Revista de Sistemas de Informacao da FSMA n. 5 (2010) pp. 14-21 http://www.fsma.edu.br/si/sistemas.html

A framework for database optimization and workload control

Antonio Carlos Alves Junior , Fernando Dias Scarabelin, Luciano Antonio Digiampietri
Escola de Artes, Ciencias e Humanidades da USP
Av. Arlindo Bettio, 1000 – 03828-000 – Sao Paulo – SP
digiampietri@usp.br

***Abstract*—Nowadays, database optimizers take advantage of indexes and materialized views (MVs) to produce query execution plans. While indexes and MVs can speed up the execution of queries, there are costs to store and manage them. This paper presents a mathematical model and a greedy implementation to recommend a set of indexes and MVs in order to optimize the database, given a database workload and a disk space constraint. Our approach is generic, so it can be used to any relational database system that takes advantage of MVs in plan selection. While it was developed for frequently updated databases, it also produced interesting results in read only databases and to estimate the behavior of new databases (with predefined workloads).**

***Index Terms*—Database optimization, materialized view, index, query execution plan.**

I. Introduction

In the last decades, there was a growing of the needs of Database Management Systems (DBMSs) to manage the great amount of digital data produced every day. Several researchers worked on the development of new optimization mechanisms, in order to speed up queries over these databases.

While some approaches re-designed the databases to store information more efficiently [Ma et al. 2007], other developed new heuristic algorithms to produce better query execution plans. In the last ten years, several commercial databases started to take the advantage of the Materialized Views (MVs) in the selection of the execution plan (instead of use only regular tables and indexes information). This approach increases the reuse of pre-processed queries (the ones that generated the MVs) to optimize the speed of a query.

One new trend in database systems is how to select a set of indexes and materialized views to optimize the query processing cost. The selection of MVs to work in data warehouses was explored for several researches and have some interesting solutions for read only databases [Harizopoulos et al. 2006] (databases that are not updated or are updated only rarely).

The new challenge is the development of an optimization system that, given the profile of use (workload) for a given read and write database, recommends the best set of indexes and MVs for helping in the plan selection, regarding some restrictions. This issue is particularly interesting because the end user (that already works with some DBMS) does not have to install a new DBMS, he/she only has to create the new indexes and MVs in order to optimize the use of the DBMS.

The selection of indexes and MVs is a complex task because it must consider the disk space constraint (each index and MV represents redundant information stored in the database) and the update/management costs for the new indexes and MVs (whenever a table is updated some indexes and MVs must be updated to keep the database consistency). Thus, a sophisticated mechanism to evaluate the costs and benefits must be developed.

This paper presents a generic optimization mathematical model for the selection of indexes and MVs for read and write databases (databases that are frequently updated). Since the evaluation of all combinations of indexes and MVs is an intractable problem, we developed a simplification of this model that uses a greedy algorithm that produced good results in satisfactory time.

The rest of this paper is organized as follows. Section II defines some basic concepts that are need to the understanding of the paper. Section III presents the mathematical model for selection of indexes and materialized views. Section IV contains the greedy algorithm implemented to recommend the indexes and MVs. Section V discusses some preliminary results in a real use case. Section VI presents the related work and Section VII contains the conclusions and future steps.

II. Basic Concepts

This section presents some database concepts [Loney and McClain 2004], [Lightstone et al. 2007], [Chaudhuri 1998], [Connolly and Begg 2004] that will be used in this paper.

a) Database Indexing.: All of the modern DBMSs can take advantage of indexing to speed up their processing. The most simple kind of index is a sorted list of the contents of some particular table column, with pointers to the row associated with the column value. An index allows that a set of table rows matching some criterion to be located quickly. Typically, indexes are also stored in common data-structure (such as trees, hashes, and linked lists). Usually, a specific technique is chosen by the database designer to increase efficiency in the particular case of the type of index required.

Relational Database Management Systems (relational DBMSs) have the advantage that indexes can be created or dropped without changing existing applications making use of it. The database chooses among many different strategies based on which one it estimates will run the fastest.

