Proactive Index Maintenance: Using Prediction Models for Improving Index Maintenance in Databases

André Medeiros[1,2], Aristênio Saraiva[2], Gustavo Campos[2],
Pedro Holanda[3], José Maria Monteiro[3], Angelo Brayner[4], Sérgio Lifschitz[5]

[1] IBM - Brasil
agbm@br.ibm.com
[2] Departamento de Computação, Universidade Estadual do Ceará (UECE) - Brasil
{aristenio, gustavo}@uece.br
[3] Departamento de Computação, Universidade Federal do Ceará (UFC) - Brasil
{pedroholanda, monteiro}@lia.ufc.br
[4] Programa de Pós-Graduação em Informática Aplicada, Universidade de Fortaleza (UNIFOR) - Brasil
brayner@unifor.br
[5] Departamento de Informática, PUC-Rio - Brasil
sergio@inf.puc-rio.br

Abstract. This article presents a mechanism, denoted Proactive Index Maintenance (PIM, for short), for proactive index management based on the use of prediction models. The main objective of the proposed mechanism is to predict when a time-consuming query q will be executed, in order to proactively create index structures which reduce $q's$ response time. After q is executed, PIM drops the created indexes for avoiding the overhead of updating index structures. Thus, indexes are automatically created and dropped by PIM in a proactive manner. PIM is DBMS-independent, runs continuously and with no DBA intervention. Experiments show that PIM presents low overhead, can be effectively deployed to predict time-consuming query execution and provides significant performance gain during time-consuming query execution. Different prediction models have been evaluated: neural networks (Multi-Layer Perceptron - MLP and Radial Basis Function - RBF) and Linear Regression. The results indicate that the prediction model is query-specific, i.e., it should be defined according to the statistical distribution (normal, poisson, binomial) of the query execution history.

Categories and Subject Descriptors: H.2 [**Database Management**]: Miscellaneous; H.3 [**Information Storage and Retrieval**]: Miscellaneous; I.7 [**Document and Text Processing**]: Miscellaneous

Keywords: prediction models, neural network, linear regression, index maintenance

1. INTRODUCTION

Nowadays, database applications may be very complex, dealing with a huge volume of data and a high demand concerning availability, query response time and transaction throughput. In this context, index structures play a fundamental role in database performances. This is because the existence of suitable indexes speeds up the execution of queries submitted to DBMS (Database Management System) [Morelli et al. 2009]. Thus, one of the main activities of a database administrator (DBA) is to maintain an appropriate index configuration, i.e., that minimizes the workload (set of queries and updates submitted to the DBMS) response time. However, manual adjustments (creation, drop and reindex) in index configuration have become very complex for current applications. They require a thorough knowledge of DBMSs implementation aspects, hardware capacity, physical database design strategies, characteristics of the stored data and complex workload [Chaudhuri and Narasayya 2007].

Several tools (wizards, advisers) and approaches have been proposed aiming to solve the problem of automatic index selection, such as DB2 Advisor [Valentin et al. 2000], Database Tuning Advisor (MS SQL Server) [Agrawal et al. 2004] and SQL Adjust Advisor (Oracle) [Dageville et al. 2004]. In recent years, some initiatives had present prototypes for automatic and continuous (online) index tuning [Bruno and Chaudhuri 2007a; 2007b; Schnaitter et al. 2006; 2007; Luhring et al. 2007; Sattler et al. 2003; 2003; Sattler et al. 2004]. However, these solutions are reactive, since they analyze a previously executed workload and infers the best index configuration *after* that workload has been executed. Therefore, reactive approaches react *after* performance problems are detected. In this sense, it would be interesting to predict characteristics of future workloads in order to proactively and automatically make the necessary adjustments in the index configuration.

Generally speaking, a database workload is a set of queries and transactions submitted to DBMS in a given period of time. In this work, we are especially interested in a particular type of queries: time-consuming queries. A given query q is characterized as time-consuming if *(i)* q has high response time whenever it is executed and; *(ii)* q is not executed in regular time intervals, but it is executed very often. It is important to note that a time-consuming query does not have a cyclic execution behavior, i.e., it is not executed every day, every week, every 15 days and so on and so forth.

It is well known that for time-consuming queries, the existence of indexes may reduce drastically the time required to process them. In this sense, it is quite appropriate to develop a tool which could predict the moment when time-consuming queries will be executed in order to create efficient index structures in an automatic and proactive manner.

