Journal of Computer and Communications, 2016, 4, 47-53
Published Online May 2016 in SciRes. http://www.scirp.org/journal/jcc
http://dx.doi.org/10.4236/jcc.2016.45007

Scientific Research Publishing

Automated Performance Tuning of Data Management Systems with Materializations and Indices

Nan N. Noon, Janusz R. Getta

School of Computer Science and Software Engineering, University of Wollongong, Wollongong, Australia
Email: nnn326@uowmail.edu.au, jrg@uow.edu.au

Received 21 April 2016; accepted 19 May 2016; published 26 May 2016

Abstract

Automated performance tuning of data management systems offer various benefits such as improved performance, declined administration costs, and reduced workloads to database administrators (DBAs). Currently, DBAs tune the performance of database systems with a little help from the database servers. In this paper, we propose a new technique for automated performance tuning of data management systems. Firstly, we show how to use the periods of low workload time for performance improvements in the periods of high workload time. We demonstrate that extensions of a database system with materialised views and indices when a workload is low may contribute to better performance for a successive period of high workload. The paper proposes several online algorithms for continuous processing of estimated database workloads and for the discovery of the best plan for materialised view and index database extensions and of elimination of the extensions that are no longer needed. We present the results of experiments that show how the proposed automated performance tuning technique improves the overall performance of a data management system.

Keywords

Automated Performance Tuning, Query Processing, Materialization, Indexing, Data Management Systems

1. Introduction

Database management systems used by the business organisations need expert database administrators (DBAs) to configure the systems before a startup time and to tune performance while the systems are running. Currently, to achieve acceptable performance of a database system an administrator has to tune the system by herself/himself with her/his knowledge of the anticipated workload [1]. A sudden increase in the total number of users accessing a database system during peak hours may cause the significant delays, and it may even block entire system. One of the solutions to increase efficiency and reliability of database systems is *automated* or *self-performance tuning* [2].

How to cite this paper: Noon, N.N. and Getta, J.R. (2016) Automated Performance Tuning of Data Management Systems with Materializations and Indices. *Journal of Computer and Communications*, **4**, 47-53.
http://dx.doi.org/10.4236/jcc.2016.45007

Self-tuning database management systems automatically create and drop persistent storage structures like indices and materialised views and dynamically re-allocate transient storage resources such as data buffer cache, library cache, and so on. Materialzied views, indices, and better management of cache in transient storage can speed up a database system to reduce response time.

Automated performance tuning of database systems is a challenging problem. At the periods of high workload, the query optimizer has to assign different schedules to execute the query statements that contribute to delays and high processing costs [3]. Ad hoc optimisation of query processing with materializations and indexing would reduce those costs.

Materialized views are created as precomputed joins, stored aggregated data, stored summarised data and so on. Materializations are less expensive for joins and aggregations for processing of complex queries of high importance. In-addition, materializations can also increase the speed of query processing in large database systems which include expensive operations such a complex aggregation with joins. Moreover, it improves the performance of query processing by pre-calculating expensive operations on the database before execution and sorting the results in the database. Appropriate use od the materialized view and indices allows for optimal adjustment of the size of data sets to a given collection of queries through partitioning and creating indices on the database. For example, a materialised view restricts relational tables vertically only to the columns needed by a query. An index restricts relational tables horizontally only to the rows that satisfy the conditions used in a query.

The main objective of this research is to invent the algorithms to generate performance tuning task that reduces high workload and arrange the job within limited time and limited I/O cost level (limited workload). The algorithm uses the predicted query workload [4] of the database system. The predicted workload is used to prepare a database system for better performance during future data processing. First, a database is extended with materializations discovered from the analysis queries. Next, the algorithms generate a set of operations and create indices on extended database system from the previous step.

We present the algorithms, which are used to reduce a high workload for a given set of queries. The given workload can be a high or low workload. To reduce the high workload, we choose the best preparation plan before that high workload occurs. Besides, the best plan cannot be greater than the limited workload, and it also execute within the short time. **Figure 1** shows the high and low workload structure with limited time and workload level. The limited time t and workload level l located at the period of low-level workload. Our preparation plans will generate at the duration of the low-level workload with the limited t and l to reduce heavy workload.