An index speeds up access to data, but it has disadvantages as well. First, every index increases the amount of storage on the hard disk, and second, the index must be updated each time the data is altered. Thus, an index saves time in the reading of data, but it costs time in inserting and updating data.

b) Materialized View.: In a relational DBMS, a view is a virtual table representing the result of a database query. Whenever an ordinary view's table is queried or updated, the DBMS converts these into queries or updates against the underlying base tables. In a materialized view (MV), the results of the query are stored as a concrete table that may be updated from the original base tables. In addition, because the MV is manifested as a real table, anything that can be done to an ordinary table can be done to a MV, including the creation of indexes.

c) Query Execution Plan.: A query plan (or query execution plan) is a set of steps used to access information in a SQL relational DBMS. Since SQL (Structured Query Language) is declarative, there are, typically, a large number of alternative ways to execute a given query, with widely varying performance. When a query is submitted to the DBMS, the query optimizer evaluates some of the different possible plans for executing the query and returns the one it considers to be the best alternative.

d) Query Optimizer / Plan Selection.: The query optimizer is the component of a database management system that attempts to determine the most efficient way to execute a query. The optimizer considers the possible query plans for a given input query, and attempts to determine which of those plans will be the most efficient. The query optimizers use heuristics to select the best execution plan.

e) Nested Query.: A SQL nested query is a *SELECT* query that is nested inside a *SELECT*, *UPDATE*, *INSERT*, or *DELETE* SQL query.

III. Mathematical Model

This section presents a mathematical model that were developed to calculate the costs and benefits of creating any index or materialized view in a given database. This model was used in the implementation presented in Section IV.

The basic idea of the mathematical model is to recommend the best set of indexes and materialized views to help the database optimizer in the selection of the queries execution plan. Since the model will work in read and write databases it must always consider the disk space that will be used to store new indexes and MVs, the processing costs of maintaining new indexes and MVs, and the improvements (in query processing speed) after optimization.

The model assumes that there is a profile of the DBMS's use, i.e., there is statistical information about queries and updates, including the frequency of execution for each query and the frequency of updating (insert, delete or update) for each table. This information can be either a log of use or the expected use (defined by the user). With this information, the mathematical model will recommend the indexes and MVs that will be able to execute these queries and updates as fast as possible (considering some restrictions).

The first restriction of the model is about disk space:

$$UsedSpace \leq AvailableSpace \tag{1}$$

where $UsedSpace$ is the total disk space that will be used for the indexes and materialized views recommend by the mathematical model and $AvailableSpace$ is the total space available for the optimization. The $AvailableSpace$ is an input parameter of the model.

Since we are working with read and write databases it is necessary to care about updates that will be made in the original tables and will affect the new indexes and MVs. The next restriction of the model is about processing costs:

$$UpdateCost + QueryCost \leq InitialCost \tag{2}$$

where $UpdateCost$ is the total processing (and storage) cost for updating tables, indexes and MVs; $QueryCost$ is the total processing cost for queries; and $InitialCost$ is the processing (and storage) cost of the DBMS before the optimization. To calculate the $InitialCost$, the model needs some constants from the specific instance of the DBMS that will be optimized. Section IV presents a module of our implementation that calculates these constants for Oracle DBMS instances.

The two first equations ensure the recommended solution is a feasible solution (in the sense that it can be implemented without breaking the basic restrictions of disk space and it will not slow down the DBMS).

The values of $UpdateCost$ and $QueryCost$ are hard to calculate. The model must simulate the DBMS optimizer to obtain the most accurate values possible. Here, it is important to remember the DBMS query optimizers use heuristics to select the best execution plan for a query, our model assumes two things: (i) the query optimizer will not use any pre-processed query (for example, caches), and (ii) the query optimizer always will select the best execution plan. These two assumptions are necessary to keep the solution generic, in the sense that it will not be limited to the technology used in an specific DBMS.

The main goal of our approach is to select a set of indexes and MVs that minimizes the sum of $UpdateCost$ and $QueryCost$. Considering that $INDEXES$ and $MVIEWS$ are the set of indexes and MVs recommended by our approach, the main goal is to identify the indexes and views that compose these sets in order to minimize the sum presented in Equation 3.

$$\begin{aligned} MINIMIZE = \quad & UpdateCost_{(INDEXES,MVIEWS)} + \\ & QueryCost_{(INDEXES,MVIEWS)} \end{aligned} \tag{3}$$

where $UpdateCost_{(INDEXES,MVIEWS)}$ is the total update cost of the databases regarding the creation of all the indexes in $INDEXES$ and all materialized views in $MVIEWS$.

Equation 1 can be extended to:

$$StorageSpace_{(INDEXES,MVIEWS)} \leq AvailableSpace \quad (4)$$

where $StorageSpace_{(INDEXES,MVIEWS)}$ is the disk space used for storing the new indexes and MVs.

The next two subsections detail how the model estimates the costs for indexes and MVs.