This article proposes a mechanism for automatic and proactive index maintenance to speed up the execution of time-consuming queries. Our solution, denoted PIM, has the ability of predicting when time-consuming queries are going to be executed. By doing this, PIM forecasts future workloads in order to change the current index configuration *before* time-consuming queries are submitted to the DBMS. PIM uses different prediction models and it is independent of any human intervention. The proposed approach relies on the premise that a workload submitted to the DBMS in the future is similar to an already captured one.

In order to achieve its goal, PIM implements the following five steps:

(1) Continuous monitoring of the workload submitted to DBMS. The database workload is continuously captured and stored in the PIM metabase, denoted local metabase.
(2) Identification of time-consuming queries and appropriate index structures to execute such queries. PIM analyzes the previous captured workload to identify time-consuming queries and the most adequate index structures to each query. This analysis is based on the concept of *hypothetical query execution plans*.
(3) Selection of the most appropriate prediction model (for example, Linear Regression, Multi-Layer Perceptron - MLP, Radial Basis Function - RBF, etc) and the best configuration (values of required parameters for the chosen model), for each time-consuming query.
(4) Prediction of the moment in which a time-consuming query is executed. By means of a prediction model instance, PIM may forecast the moment when a time-consuming query will run. The prediction model instance uses workload logs (PIM metabase) to infer the next time a specific time-consuming query will be executed.
(5) Automatic and proactive index creation/drop. PIM has the functionality of physically changing the current index configuration to speed up the execution of predicted queries. By doing this, PIM acts proactively and autonomously to improve the performance of time-consuming queries. It is important to note that only in this step indexes are physically created or dropped.

The architecture used to implement PIM is presented. The architecture is DBMS-independent, its tasks are executed in a continuous manner and with no DBA intervention. Two prediction models were implemented. One model is based on linear regression and the other one implements in fact

two different types of neural networks: multilayer perceptron (MLP) and radial basis function (RBF). These models were chosen because they are widely used in time series forecasting. Nonetheless, others models can be added to the proposed architecture.

The proposed architecture has been implemented and tested its efficiency in a real environment. The results show the feasibility of predicting when time-consuming queries are executed. Furthermore, the results also indicate that the proactive index creation for time-consuming queries provides significant performance gains, even if the created indexes are dropped shortly after queries execution.

The remainder of this article is organized as follows. Section 2 describes works related to the index maintenance problem. In turn, Section 3 presents and discusses the proposed approach for proactive index maintenance. Section 4 describes the experimental results. Finally, Section 5 concludes this article and points out directions for future research.

2. RELATED WORK

In this section we discuss some relevant approaches for index maintenance. For the sake of clarity, we mention explicitly the main characteristics (e.g. online, instrusive) of each strategy, in order to motivate the proposed approach. It is important to note that no proactive approach was identified.

In [Bruno and Chaudhuri 2007a; 2007b], Bruno *et al* present an intrusive index tuning tool implemented as a component of Microsoft SQL Server 2005. This tool runs continuously and, either reacting to variations in the workload or considering data characteristics, it modifies the database physical design. The tool proposed in [Bruno and Chaudhuri 2007a; 2007b] works as follows: during the optimization process for a given query q, the optimizer call is diverted to an *Index Analysis* (IA) module, which identifies a set of candidate indexes CI that could potentially reduce the execution time for q. For this purpose, IA uses AND/OR request trees and local transformations. Thereafter the query q is optimized and processed as usual. During the execution of q, IA estimates the potential benefits if candidate indexes were used to process q. Moreover, it evaluates the benefits of real indexes, which were used during query execution. These steps are performed by the *Cost/Benefit Adjustment* (CBA) module. After a query q is executed, IA is triggered once more to analyze the cost/benefit of using candidate indexes belonging to CI. Based on this analysis, the IA module sends requests for creating or dropping indexes to the *Asynchronous Task Manager* module (ATM). It is important to note that this approach adds some new tasks which are executed before and during query optimization. Consequently, such a feature may negatively impact the query processing performance.

