that an input, which is a workload for which a recommended set of materialized views is needed. Logically, a judicious selection is crucial to significantly improve performance. For this, to happen the choice must be influenced by the workload. We need to observe that, in the previous approaches, the materialized view selection heuristics typically execute in an off-line manner, they can also be executed in a distinct host of the DBMS Server machine. Our approach allows us to obtain the workload in an automatic and dynamic manner, allowing decisions about the view creation or exclusion to be made without human intervention.

An architecture overview of our approach to materialized view selection is shown in Figure 1.

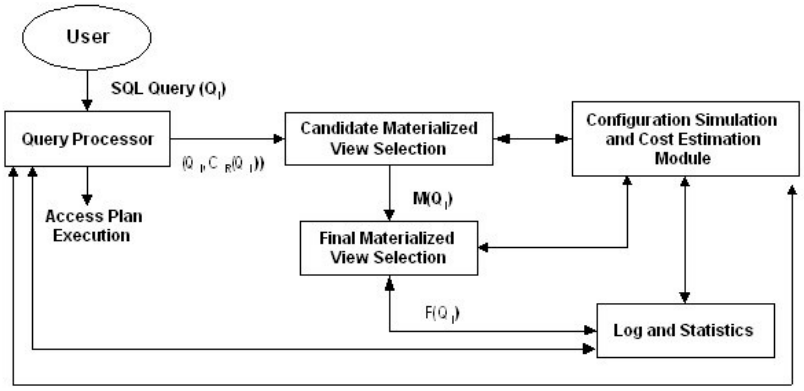

Figure 1. Architecture of Materialized View Selection.

Whenever a new operation (SQL command) is sent to DBMS, the optimizer generates the best access plan based on the real configuration. For this, the SQL optimizer uses the cost estimation module, logs and statistics. Then, the SQL optimizer informs the candidate materialized view selection module about the executed query and the selected access plan cost. Next, the candidate materialized view selection module, using a heuristics based on [1] will select a set of materialized views called candidate materialized views (syntactically relevant materialized views that can potentially be used to answer the query). As mentioned previously, searching the space of all syntactically relevant materialized views for a SQL query is unfeasible in practice, particularly when the selection heuristics run together with the other DBMS services. Therefore, is crucial to eliminate spurious materialized views from consideration early, thereby focusing the search on a smaller, and more interesting subset. The candidate selection module is responsible for identifying a set of materialized views for the given SQL query that are worthy of further exploration.

Once we have chosen a set of candidate materialized views, we need to search among these structures to determine which of them can effectively be used for the optimizer to improve performance on the query processing. The final materialized view selection module is responsible for this work. The algorithm used in this step is based on the benefit heuristics. This module tries to decide in an on-line manner, while the system receives new SQL commands, the materialized views that must be created or removed to improve the processing of the complete workload.

The final materialized view selection heuristic uses the configuration simulation and cost estimation module to determine the benefit (impact) of a candidate materialized view on the executions cost of the submitted query (and on the complete workload). For this, we need to extend the (PostgreSQL) query optimizer to simulate the presence of materialized views that do not exist (referred to as "what-if" materialized views), 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. We note that materialized view maintenance costs are accounted for in our approach by the inclusion of updates/inserts/deletes statements in the workload. Finally, the statistics are updated.

3 Candidate Materialized View Selection

Considering all syntactically relevant materialized views for a query (or workload) in the final materialized view selection phase is not scalable since it would greatly increase the search space. The space of syntactically relevant materialized views for a query is very large, since in principle, a materialized view can be proposed on any

subset of the table in the query. Furthermore, even for a given table-subset (a table-subset is a subset of tables referenced in a query, there is a great increase in the space of materialized views arising from selection conditions and grouped 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 the candidate materialized view selection phase is to eliminate materialized views that are syntactically relevant for one or more queries in the workload but are never used in answering any query.

Our approach for candidate materialized view selection is based on a similar strategy used by [1]. In this work the authors make some interesting observations. First, they observe that the obvious approach of selection of 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. Moreover, they also observered that 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 TCP-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
WHERE l_shipdate BETWEEN <Date1> and <Date2>
GROUP BY l_shipdate, l_returnflag, l_linestatus

A second observation that influences their 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 TCP-H 1 GB 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 nation and region tables 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 and the approach proposed in [1], we approach the task of candidate materialized view selection using three steps:

(1) From the large space of all possible table-subsets for a given query (input), we arrive at a smaller set of interesting table-subsets.