A. Index

As presented, the model must consider the disk space and the cost (in time) for querying and updating the database. Defining $IndexSpace$ as the disk space required to store the indexes in $INDEXES$, we have the following equation:

$$IndexSpace_{(INDEXES)} = \sum_{\forall i \in INDEXES} (\#OfRows_{Table_i} * (Length_i + PointerSize)) \quad (5)$$

where $IndexSpace_{(INDEXES)}$ is the total space to store the indexes in the set $INDEX$; $\#OfRows_{Table_i}$ is the number of records from table i, and $TABLE_i$ is the table that is indexed by index i; $Length_i$ is the size (in bytes) of the fields that belong to the index i and $PointerSize$ is a constant that denotes the size of the pointer in the index i (the pointer is a field in the index that links each row in the index with a row of a table). The $PointerSize$ typically ranges from 2 to 8 bytes.

For each query that searches a field that is not indexed in the database, the query cost $QueryCost_{noindex}$ can be calculated as:

$$QueryCost_{noindex} = c1 * \#OfRows_{Table} * LengthOfPredicate \quad (6)$$

where $c1$ is a constant of search that must be calculated to the instance of the DBMS that will be optimized; $\#OfRows_{Table}$ is the number of records of the table that is being queried; and $LengthOfPredicate$ is the average size of the fields in the predicate of the query.

The cost for a query that searches one indexed table $QueryCost_{index}$ can be calculated as (considering indexing using the most common database index structure, such as B-trees):

$$QueryCost_{index} = c2 * log(\#OfRows_{Table}) * LengthOfPredicate \quad (7)$$

where $c2$ is a constant of indexed search that must be calculated to the instance of the DBMS that will be optimized. The value of $c2$ encapsulates two things (i) specific characteristics of the hardware and operation system where the DBMS is running (including block size), and (ii) the number of child nodes in the tree based index implementation (the implementation of B-tree or B-star trees in commercial DBMS typically use a value from 1,000 to 100,000 child nodes for each node).

The management cost (update, insert and delete) of an index, $ManagementCost_i$, is:

$$ManagementCost_i = c3 * FreqUpdate_T * (Length_i + PointerSize) \quad (8)$$

where $c3$ is a constant of management specific for each instance of the DBMS; $FreqUpdate_T$ is the frequency of updating from table T, which is indexed by index i and $Length_i$ is the average size of the fields from index i.

The proportional gains of creating an index i, $Gain_i$ is:

$$Gain_i = \frac{Freq_i * CostWithout_i}{Freq_i * CostWith_i + ManagementCost_i} - 1 \quad (9)$$

where $Freq_i$ is the frequency the index i will be used in query execution plans; the $CostWithout_i$ is the total $QueryCost_{noindex}$ (see Equation 6) of queries that can be improved with index i; $CostWith_i$ is the total $QueryCost_{index}$ (see Equation 7) of the queries regarding the use of index i; and $ManagementCost_i$ is the management cost for maintaining the new index i (see Equation 8). As presented before, the frequencies are input parameters from the DBMS use profile (workload).

This equation can be extended replacing $QueryCost_{noindex}$, $QueryCost_{index}$ and $ManagementCost_i$ by Equations 6, 7 8, respectively.

The Equation 9 calculates the proportional improvement of the DBMS considering queries to a specific table T that can be indexed by index i. Note that if $Gain_i$ is negative, the creation of index i will not help in improving the DBMS. Thus, this index will not be created. A second important aspect is that improvements of the creation of an index must be calculated considering the creation of all other indexes and MVs. One index can improve the DBMS, for example, only if there is no materialized view that already optimized some table.

The management cost $ManagementCost_mv$ of a materialized view is:

$$ManagementCost_i = c3 * FreqUpdate_i * (Length_i + PointerSize) \quad (10)$$

where $c3$ is a constant of management specific for each instance of the DBMS; $FreqUpdate_T$ is the frequency of updating from table T, which is indexed by index i and $Length_i$ is the average size of the fields from index i.

Now, we define a new variable $Efficiency_i$ that is the efficiency of the creation of index i, in terms of improvements, multiplied by the frequency the index i will be used by the DBMS optimizer and divided by space used:

$$Efficiency_i = Freq_i * Gain_i / IndexSpace_i \quad (11)$$

And this equation can be extended, replacing $Gain_i$ for Equation 9 and $IndexSpace_i$ for Equation 5.

B. Materialized Views

This subsection presents the calculus of the MVs storage space, costs, gains and efficiency following the same ideas used in the previous subsection.

While the number of rows in a view is limited by the product of the number of rows of each table involved (the Cartesian product), typically a view has only a fraction of these rows. If the user has an already populated database, the number of rows for each new materialized view can be easily calculated executing a simple *SELECT* over the database. However if the optimization will run in a new database the number of rows must be estimated considering the (estimated) size of each table related with the materialized view. This estimation can be calculated combining the following equations (the first for Cartesian product and the second for inner joins):

Thus, the $\#OfRows_{mv}$ (number of rows from a materialized view mv, considering the Cartesian product) is:

$$\#OfRows_{mv} = \prod_{\forall T \in TABLES_{mv}} \#OfRows_T \tag{12}$$

where $TABLES_{mv}$ is the set of all tables that compose the materialized view mv; and $\#OfRows_T$ is the number of rows from table T.