Schnaitter *et al* present in [Schnaitter et al. 2006; 2007] a prototype of a self-tuning framework called COLT (Continuous On-Line Tuning). This tool continuously monitors queries submitted to the DBMS and autonomously adjusts the index configuration, considering space restrictions for these structures. COLT was implemented in an intrusive manner into PostgreSQL, since it replaces the PostgreSQL's query optimizer by a module, denoted *Extended Query Optimizer* (EQO), and adds a Self-Tuning Module (STM) to PostgreSQL's query engine. COLT implements the classic self-tuning steps composed by Observation, Prediction and Reaction phases. In order to achieve its goal, for each submitted query q the STM selects a set of candidate indexes (CI), which includes hypothetical and real indexes, and sends CI to EQO. For each index $i \in CI$, EQO generates a new query plan for q that uses the index i and computes the corresponding gain. Thus, if CI has n indexes, EQO has n query plans for q. In COLT, the duration of the observation phase is defined by the notion of an *epoch*. More specifically, an *epoch* represents the time for executing ten queries. During an *epoch* the STM gathers statistics about the workload and computes the gains of using candidate indexes. At the end of an *epoch*, the set of candidate indexes (materialized or hypothetical) is analyzed to verify the need for changes in the index configuration. In this case, hypothetical indexes may be materialized and materialized indexes may be removed (in this case, they become hypothetical indexes). The main drawback in this COLT approach stem from the notion of *epoch*. By assigning to an *epoch* a fixed

number of 10 queries, COLT may take too much time to react. Actually, any fixed number may not be consistent with the database system needs.

Luhring *et al* [Luhring et al. 2007] propose extensions to PostgreSQL for supporting index self-tuning functions. That approach is intrusive and uses the classic self-tuning strategy composed of the Observation, Prediction and Reaction phases. These steps are performed sequentially and continuously. During the observation phase, each submitted query q is analyzed in order to discover candidate indexes and the heuristic described in [Valentin et al. 2000] is used. Thereafter, q is optimized twice, once without considering any index and another considering all candidate indexes. This approach also applies the notion of an *epoch* [Sattler et al. 2004] to define the duration of the observation phase. However, an epoch in [Luhring et al. 2007] is defined in terms of the maximum number of recommendations for the same index.

In [Sattler et al. 2003; Sattler et al. 2004], Sattler *et al* propose a middleware for IBM DB2 that automatically suggests the creation of indexes. That solution is based on DB2 proprietary commands (e.g *SET CURRENT EXPLAIN MODE RECOMMEND INDEXES*), which are not available for other DBMSs. Furthermore, it requires all SQL clauses to be forwarded to the middleware, and not to the DBMS. Their approach presents two critical drawbacks: on one hand, code rewriting becomes mandatory for previously existing applications, and on the other hand, workloads directly submitted to the DBMS cannot be managed since the workload is forwarded to the middleware.

It is worth mentioning that the strategies proposed in [Schnaitter et al. 2006; Luhring et al. 2007; Sattler et al. 2003; Sattler et al. 2004] present the following drawbacks: *(i)* they require additional calls to the database optimizer; *(ii)* they only consider the use of secondary (non-clustered) indexes; and *(iii)* they ignore the problem of tracking the fragmentation rate of index structures.

The work proposed by Morelli *et al* [Salles and Lifschitz 2005; de Carvalho Costa et al. 2005; Morelli et al. 2009] presents a self-tuning component implemented within the PostgreSQL code, allowing autonomous index creation, dropping and rebuilding. The optimizer takes into account hypothetical indexes for the construction of alternative query plans. The developed prototype does not consider important restrictions such as those regarding the physical space available for the materialization of the suggested (hypothetical) indexes.

A framework, and associated tools, that can help an administrator understand the interactions within the recommended set of indexes is proposed in [Schnaitter et al. 2009]. The authors formalize the notion of index interactions and present a novel algorithm to identify the interaction relationships that exist within a set of indexes. Furthermore, it describes two new intrusive database tuning tools for IBM DB2 that utilizes information about index interactions. The first tool visualizes interactions based on the partitioning of the index-set into non-interacting subsets, and the second tool computes a schedule that materializes the recommended indexes over several maintenance windows with maximal overall benefit.

In [Maier et al. 2010; Alagiannis et al. 2010] an intrusive solution, called PARINDA (PARtition and INDex Advisor), is presented. It is an interactive physical designer for PostgreSQL that works as follows: given a workload containing a set of queries, PARINDA allows the DBA to manually suggest a set of candidate indexes and the tool shows the benefits of the suggested index set and index interactions visually. Besides, the tool can find the optimal index partition for a given query workload. It also suggests a schedule to implement the suggested indexes. Finally, the tool can continuously monitors the performance of the DBMS under incoming queries, and it suggests new indexes when they offer sufficient workload speedup. Bruno et al propose in [Bruno and Chaudhuri 2010] an intrusive and interactive tool for Microsoft's SQL Server, which is similar to PARINDA since it makes tuning sessions interactive, allowing DBAs to try different tuning options and interactively obtain a feedback.