(2) Based on these interesting table-subsets, we propose a set of materialized views for the given query. This step uses a cost-based analysis for selecting the best materialized view for the given query.

(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. The new set of merged materialized views, along with the materialized views selected in (2) is the set of candidate materialized views that enters to the final materialized view selection phase. Now, we present the details of each of these steps.

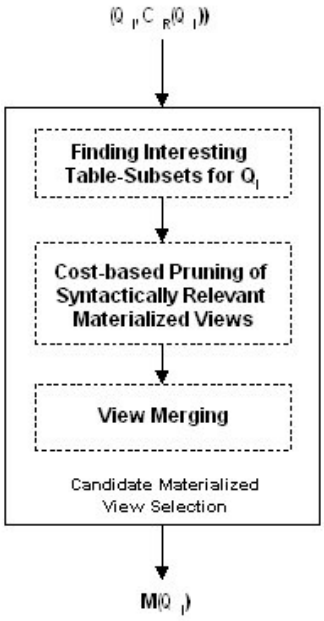

Figure 2. Candidate Materialized View Selection.

3.1. Finding Interesting Table-Subsets

Our goal is to find "interesting" table-subsets from among all possible table-subsets for a given query, and restrict the space of materialized views considered for 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 query (or workload) significantly. In [1] metrics are defined that capture the relative importance of a table-subset.

Consider the following metric: TS-Cost(T) = total cost of all queries in the workload (workload dynamically generates until this time) 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 TS-Cost(T) metric, the table-subsets T1 = {lineitem, orders} would have the same importance as the table-subset T2 = {nation, region} even though a materialized view on T1 is likely to be much more useful than a materialized view on T2. Therefore, [1] proposes the following metric that better captures the relative importance of a table-subset: TS-

Weight(T) = ∑I Cost(Qi) * (sum of sizes of tables in T)/(sum of sizes of all tables referenced in Qi)), 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.

1. Let S_1 = {T | T is a table-subset (of Q_i) of size 1 satisfying TS-Cost(T) ≥ C; i = 1
2. While i < TAM(Q_i) and |S_i| > 0
3. i = i + 1; S_i = { }
4. Let G = {T | T is a table-subset of size i, and ∃ s ∈ S_{i+1} such that s ⊂ T}
5. For each T ∈ G
6. If TS-Cost(T) ≥ C Then S_i = S_i ∪ {T}
7. End For
8. End While
9. S = S_1 ∪ S_2 ∪ … $S_{TAM(Qi)}$
10. R = {T | T ∈ S and TS-Weight(T) ≥ C}
11. Return R

Figure 3. Algorithm for finding interesting table-subsets for Q_i.

In figure 3, the size of a table-subset T is defined to be the number of tables in T. TAM(Q_i) is the number of tables referenced in a given query Q_i. A lower threshold C leads to a larger space being considered and vice versa. Based on experiments [1] 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.

3.2 Cost-based Pruning of Syntactically Relevant Materialized Views

The algorithm for identifying interesting table-subsets presented in Section 3.1 significantly reduces the number of syntactically relevant materialized views that must be considered for a query Q_i. 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 the final materialized view selection phase. We achieve this goal using the algorithm shown in Figure 4, which is based on the intuition that if a materialized view is not part of the best solution for even a single query, then it is unlikely to be part of the best solution for the entire workload. This approach is similar to the one used in [1]. For a given Query Q_i, and a set S_i of materialized views proposed for Qi, the algorithm assumes the existence of the function Find-Best-Configuration(Q_i, S_i) that returns the best configuration for Q_i from S_i. 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_i. Any suitable search method can be used in this function, e.g., the Greedy(m, k) algorithm described next. Also, a hypothetical query can be used.

```
1. Let S_i = Set of materialized views proposed for query Q_i
2. M = Find-Best-Configuration(Q_i, S_i)
3. Return M
```

Figure 4. Cost-based pruning of syntactically relevant materialized views for Q_i.

For each such interesting table-subset T, we propose (in step 1):

(1) A "pure-join" materialized view on T containing join and selection conditions in Qi on tables in T.

(2) If Qi has grouping columns, then a materialized view similar to (1) but also containing a GROUP BY columns and aggregate expression from Qi 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 3.3).