In this paper, we present two group of algorithms for automated tuning of database management systems. In the first step, we accept a set of queries which suppose to be processed in high workload time. Then, we find the minimized projections for each query and store the projections as the schemas of materialized views. After that, we minimize a set of schemas of materialized views and arrange them from high to low priority. Next, we check whether the selected schemas can execute within the limited workload time or not. Finally, we generate the materialized views. In the second step, we analyse the queries and extract the operations from extended relational algebra expressions. Then, we eliminate the duplicated operations and arrange them by priority level of cost multiply by frequencies of the operations. Next, we estimate the cost of indices and check whether the cost fits into limited workload time. Finally, we create indices on materialized views.

The major contributions of our work are the following.

- We show how to discover the schemas needed to execute the queries and eliminate the duplicated one from given collection of queries $q_1, \cdots, q_n$ for extended database system with materialized view.
- We show how to discover common operations from database extended with materialized view.
- We show how to apply indexing technique on database extended with materialized view.

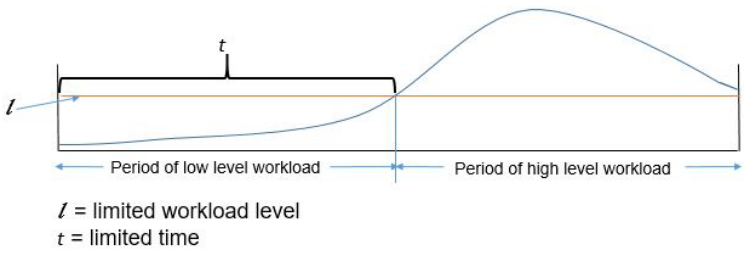

Figure 1. Sample circle for low and high workload.

- We show how to generate tuning scripts (preparation scripts) within the limited time and workload level.

The paper consists of 6 sections. The previous works in related research areas will discuss in the next section. Sections 3 and 4 explain the algorithms. Experimental results are presented and discussed in Section 5. Finally, Section 6 concludes the paper.

2. Related Work

The manual processes are needed to achieve high-performance of the database system. It is an enormous effort and the complicated job which database management administrators (DBAs) or users are done [5]. There are too many techniques to tune the database system which is proposed by [6]. They discussed papers written over the past decade, which are related to self-tuning database systems. At first, they described papers that are using indexes [7] to tune the database. Next, they discussed materialized view and partitioning, which is another technique to tune the database system. Finally, they listed commercial DBMS tools for tuning database system and analysis the pros and cons of those tools.

There are some commercial databases tools like Oracle Database 11 g [3] providing self-tuning tools to tuning database system automatically. This tool accepts a set of SQL statements, then generates execution plans and allows users to choose the best plan. Once users have accepted the recommended plans, the performance tunning progress will complete and execute the tuning scripts.

Moreover, the authors from [8] show that how to self-tuning for the Database system and proposed Fuzzy-based tuning. They proposed buffer cache size (BCS), Buffer-Hit-Ration (BHR), and Database size (DBSize) to show how to self-tuning the database system. Their methods might have the problem of limited buffer size.

Furthermore, there are many research works proposed indexing technique to tune the database system. Among them, the authors from [9] proposed two automatic on-line index selection algorithms and one semi-automatic on-line index algorithm. They proposed both automatic and semi-automatic tuning by using best selection of index technique. The semi-automatic on-line index provides the list of the index (recommends the set of indexes) and allows DBA to select to create or drop the index.

Materialized view is one of the best techniques for tuning database. In this book [10] shows how materialized views can speed up queries especially for table joins and the use of aggregates like SUM. Materialized view can reduce the queries cost because it has already had the summary information pre-computed in them. The queries cost means I/O, CPU, and memory costs which are related to processing queries.

Above researchers focused on how to achieve the best performance for given the set of queries, and they show the results that reduced cost and time for given the set of queries. Although, they did not describe the consuming cost and time to execute for tuning scripts or tuning processes. In our research, we discuss how tuning scripts occupy within limited time and cost before high workload occurs. Furthermore, we discuss how workload reduces by using our new techniques.