The $\#OfRows_{mv}$, considering only inner joins is upper limited by:

$$\#OfRows_{mv} \leq_{\forall T \in TABLES_{mv}} MAX(\#OfRows_T) \tag{13}$$

where $TABLES_{mv}$ is the set of all tables that compose the materialized view mv; and $\#OfRows_T$ is the number of rows from table T. Remember this equation gives an upper limit by $\#OfRows_{mv}$, that will be used only when the user wants to optimize a database that was not populated yet, otherwise it is easy to obtain the exact value of $\#OfRows_{mv}$. Typically, to estimate the number of rows from a view it is necessary to combine these two last equations (according to the predicates from the query that generates the view).

Let us create a new variable called $Length_{mv}$ that represents the average length of a row in the materialized view mv. Its value can be calculated as follows:

$$Length_{mv} = LengthExclusiveFields_{mv} + \sum_{\forall T \in TABLES_{mv}} (Length_{T_{mv}}) \tag{14}$$

where $LengthExclusiveFields_{mv}$ is the sum of the average length of each field that exists only in the materialized view mv (for example, composed fields or summary fields); $TABLES_{mv}$ is the set of all tables related with the materialized view mv; and $Length_{T_{mv}}$ is the sum of the average length of the fields from table T that are present in the view mv.

For each materialized view mv its storage space $Space_{mv}$ is:

$$Space_{mv} = \#OfRows_{mv} * Length_{mv} \tag{15}$$

The query cost $QueryCost_{no\_i\_mv}$ for a query that searches more than one table without any index is:

$$QueryCost_{no\_i\_mv} \leq c4 * \prod_{\forall T \in TABLES_Q} \#OfRows_T * LengthPredicate_T \tag{16}$$

where $c4$ is a constant for querying, specific of each instance of the DBMS; $TABLES_Q$ is the set of tables related to the query Q; and $LengthPredicate_T$ is the average length of the field from table T used in the predicate of the query. If no field from table T is used in the predicate of query Q, this value is one.

The query cost $QueryCost_{ind}$ for a query that searches more than one table with all tables indexed but no materialized view is:

$$QueryCost_{ind} \leq c5 * \prod_{\forall T \in TABLES_Q} log(\#OfRows_T) * LengthPredicate_T \tag{17}$$

where $c5$ is a constant of querying specific of the instance of the DBMS.

The query cost $QueryCostPerfectMatch_{mv}$ for a query that searches more than one table with a materialized view that matches exactly the query is a constant $c6$ (there is no processing cost to search the database because the result is the materialized view):

$$QueryCostPerfectMatch_{mv} = c6 \tag{18}$$

where $c6$ is a constant of execution query plan selection that is specific of each instance of the DBMS.

The costs calculated in Equations 16, 17 and 18 are extremes. Typically we found a combination of them: part of query is answered using a sequential search (no index), part is solved searching in an index and part is answered with the use of a materialized view. The sum of these costs corresponds to the total cost to answer a query in a database that has indexes and MVs.

The model must split each query in predicates and try to evaluate each predicate (or combination of predicates) in the best way, in order to minimize the total cost of answering a query (the sum of the costs for answering its parts). Thus, the total cost of query Q ($TotalQueryCost_Q$) is:

$$TotalQueryCost_Q = \sum_{\forall P \in PARTS_Q} QueryCost_P \tag{19}$$

where $PARTS_Q$ is the set of parts query Q was divided in, each part corresponds to one step in the execution plan for answering query Q; and $QueryCost_P$ is the cost to answer the part P of query Q. It is calculated following the equations 16, 17 and 18.

It is important to note that one of the main functionalities of a DBMS optimizer is to determine what will be the execution plan to answer one query (and thus it will determine what are the set of $parts$ that will be executed and in which order). In the next section we will present how we simulated a DBMS

optimizer to evaluate what indexes and MVs will be the best recommendation to help in the database optimization.

The management cost $ManagementCost_{mv}$ of a materialized view is:

$$ManagementCost_{mv} = c7 * \left(\sum_{\forall T \in TABLES_{mv}} FreqUpdate_T \right) * Length_{mv} \quad (20)$$

where $c7$ is a constant of management specific for each instance of the DBMS; $TABLES_{mv}$ is the set of tables related to the materialized view mv; $FreqUpdate_T$ is the frequency of updating from table T; and $Length_{mv}$ is the average length of each row in the materialized view, calculated using Equation 14.