3.3 View Merging

In [1], the authors observe that we need to consider the space of materialized views which although they are not optimal for any individual query, are useful for multiple queries, and therefore may be optimal for the workload. Also, the approach proposed in [1] notes that the set M (the set of materialized views returned by the algorithm in Figure 4) 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 the candidate materialized views.

The goal for merging 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. We use the algorithm proposed in [1] for merging a pair of views, called MergeViewPair (Figure 5). Intuitively, the algorithm achieves the first property by structurally modifying the parent views as little as possible when generating the merged view.

Note that a merged view v may be derived starting from views in M through a sequence of pair-wise merges.

Parent-Closure(v) is 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). The factor x that determine the value of the size increase threshold(x) was typically a setting between 1 and 2 [1].

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.

We use the algorithm proposed in [1] (Figure 6) for generating a set of merged views from a given set of views M (obtained using the algorithm in Figure 4).

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 $((|v_{12}| > \text{Min Size}(\text{Parent-Closure}(v_1) \cup \text{Parent-Closure}(v_2))*x)$ Then Return Null.
5. Return v_{12}

Figure 5. MergeViewPair Algorithm.

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

Figure 6. Algorithm for generating a set of merged views from a given set of views M.

4 Final Materialized View Selection

Once we have chosen a set of candidate materialized views, we need to search among these structures to determine which of them can effectively be used for the optimizer to improve performance on the query processing. The final materialized view selection module is responsible for this work. The algorithm used in this step is based on the benefit heuristics. This module tries to decide in an on-line manner, the materialized views that must be created or removed to improve the processing of the complete workload, while the system receives new SQL commands.

Initially, the materialized view selection module receives with input the executed SQL clause (Q_i), the best access plan costs based on the real configuration ($C_r(Q_i)$) and the set of candidate materialized view ($M(Q_i)$). Next, this module obtains, together with the configuration simulation and cost estimation module (or optimizer), some additional information, such with, the best access plan costs in face of total absence of materialized views. After this, if the SQL clause (Q_i) is a query then the query evaluation heuristic is executed, in other case the update evaluation heuristic is executed. Finally, the statistics are updated.

These heuristics use the configuration simulation and cost estimation module (or optimizer) to determine the benefit (impact) of a candidate materialized view on the executions cost of the submitted query (and on the complete workload). For this, we need extend the (PostgreSQL) query optimizer to simulate the presence of materialized views that not exist (referred to as "what-if", or hypothetical, materialized views), 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.

Before we present the query evaluation heuristic and the update evaluation heuristic, we need define the following factors:

(1) C_R: the best access plan costs (generated by the optimizer) in face of the real materialized view configuration.
(2) C_H: the best access plan costs (generated by the optimizer) in face of the real and hypothetical materialized view configuration. Usually, the hypothetical materialized views used in this plan corresponds to the candidate materialized views (obtained in the previous phase).
(3) C_N: the best access plan costs (generated by the optimizer) in face of the absence of real and hypothetical materialized views.
(4) C_A: the estimated cost for update an materialized view during the processing of a SQL clause update (update/insert/delete) on a base relation. Usually, this cost doesn't be supplied by the optimizer and must be estimated in agreement of the SGBD cost model.
(5) B_K: The benefit of the materialized view k for the current SQL clause.
(6) BAc_K: The accumulated benefit of the materialized view k for the set of all processed (executed) SQL clause (dynamic workload).
(7) CC_K: The creation materialized view estimated cost. Usually, this costs doesn't be calculated by the SGBD optimizer. Thereby, this cost will be estimate by by our implementation.

The following, is a description of the query evaluation heuristic. First, for each candidate materialized view (CMV), if it is used in the best access plan (generated by the optimizer) based on the real and hypothetical materialized view configuration, we will compute its benefit, B_{CMV}, (for the current SQL query) and its accumulated benefit, BA_{CMV}. The benefit is the difference between C_R and C_H. The candidate materialized view will be materialized only if its accumulated benefit is greater than the creation materialized view estimated cost (CC_{CMV}). The goal is to create materialized views whose repetitive use is interesting enough to compensate its creation cost. Note that there is an implicit supposition in this heuristic: materialized views used in previous SQL queries will continue to be interesting for future SQL queries. Thus, this heuristic tends to find goods solutions for reasonably stable workloads. Next, for each real materialized view (RMV), if it is used in the best access plan (generated by the optimizer) based on the real materialized view configuration, we will compute its benefit, B_{RMV}, (for the current SQL query) and its accumulated benefit, BA_{RMV}. The benefit is the difference between C_N and C_R.