3. Materializations

This section presents the algorithms that find the schemas of materialized views to be created in a period of low workload time. The algorithms maximise the performance improvements during a high workload period and minimise the costs of materialised views creation at a low workload period.

Algorithm 1 takes on input a set of queries $\{q_1, \cdots, q_n\}$ predicted for the nearest period of high workload. The output of the algorithm is a multiset of schemas of materialized views $V = \{v_1, \cdots, v_m\}$ that improve performance of query processing. Each v_i V is a schema of materialized view such as $v_i = r_i[c_{i1}, \cdots, c_{ik}]$ where r_i is a relational table processed by one of the queries $q_j\{q_1, \cdots, q_n\}$ and $\{c_{i1}, \cdots, c_{in}\}$ is the smallest set of columns from r_i needed to process a query q_j.

Algorithm 1. performs the following actions.

1. Make a multiset of schemas of materialized views V empty.
2. Iterate from q_1 to q_n. Let current query be q_i. Let $r_{i1}, \cdots, r_{in}$ be the relational tables processed by q_i.
   2.1. Apply EXPLAIN PLAN statement to q_i to get query processing plans. Then, for each relational table r_i processed in q_i find its smallest projection $ri[c_{i1}, \cdots, c_{ik}]$ needed to process a query q_i.
   2.2. Append each projection $r_i[c_{i1}, \cdots, c_{ik}]$ found to a multiset of schemas of materializations V.
3. Repeat until all the queries are processed.

As a simple example, we consider the queries q_1: SELECT a FROM r WHERE b > 10; and q_2: SELECT s.a, t.b FROM s JOIN t ON s.a = t.b; and q_3: SELECT r.a, s.a FROM r JOIN s ON r.a = s.a. The output from **Algorithm 1** is a multiset $V = \{r[a, b], r[a], s[a], t[b], s[a]\}$.

Algorithm 2 takes on input a multiset of schemas of materialized views V. The algorithm replaces each schema v_i V with a pair v_i:f_i where f_i is a counter how many times the respective materialised view will be used at high workload time when processing the queries $\{q_1, \cdots, q_n\}$ and it eliminates the schemas that do not significantly improve performance and the schemas which are to expensive to be created.

Algorithm 2 performs the following actions.

1. Replace each v_i V with a pair v_i:0.
2. Iterate from v_1 to v_n in V. Let current schema of materialised view be $v_i = r_i[X_i]$ where X_i is a set of columns in r_i.
   2.1. Iterate from v_1 to v_n in V - $\{v_i\}$. Let current schema of materialised view be $v_j := r_j[X_j]$ where X_i is a set of columns in r_j.
      2.1.1. If $X_i = X_j$ then increase a frequency counter f_i in v_i:f_i by one and eliminate the duplicated schema v_j.
      2.1.2. If v_i v_j or vi v_j, then estimate the costs $cost(v_i)$ and $cost(v_j)$. Let a limit workload level for materialised views be l_v.
         2.1.2.1. If $l_v > (cost(v_i) + cost(v_j))$ then we calculate the profit cost by computing as $p = |cost(v_i) - cost(v_j)|$ where p is profit.
         2.1.2.2. If p $(cost(v_i) + cost(v_j)) * 0.5)$, *i.e.* profit must be greater than 50% of total cost for both schemas then we take both schemas of materialized views (no elimination). Else if cost(v_i) > $cost(v_j)$ then eliminate the v_j and extend frequency f_i.

Assuming that $cost(r[a])$ in the previous example is almost the same as $cost(r[a, b])$ then **Algorithm 2** applied to a multiset $V = \{r[a, b], r[a], s[a], t[b], s[a]\}$ returns a set $V' = \{r[a, b]:2, s[a]:2, t[b]:1\}$.

Algorithm 3 takes on input a set of schemas of materialized views V' created by **Algorithm 2**. The algorithm replaces each schema v_i:f_i V with v_i:f_i:c_i where c_i is an estimated cost of creation of materialized view v_i. Then, it allocates schemas of materialized views $V'' = \{v_1, \cdots, v_n\}$ within the limited workload l_v and time t_v for materialized views and execute them. Finally, Algorithm 3 verifies whether the queries $q_1, \cdots, q_n$ benefit from the existence of the views in V''.