We defined seven constants specific to each instance of the DBMS ($c1$ to $c7$). Their values must be obtained for each specific DBMS to keep the model generic and more accurate. Section IV describes how these values are measured in our implementation.

As we defined for indexes, we now define the gain and the efficiency for a materialized view. The proportional gains of creating a materialized view mv, $Gain_{mv}$ is:

$$Gain_{mv} = \frac{Freq_{mv} * CostWithout_{mv}}{Freq_{mv} * CostWith_{mv} + ManagementCost_{mv}} - 1 \quad (21)$$

where $Freq_{mv}$ is the frequency the materialized view mv will be used in query execution plans; the $CostWithout_{mv}$ is the total query cost $TotalQueryCost$ (see Equation 19) of queries that can be improved with the materialized view mv; $CostWith_{mv}$ is the total $TotalQueryCost$ of the queries taking into account the use of mv; and $ManagementCost_{mv}$ is the management cost for maintaining the new materialized view mv.

The efficiency $Efficiency_{mv}$ of a materialized view mv is:

$$Efficiency_{mv} = Freq_{mv} * Gain_{mv} / Space_{mv} \quad (22)$$

where $Freq_{mv}$ is the estimated frequency mv will be used by the DBMS in query execution plans; $Gain_{mv}$ is the improvement (benefits) that mv will introduce in the database (see Equation 21); and $Space_{mv}$ is the disk space necessary to store the materialized view mv.

C. Conclusions About the Model

While the model is generic and can be used to any DBMS, the selection of the best set of indexes and MVs is a computationally hard problem. The next section presents a heuristic algorithm that we developed in order to execute the optimization model in a feasible time.

The $Efficiency_i$ and $Efficiency_{mv}$ will be used only in our implementation. They are not necessary in the mathematical model. The basic idea of our implementation is, instead of testing all combinations of indexes and MVs to evaluate what is the best combination, we will use a greedy algorithm to construct the set of indexes and MVs. In each iteration, the greedy algorithm will add to the sets $INDEXES$ or $MVIEWS$ the index or MV that have the greatest $Efficiency$ and then the algorithm will re-calculate the $Efficiency$ of all the other indexes and MVs before execute the next iteration.

IV. Implementation Issues

Our implementation is composed by four modules: (i) Optimizer; (ii) Database Schema Identification; (iii) DBMS Performance Evaluation; and (iv) Database Query Identification. While Optimizer is a generic module for identify the best sets of indexes and MVs to optimize the DBMS, the other modules are specific for each kind of DBMS and produce the input parameters for the Optimizer.

The optimization process has three steps: (a) the production of input data for the Optimizer Module; (b) the execution of the Optimizer which produces a set of recommended indexes and MVs; and (c) the creation of the desired indexes and MVs. The result of the first step is a set of four XML based files. These files contain (i) the database schema; number of rows for each table and frequency of updating – it is produced by the Database Schema Identification; (ii) metrics of the specific instance of the DBMS – it is produced by the DBMS Performance Evaluation; (iii) the set of queries that are used to search the database and the frequency of use for each query – it is produced by Database Query Identification; and (4) the available disk space – this value is filled by the user. While some modules were developed to produce the input data for the Optimizer Module, the user can generate and/or update any of these files – for example, to introduce a new query that was not present in the queries input file.

We implemented the modules Database Schema Identification, DBMS Performance Evaluation and Database Query Identification for Oracle 10g DBMS, and a generic Optimizer Module that can be used to optimize any relational database management system that takes advantage of indexes and MVs during the plan selection.

The **Database Schema Identification** has three main functionalities: (i) identify the database schema, including all tables, synonymous, indexes and MVs; (ii) count the number of rows for each table; and (iii) identify the updating frequency for each table. The last information can be found only if the statistics are enabled in the DBMS. If it is disabled, the user will have to fill this field.

The **DBMS Performance Evaluation** executes several database operations in order to measure the DBMS constants presented in the mathematical model (see Section III). Table I summarizes the kind of operations that are executed to measure the values of these constants.

The **Database Query Identification** asks the statistics information from the DBMS to obtain all queries and their frequencies. This is the standard way to produce the query and frequency input file for the Optimizer Module.

In our use case, where we are optimizing the Brazilian Federal Revenue Database the queries and frequencies are obtained in a different way. Before the execution of any query, a custom officer creates a set of rules to identify, for example, suspicious tax declarations. These rules will generate database

Constant	Operations
c1	Queries in one table using predicates that involve not indexed fields.
c2	Queries in one table using predicates that involve indexed fields.
c3	Updates, inserts and deletes in one simple table.
c4	Queries involving more than one table using predicates over not indexed fields (without materialized views).
c5	Queries involving more than one table using predicates over indexed fields (without materialized views).
c6	Queries whose results are identical to some materialized view.
c7	Updates, inserts and deletes in tables that constitute parts of some materialized view.