A new index recommendation technique, called semi-automatic index tuning, is proposed in [Sch-

naitter and Polyzotis 2012] based on the notion of a *work function* algorithm. The idea behind the proposed technique is to keep the DBA "in the (tuning) loop" by generating recommendations that use feedback about the DBA's preferences. The technique also works online, that is, it does not require the workload to be known in advance. However, this semi-automatic tuning delegates to the DBA the final decisions on index materialization.

In [Jimenez et al. 2011] the authors provide a comparative evaluation of the three online index-tuning algorithms ([Bruno and Chaudhuri 2007a], [Schnaitter et al. 2007] and [Luhring et al. 2007]), in the same environment and with the same experimental methodology. The implementation of this performance benchmark provides some insights on the relative performance, strengths and weaknesses of these algorithms.

It is important to emphasize that all works presented in this section adopt a reative approach. Reactive approaches only react *after* performance problems are detected. In other words, reactive approaches change index configuration *after* a workload is captured. Such a characteristic represents a potential drawback and may negatively influence the overall DBMS' performance. On the other hand, PIM forecasts future workloads in order to automatically create indexes *before* time-consuming queries are submitted to the DBMS.

3. PIM: ENSURING PROACTIVE INDEX MAINTENANCE

In this section we will describe the proposed architecture and approach for implementing proactive index maintenance in DBMSs.

PIM executes workload prediction and index maintenance concurrently to the normal DBMS operation. Such a feature is guaranteed by means of a software component collaboration, which uses drivers to access and manipulate information from the analyzed DBMS. Figure 1 depicts an abstract model of PIM architecture. To utilize PIM in a given DBMS, it is necessary to instantiate three specific drivers: a driver for workload access (DWA), a driver for statistics access (DSA) and finally a driver for DBMS update (DDU). These three drivers are the only components of PIM which are DBMS-specific. Nevertheless, these drivers are implemented in a non-intrusive fashion. We have so far implemented drivers for the following DBMSs: PostgreSQL 8.4, Oracle 11g and SQL Server 2008.

The main components of the architecture illustrated in Figure 1 are the following:

—**Workload Obtainment (WO)**. The WO component is responsible for periodically capturing the workload submitted to the DBMS. Thus, WO has to access the DBMS catalog in order to get the tasks executed by the DBMS. Thereafter, the triple *<SQL Expression, execution plan, estimated cost>* representing each captured task is stored in a structure, called Local Metabase (see below). Therefore, not only queries but their corresponding execution plans and execution costs are stored and become available for proactive index maintenance.

—**Local Metabase (LM)**. LM stores the workload captured by WO, the set of time-consuming queries identified by PIM, a set with the more appropriate index structures to reduce the response time for each time-consuming query, the most appropriate prediction model (linear regression, MLP, RBF) for each time-consuming query and a set of data on index structures managed by PIM. For each index structure, PIM stores in LM the following data: the structure id, the name of the table on which the structure is defined, the columns that compose the search key, the index type (primary or secondary) and the estimated cost for physically creating the structure.

—**Driver for Workload Access (DWA)**. This component enables PIM to access metabase (catalog) of a given DBMS.

—**Driver for Statistics Access (DSA)**. This driver is responsible for ensuring the access to DBMS statistic data, such as relation cardinalities.