Algorithm 3 performs the following actions.

1. Replace each v_i:f_i V with v_i:f_i:0.
2. Iterate from v_1 to v_n in V. Let current schema of materialised view be $v_i = r_i[X_i]$:f_i:c_i where X_i is schema of materialized view in r_i.
   2.1. Estimate the cost of materialized view c_i and add such value to v_i:f_i:c_i.
3. Sort the v_i:f_i:c_i in descending order of c_i*f_i and update the V.
4. Iterate from v_1 to v_n in V. Let current schema of materialized view be $v_i = r_i[X_i]$:f_i:c_i. Let estimated creation time for materialized view be i_v, limited workload level be l_v, and limited time be t_v.
   4.1. If $l_v > c_i$ and $t_v > i_v$ then we create a materialized view v_i and remove v_i from V'. Next, update the $l_v = l_v - c_i$.
   4.2. Iterate until all schemas are allocated into limited workload.
5. In the final step, EXPLAIN PLAN statement is used to find the query processing plans for $q_1, \cdots, q_n$ and to verify whether all materialized views created in the previous step are used by the queries.

4. Indexing

This section presents the algorithms that find the best index to be created in a low workload time. The algorithms improve performance in a database system and reduce the high workload period.

Algorithm 4 processes a set of queries $\{q_1, \cdots, q_n\}$. Then, we transform operations $_1, \cdots, _n$ into a sequence of sets of statements $S = <S_i, \cdots, S_n>$. Each statement in a set S_i, for $I = 1, \cdots, n$ takes a form $s := (x\ y)$ where x, y are the arguments of the operation, and s is a result of operation $(x\ y)$. The arguments x and y can be the database

relational tables or the results of operations which computed earlier.

Algorithm 4 performs the following actions.
1. Let a sequence of sets of statements S be empty.
2. Formulate set of queries $\{q_1, \cdots, q_n\}$ as expressions $<e_1, \cdots, e_n>$ of an extended relational algebra.
3. Iterate from e_1 to e_n and reduce expressions into a single name of the temporary result then stop the iteration.
   3.1. Let the new current set of statements be S_i.
   3.2. Find all operations like $(x\ y)$ in the expressions $e_1, \cdots, e_n$.
   3.3. Take each operation from the previous step and transforms into a form like $s_{ij} := (x\ y)$ and added into the current set of statements S_i.
   3.4. Get all operations like $(x\ y)$ from expressions $e_i, \cdots, e_n$ and store into temporary result like s_{ij}. Then append into current set Si.

As a simple example, consider the queries are q_1, q_2 and q_3. Their processing plans expressed as the expressions of extended relational algebras are q_1:$(a_1 r)$, q_2:$(a_1(s_2 t))$, q_3:$(b_1(s_2 t))$ where $_1$ and $_2$ are the operations and a, b, r, t, and s are the relational tables or the results of operations computed earlier. The transformation results of sets of statements are $<\{s_{11} := (a_1 r), s_{12} := (s_2 t), s_{13} := (s_2 t)\}, \{s_{21} := (a_1 s_{12}), s_{22} := (b_1 s_{13})\}>$.

Algorithm 5 processes a sequence of sets of statements S. The algorithm appends each operation with a frequency like $(x_i y)$:f_i where fi is a counter of how many time appear $_i$ in S. Then algorithm finds the common operations and eliminates one from a sequence of sets of statements and increase the frequency.

Algorithm 5 performs the following actions.
1. Make all frequency be 0.
2. Iterate over the sets of statements $<S_1, \cdots, S_p>$ in S and let the current set statements be S_i.
   2.1. For each statements from S_i and let current statement be $s_{ii} := (X_i)$:f_i where X_i is $(x_i y)$.
      2.1.1. For each statemets from $S_i - \{s_{ii}\}$ and let current statement be $s_{ij} := (X_j)$:f_j.
         2.1.1.1. If $X_i = X_j$ then eliminate s_{ij} and incease the counter $f_i = f_i + 1$.
         2.1.1.2. Get eliminated statement s_{ij} and replace all s_{ij} with s_{ii} in S.