TABLE I
CONSTANTS AND OPERATIONS EXECUTED TO MEASURE THEM

queries that will be applied to each taxpayer (identified in Brazil for a document called CPF). In other words, we have a set of queries that will be executed millions of times (one to each CPF from a taxpayer) and we can optimize the database (to answer these queries) before execute any of them. Thus, the query input file will be produced from the set of rules (instead of database statistics information) and the frequencies of the queries will be filled by the custom officer and it is calculated in terms of the amount of time the DBMS will have to answer these millions of queries (or the deadline that the officer has to identify the suspicious tax declarations).

The **Optimizer Module** is responsible for presenting the set of indexes and MVs that optimize the database. It takes as input the four files presented (three produced by the other modules and the fourth filled by the user – with the available disk space value).

The main idea of this module it to execute a simplification of the model presented in Section III. Instead of testing all the possible combinations of indexes and MVs to identify what is the best sets to optimize the database, our Optimizer Module uses a greedy algorithm to produce these sets. Another important idea is to identify what are the indexes and MVs that can improve the database (this module does not need to calculate the efficiency of all possible indexes because, for example, a lot of them will not be used to answer any query).

The algorithm of the Optimizer Module works as follows:
(i) it reads the four input XML files, translating them to its internal representation model;
(ii) it splits the nested queries in sub-queries;
(iii) for each field in the query statements *WHERE*, *GROUP BY* and *ORDER BY*, it creates a potentially useful index that receives as frequency of use, the frequency from the query. These indexes are inserted in a set called $INDEXES_{useful}$, if the index already exists in the $INDEXES_{useful}$ it creates a new copy of this index in the set.
(iv) for each index i in $INDEXES_{useful}$ with more than one field, the algorithm creates all combinations of indexes with a sub-set of the fields from the index i and assigns the frequency from index i to all new indexes;
(v) it merges all identical indexes (indexes with, exactly, the same fields) in $INDEXES_{useful}$, updating the frequency of the index with the sum of the frequencies of all identical indexes;
(vi) for each query, it creates a potentially useful MV that corresponds to the query and assigns to this view, the query frequency. The materialized view is inserted in a set called $MVIEWS_{useful}$;
(vii) a copy of the set $MVIEWS_{useful}$ called $MVIEWS_{initial}$ is created;
(viii) for each materialized view $mv1$ in $MVIEWS_{useful}$ it compares $mv1$ with the other MVs looking for a view $mv2$ that is a sub-set of $mv1$. Here, we say that $mv2$ is a sub-set (or a sub-view) of $mv1$ if $mv2$ is more restrictive than $mv1$ and $mv1$ has all information (fields) need to perform the predicates of $mv2$. Whenever this happens, the program adds the frequency of $mv2$ in $mv1$ (this means that if we create the materialized view $mv1$ and do not create $mv2$, $mv1$ is able to speed up the execution of the query that originate $mv2$).
(ix) for each materialized view mv from $MVIEWS_{initial}$ with more than one column in the *FROM* statement, it creates in $MVIEWS_{useful}$ the sub-materialized views (in this case, views that queries a lesser number of tables) from mv. Here we limited the creation of sub-materialized views with at most three tables (to avoid the exponential number of possible sub-views). Each one of the new (sub-)materialized views will receive the same frequency of mv.
(x) for each materialized view mv from $MVIEWS_{initial}$ with more than one predicate in the *WHERE* statement, it creates in $MVIEWS_{useful}$ the super-materialized views – in this case, the views less restrictive than mv and whose queries the same tables of mv. Here we limited the creation of super-MVs with at most three predicates (to avoid, again, the exponential number of views in $MVIEWS_{useful}$). Each one of the new (super-)MVs will receive the same frequency of mv.
(xi) it merges all identical MVs in $MVIEWS_{useful}$, updating the frequency with the sum of the frequencies of all identical MVs;
(xii) it starts an iterative algorithm that:

(xii.1) calculates the efficiency of each index in $INDEXES_{useful}$ and materialized view in $MVIEWS_{useful}$ (using the equations presented in Section III);

(xii.2) it inserts the index or MV that has the best efficiency and which storage space is lesser than the available disk space in the set $RECOMMENDATIONS$; and inserts this index or view in the internal representation of the database schema (so it will be considered as part of the database schema in the calculus of the next efficiency values);

(xii.3) it decreases the value of available disk space;

(xii.4) it removes the index or MV from its set ($INDEXES_{useful}$ or $MVIEWS_{useful}$);

(xii.5) it finishes the iteration if there is not enough available disk space or the sets $INDEXES_{useful}$ and $MVIEWS_{useful}$ are empty.
(xiii) it produces an output file with all indexes and MVs from $RECOMMENDATIONS$.