1 . For each candidate materialized view CMV used in the best access plan (with real and candidate materialized views) do:
2. $B_{CMV} = C_R - C_H$
3. $BAc_{CMV} = BAc_{CMV} + B_{CMV}$
4. If $BAc_{CMV} > CC_{CMV}$ Then
5. Create materialized view CMV
6. $BAc_{CMV} = 0$
7. End If
8. End For
9 . For each real materialized view RMV used in the best access plan (with real materialized views) do:
10. $B_{RMV} = C_N - C_R$
11. $BAc_{RMV} = BAc_{RMV} + B_{RMV}$
12. End For

Figure 7. Algorithm for query evaluation.

The following is a description of the update evaluation heuristic. This proceeding initially executes the query evaluation heuristic. After this, the materialized views maintenance costs needed to process the (update) SQL clause are computed. These costs will be used to update the accumulated benefit of real and candidate materialized views. Next, we verify if the real materialized views continue being gainful. For each updated materialized view, we verify if its benefit is negative and greater than, in module, its creation materialized view estimated cost (CC_{RMV}). In this case, the materialized view is removed. Note that again there is an implicit supposition in this heuristic: materialized views harmful in previous SQL queries will continue being harmful for future SQL queries. Thereby, the view maintenance costs are accounted for in our approach by the inclusion of updates/inserts/deletes statements in the workload.

1. Execute algorithm for query evaluation
2. For each materialized view MV affected by the update SQL clause do:
3. $BAc_{MV} = BAc_{MV} - C_A$
4. If (MV is real) and ($BAc_{MV} < 0$) and ($|BAc_{MV}| > CC_{MV}$) Then
5. $BAc_{MV} = 0$
6. Remove materialized view MV
7. End If
8. End For

Figure 8. Algorithm for update evaluation.

The benefit heuristic is very interesting because its on-line manner, that allow the dynamic creation and removal of materialized views, without human interference.

5 Conclusions and Future Works

Materialized views can provide massive improvements in query processing time. Due to the space constraint and maintenance cost constraint, the materialization of all views is not possible. Therefore, a subset of views needs to be selected to be materialized. In this paper, we present an architecture and novel algorithms for dynamic and completely automated selection and creation (maintenance) of materialized views that can significantly improve query processing performance, without human interference. Also, our approach makes a judicious choice, which is cost-driven and influenced by the workload experienced by the system. As future work we will investigate the use and performance of our approach for automated selection and creation of materialized view together with the approach proposed in [6] for automated index selection and creation.

References

1. Agrawal, S., Chaudhuri, S., Narasayya, V.: Automated Selection of Materialized Views and Indexes for SQL Databases. In Proceedings of VLDB, pages 496--505, 2000.
2. Chaudhuri, S., Narasayya, V.: An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server. In Proceedings of VLDB, pages 146-155, 1997.
3. Chirkova, R., Li, C.: Materializing Views with Minimal Size To Answer Queries. In Proceedings of PODS, pages 38--48, 2003.
4. Valluri, S.R., Vadapalli, S., Karlapalem, K.: View Relevance Driven Materialized View Selection in Data Warehousing Environment. In Proceedings of the Australasian Database Conference, 2002.
5. Goldstein, J., Larson P.: Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. . In Proceedings of the ACM SIGMOD, 2001.
6. Salles, M. A. V.: Automated Creation of Indexes for Databases. Master's Thesis, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), July, 2004 (in Portuguese).
7. Lifschitz, S., Milanês, A. Y., Salles, M. A. V.: Estado da arte em auto-sintonia de sistemas de bancos de dados relacionais. Technical report, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), 2004 (in Portuguese).
8. Costa, R. L. C., Lifschitz, S.: Index self-tuning and agent-based databases. In Proceedings of the Latin-American Conference on Informatics (CLEI), 2002.
9. Larson, P. A., Yang, H. Z.: Computing Queries from Derived Relations. In Proceedings of VLDB, pages 259-269, 1985.
10. Skelley, A. DB2 Advisor: An Optimizer Smart Enough to Recommend its own Indexes. Proceedings of the 16th International Conference on Data Engineering, 2000.