According to **Algorithm 5**, we found that s_{12} and s_{13} are the same and eliminate the s_{12} then, extend the frequency by one. After that, algorithm replace s_{13} with s_{12} in S. Then, algorithm return the $S = <\{s_{11} := (a_1 r)$:1, $s_{12} := (s_2 t)$:2$\}, \{s_{21} := (a_1 s_{12}), s_{22} := (b_1 s_{12})\}>$.

Algorithm 6 processes updated a sequence of sets of statements S. The algorithm changes each statement s_{ii}:f_i S_i with s_{ii}:f_i:o_i where o_i is estimated cost for index of operation $_i$. Then, it allocates the indexes if they can fit into low workload l_i and time t_i.

Algorithm 6 performs the following actions.
1. Replace each s_{ii}:f_i with s_{ii}:f_i:0 in S.
2. Iterate over the sequence of sequences $<S_1, \cdots, S_n>$ in S. Let current set of statements be S_i. Let estimated creation time for index be i_i, limited workload level be l_i and limited time be t_i.
   2.1. For each statement in S_i like $s_{ii} := (x_i y)$:f_i:o_i use a specification $(x_i y)$ and the estimated cost of index o_i for operation $_i$ then, add such value to s_{ii}:f_i:o_i.
   2.2. Sort the s_{ii}:f_i:o_i in descending order of $o_i * f_i$ and update the S_i.
   2.3. Iterate each statement in S_i. Let current statement be $s_{ii} := (x_i y)$:f_i:o_i.
      2.3.1. The algorithm discovers the indexing by searching the type of operations like SELECTION, PROJECTION, NATURAL JOIN, SEMIJOIN, and so on. If $_i$ is not projection then processes below because we no need to create index on projections. If $l_i > o_i$ and $t_i > i_i$ then create the index. Then update the $l_i = l_i - o_i$.

5. Experiment

In our experiments, we show that our results are better than original respond time and cost. We used a synthetic

TPC-H 4 GB benchmark relational database [11] which ready built relational tables by the user applications. We used commercial database software which includes procedure and packages to estimate the size of materialized view to compute the cost and virtual index to estimate the complexity of the operation for indexes. As a high workload, we choose eight complex queries from TPC's Template Set.

We start our experiment by analyzing the queries and getting the schemas for materialized views which are necessary for each query. Then, we remove the duplicated set of schemas and compute the estimated cost and time for materialized view. Next, we generate the materialized view if they fit into limited workload time l_m. After that, we verify weather all created materialized views are used by queries or not. The second part of our experiment analyzing the queries. Then, we get the sequence of the set of statements of operations. Next, we remove the duplicated operations and append the frequencies. After that, we create indexes when they can fit into limited workload time l_i.

We did our experiments for several times to get the reliable results. Our experiments show that our algorithms achieve the better result than the original query. **Table 1** shows that average percentages profit results of our algorithm is the best by comparing with the original one. The profits we achieve for executions times are range between 20% - 65% and the costs are range between 66% - 89%.

6. Summary and Conclusions

In this paper, we present the algorithms for automated Performance Tuning of Database System by using Materialized View and Indexing. The input is a set of low and high workloads of queries. The algorithm focuses on reducing high workload. To reduce the high workload, we execute tuning script (preparation stage) on low workload time. There are two main steps in this paper. One is how to create materialized views within the limited workload l_v and time t_v. The other is how to execute indexes on materialized views within the limited workload l_i and time t_i.

In the first stage, we analysis the queries and extract schemas of materialized views then, remove the duplicated schemas. Next, we execute the materialized view when it is fitted into limited workload time l_v. The materialized view can extract the database schemas into smaller pieces of database schemas. It can store summarized data, precomputed joins with or without aggregations. Besides, it is suitable for large or important queries because it can eliminate the overhead associated with expensive joins and aggregations. Furthermore, it allows us to create indexes with minimal creation time and cost.

Table 1. Compare original queries execution time and cost with tuning queries execution time and cost: experiemnt results for q1-q8.