Steps (v) and (xi) are important to prize, respectively, the indexes and MVs that can speed up more than one query.

The **Creation of Indexes and Materialized Views** is not made by the Optimizer Module. It only produces a file with the list of recommended indexes and MVs. The user can copy the creation commands of all recommended items, or only the ones he/she prefers to create.

V. Preliminary Results

Our system was developed for helping the database optimizers in the plan selection. We specified the problem of optimization as: given a database workload (updates, inserts, deletes and queries), the database schema, and a set of constants measured from the instance of the DBMS and the available disk space the system must return the best set of indexes and materialized views to optimize the database.

While the main idea was to work with read and write databases and to obtain the workload from the DBMS, we obtained interesting results from other related problems. This section presents three database optimization and estimation problems that our system can help to solve: (i) optimization of read and write databases; (ii) optimization of read only databases; and (iii) estimation of database behavior for a given workload.

A. Optimization of Read and Write Databases

This problem motivated the work presented in this paper. There is a real database with statistics about its use (workload) that needed to be optimized. We used the Database Schema Identification, DBMS Performance Evaluation, and Database Query Identification to obtain the input parameters and run the Optimizer Module to recommend a set of indexes and materialized views that can speed up the database use.

This database belongs to the Product and Foreign Exporter Information System [Digiampietri et al. 2008] (one of the databases from the Brazil's Federal Revenue) running in a Oracle 10g DBMS. Our first tests presented very interesting results. We tested it using the real workload and artificial workloads to verify how the optimization worked. We had benefits in all tests varying from 10% to 60% (the set of queries and updates run from 10% to 60% faster).

The best results were obtained in the workloads that had more queries than updates. It was also interesting to note that, for these workloads, the querying processing cost was reduced from 3 to 14 times. It is important to note that the database had, before the optimization, indexes in all the primary and foreign keys and some extra indexes, but do not have any materialized view.

B. Optimization of Read Only Databases

This problem occurs, for example, when someone wants to create a data warehouse. We assume the data warehouse is rarely updated or never updated.

This is a common data optimization problem with some good solutions. We tested our approach to this problem because this is a specific scenery of the one presented in the previous sub-sections. To run these tests we updated the workload files from the previous tests replacing the tables' frequency of update for zero (no update).

Since there was no update costs for new indexes and materialized views, in these test our system produced bigger sets of indexes and MVs (the only constraint was the available disk space). The total query cost was from 6 to 18 times faster than the cost to the original database (before optimization). The costs from individual queries range from the same cost (the optimization does not helped that query) to executions 200 times faster (when, for example, there was a materialized view that matched exactly a "complex" nested query with some *joins*).

These numbers can seem impressing, but they are compatible with some results in the literature specific for the selection of indexes and MVs for read only databases [Harizopoulos et al. 2006].

C. Estimation of Database Behavior for a given Workload

The idea of this problem is to evaluate if it will be feasible to design a database given its expected workload. Here, our Optimizer Module will have as input the same inputs it had in Subsection V-A, but the difference is that the database was not implemented yet or there are no statistics about the database use. Thus, the estimated workload must be filled by the user and, if it is a new database, he/she must also fill the estimated number of rows for each table.

During the tests of the first prototype of our approach, we notice there were workloads not feasible (the DBMS will not be able to execute all the operations in the desired time). So, we include in the DBMS Performance Evaluation Module, a new component that evaluate the DBMS performance – specifically, the amount of parallel operations (selects, updates, deletes and inserts) the instance of the DBMS is able to execute per second. Since the Optimizer Module simulates the DBMS behavior to calculate the costs and benefits of each operation (with or without indexes and MVs), it is also able to say if the DBMS will be overloaded or not. One important thing here is that the DBMS can be overloaded before the optimization, but work fine after the optimization.

To evaluate the DBMS behavior for new databases, the Optimizer Module has to use, for example, Equations 12 and 13 instead of have the exact number of rows from a materialized view (because the tables does not exists yet). The initial results were satisfactory, but we are still working on obtaining more accurate results. The first tests presented a difference from 10% to 20% in the workload, i.e, some tests showed the DBMS was overload when it was only 85% used, other tests said it was not overload when the workload was about 105% of the real capacity of the DBMS.

VI. Related Work

There are several works about database optimization. While some developed new DBMSs, other developed special data structures that optimizes some kind of applications, or even developed new storing policies [Ma et al. 2007]. Moreover

there are works that introduced the use of materialized views in commercial DBMSs (e.g. [Goldstein and Larson 2001]). We focused our analysis on works that deal with the selection of indexes and materialized views for database optimization.