Name	Description	Execution Time	I/O Cost	Name	Description	Execution Time	I/O Cost
	Original	00:24.00	84,436		Original	00:38.10	82,436
q_1	Tuning	00:07.70	8,835	q_2	Tuning	00:16.60	9,330
	Profit%	**63.89%**	**89.28%**		**Profit%**	**56.54%**	**88.68%**
	Original	00:34.30	124,928		Original	01:47.90	158,720
q_2	Tuning	00:11.40	41,587	q_4	Tuning	00:40.10	41,597
	Profit%	**66.81%**	**66.71%**		**Profit%**	**44.92%**	**73.79%**
	Original	01:06.30	131,072		Original	02:03.60	159,744
q_5	Tuning	00:44.90	41,594	q_6	Tuning	01:06.90	41,591
	Profit%	**32.21%**	**68.27%**		**Profit%**	**45.83%**	**73.96%**
	Original	01:32.70	155,648		Original	01:15.20	135,168
q_7	Tuning	00:51.50	41,602	q_8	Tuning	00:59.40	41,596
	Profit%	**44.31%**	**73.27%**		**Profit%**	**20.71%**	**69.23%**

In the second stage, we extract the operations from queries. Then, we eliminate duplicated operations and create the index over materialized view they are fitted on limited workload time l_i. There are different ways to choose the best index and generally, an index created based on WHERE clause. In this paper, we make a decision based on operations like SELECTION, PROJECTION, NATURAL JOIN, SEMIJOIN, and so on. Finally, we set up our experiments base on our algorithms, and the results show that our methods perform better than original execution time and cost.

References

[1] Weikum, G., Moenkeberg, A., Hasse, C. and Zabback, P. (2002) Self-Tuning Database Technology and Information Services: From Wishful Thinking to Viable Engeering. *Proceedings of the* 28*th International Conference on Very Large DataBase*, 20-31. http://dx.doi.org/10.1016/B978-155860869-6/50011-1

[2] Almeida, A.C., Lifschitz, S. and Breithman, K. (2009) A Knowledge Representation and Data Provenance Model to Self-Tuning Database Systems. 3*rd Annual IEEE Software Engeering Workshop* (*SEW*), 144-150. http://dx.doi.org/10.1109/sew.2009.25

[3] Belknap, P., Dageville, B., Dias, K. and Yagoub, K. (2009) Self-Tuning for SQL Performance in Oracle Database 11g. *Data Engineering ICDE*'09. *IEEE* 25*th International Conference*, 1694-1700. http://dx.doi.org/10.1109/icde.2009.165

[4] Marcin, Z., Janusz, R.G. and Wolfgang, B. (2014) Deriving Composite Periodic Patterns from Database Audit Trails. Intelligent Information and Database Systems, Spring International Publishing, 310-321.

[5] Cong, G., Chung, I.H., Wen, H.F., Klepacki, D., Murata, H., Negishi, Y. and Moriyama, T. (2012) A Systematic Approach toward Automated Performance Analysis and Tuning. *IEEE Transactions on Parallel and Distributed Systems*, **23**, 426-435. http://dx.doi.org/10.1109/TPDS.2011.189

[6] Surajit, C. and Vivek, N. (2007) Self-Tuning Database System: A Decade of Progress. Very Large Database Endowment Inc. (VLDB).

[7] Valentin, G., Zuliani, M., Zilio, D.C., Lohman, G. and Skelley, A. (2000) Db2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. *Data Engineering, Proceedings*, 16*th International Conference*, 101-110. http://dx.doi.org/10.1109/icde.2000.839397

[8] Rodd, S. and Kulkarni, U. (2013) Adaptive Self-Tuning Techniques for Performance Tuning of Database Systems: A Fuzzy-Based Appraoch. *Advanced Computing, Networking and Security* (*ADCONS*), 2013 2*nd International Conference*, *IEEE*, 124-129. http://dx.doi.org/10.1109/ADCONS.2013.49

[9] Schnaitter, K. (2010) On-Line Index Selection for Physical Database Tuning. Ph.D. Thesis, Santa.

[10] Alapati, S.R. (2008) Expert Oracle Database 11g Administration. Apress, Berkely.

[11] TPC (2015) http://www.tpc.org/information/benchmarks.asp