There are two main works with the same goals of our [Zilio et al. 2004], [Agrawal et al. 2000]. The strategies of these approaches are very similar to our: costs evaluation, benefits evaluation, heuristic selection of indexes and materialized views, etc. The important difference is about the specificity. While these approaches were developed inside a specific database, our solution is generic and can be applied to several relational DBMSs.

VII. Conclusions and Future Work

We specified a mathematical model for selecting the best set of indexes and materialized views to help DBMS optimizers in the plan selection task. Moreover, we implement a greedy algorithm to efficiently recommend the indexes and MVs. While the model and the Optimizer Module are generic solutions, we developed other three programs to extract basic information from Oracle DBMSs.

We tested our approach in three different use cases (i) optimization of read and write databases; (ii) optimization of read only databases; and (iii) estimation of database behavior for a given workload. We obtained satisfactory results in all of them, mainly in our main use case: the optimization of some real Brazil's Federal Revenue databases.

As future work we intent to extend the DBMS Performance Evaluation module in order to insert regressions to estimate the database indexing used in the databases that will be optimized. In the current version of this module, we are using the equations 7 and 17 which assume a tree-based indexing, and Equation 10 assumes linear growing in management of materialized views. While these assumptions are accurate enough for almost all actual DBMSs, insert the calculus of the exact formula in our DBMS Performance Evaluation module will make our solution more generic and robust. Another future step is to investigate how to improve our estimation of a DBMS behavior.

We are also developing a graphical tool to test and presents the performance results of the database before and after the optimization. All tools produced in this work will be available in the project home page.

Acknowledgment

The work presented in this paper was partially financed by grants from CNPq (*PIBIC* fellowship), University of Sao Paulo (*Ensinar com Pesquisa* fellowship) and the Brazilian Federal Revenue. We also acknowledge the researchers Jorge Jambeiro Filho, Siome Goldenstein, Jacques Wainer and Andreia Kondo.

References

[Agrawal et al. 2000] Agrawal, S., Chaudhuri, S., and Narasayya, V. R. (2000). Automated selection of materialized views and indexes in SQL databases. In *Proceedings of 26th International Conference on Very Large Data Bases*, pages 496–505. Morgan Kaufmann.

[Chaudhuri 1998] Chaudhuri, S. (1998). An overview of query optimization in relational systems. In *Proceedings of the 17th ACM Symposium on Principles of Database Systems (PODS)*, pages 34–43. ACM Press.

[Connolly and Begg 2004] Connolly, T. M. and Begg, C. E. (2004). *Database Systems: A Practical Approach to Design, Implementation and Management*. Addison-Wesley.

[Digiampietri et al. 2008] Digiampietri, L. A., Roman, N. T., Meira, L. A. A., Filho, J. J., Ferreira, C. D., Kondo, A. A., Rezende, R. C., Constantino, E. R., ao, B. C. B., Ribeiro, H. S., Carolino, P. K., Lanna, A., Wainer, J., and Goldenstein, S. K. (2008). Uses of artificial intelligence in the brazilian customs fraud detection system. In *Proceedings of the 9th Annual International Conference on Digital Government Research (dg.o 2008)*, Montréal, Canada.

[Goldstein and Larson 2001] Goldstein, J. and Larson, P.-A. (2001). Optimizing queries using materialized views: a practical, scalable solution. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 331–342.

[Harizopoulos et al. 2006] Harizopoulos, S., Liang, V., Abadi, D. J., and Madden, S. (2006). Performance tradeoffs in read-optimized databases. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB 2006)*, pages 487–498. ACM.

[Lightstone et al. 2007] Lightstone, S., Teorey, T., and Nadeau, T. (2007). *Physical Database Design: the database professional's guide to exploiting indexes, views, storage, and more*. Morgan Kaufmann Press.

[Loney and McClain 2004] Loney, K. and McClain, L. (2004). *Oracle Database 10g: The Complete Reference*. McGraw-Hill Osborne Media.

[Ma et al. 2007] Ma, H., Schewe, K.-D., and Wang, Q. (2007). A heuristic approach to cost-efficient derived horizontal fragmentation of complex value databases. In *18th Australasian Database Conference*, volume 63 of *CRPIT*, pages 103–111, Ballarat, Australia. ACS.

[Zilio et al. 2004] Zilio, D. C., Zuzarte, C., Lohman, G. M., Pirahesh, H., Gryz, J., Alton, E., Liang, D., and Valentin, G. (2004). Recommending materialized views and indexes with ibm db2 design advisor. In *Proceedings of the First International Conference on Autonomic Computing*, pages 180–188, Washington, USA.