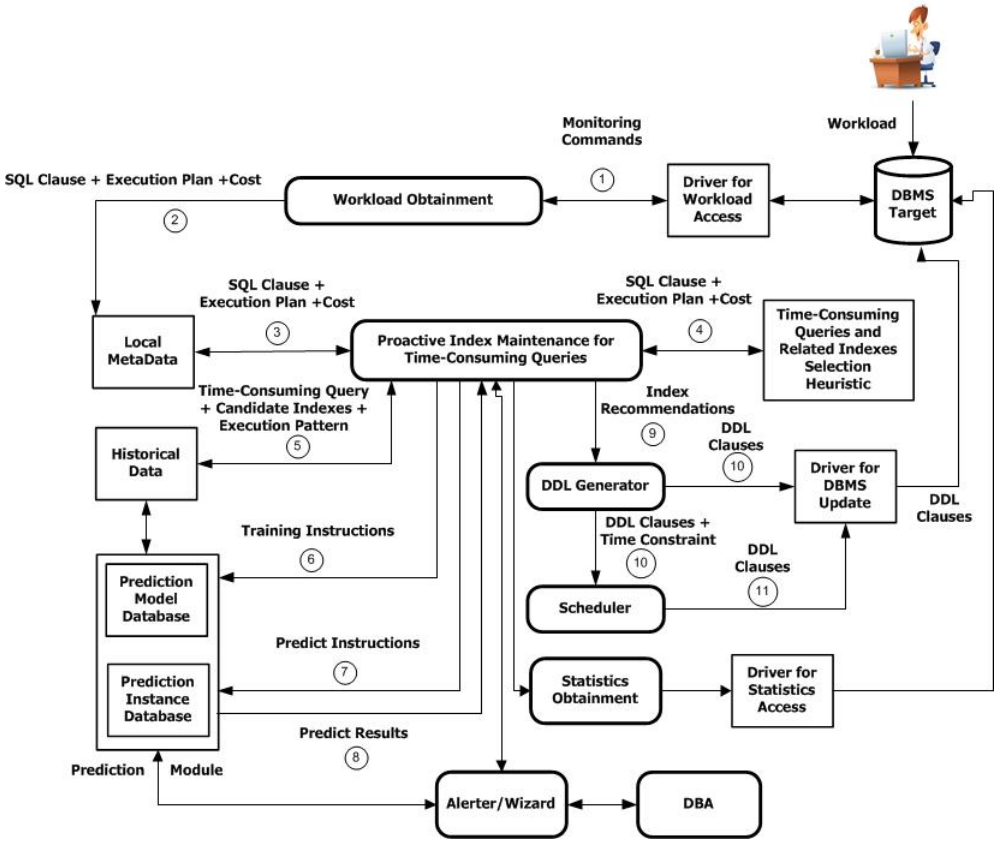

Fig. 1. Architecture for Proactive Index Maintenance.

—**Statistics Obtainment (SO)**. This component is in charge of accessing statistics information of the target DBMS, such as table cardinality, the amount of disk pages required to store a database table, the height of index structures (B^+ trees) and so forth.

—**Prediction Module (PM)**. This component is responsible for managing *(i)* a prediction model database whose main functionality is to store prediction models which can be used by PIM (eg, linear regression, MLP, RBF, etc.) and *(ii)* a database of prediction model instances. A prediction model instance consists of a Java object, which in turn implements a particular prediction model from a given set of parameters values.

—**Proactive Index Maintenance for Time-Consuming Queries (PIM)**: This component is responsible for executing index maintenance by using historical data and the Prediction Module (prediction model and instance databases).

—**DDL Generator (DG)**. This component builds DDL commands in SQL for either creating or dropping index structures into the target DBMS.

—**Driver for DBMS Update (DDU)**. It is responsible for enabling the execution of DDL commands (built by the DG component) in the target DBMS.

—**Scheduler (S)**. Creating index structures during DBMS workload peaks may compromise the performance. In order to overcome this problem, this component monitors the current DBMS workload to find the best moment to create index structures. In other words, the Scheduler tries to schedule index maintenance actions when the DBMS workload is in a low operation level.

The approach proposed in this work for proactive index maintenance consists of five steps. Next, each one of these steps will be discussed in detail.

3.1 Step 1: Monitoring and capturing the workload

This step consists of extracting from the target DBMS's log the workload submitted to the DBMS. Thus, WO has to access the DBMS catalog in order to get the tasks executed by the DBMS. Thereafter, the triple <*SQL Expression, execution plan, estimated cost*> representing each captured task is stored in a structure, called Local Metabase. Therefore, not only queries but their corresponding execution plans and execution costs are stored and become available for proactive index maintenance. Of course, the way to obtain this information varies for each DBMS product, since each DBMS has its proper catalog. Thus, to encapsulate such differences the designed architecture implements the mechanism of drivers. Thus, there is a specific driver for each DBMS. These drivers are used by WO in order to obtain the workload submitted to the target DBMS. For that reason, the WO implementation becomes independent of particular aspects of each DBMS catalog. The WO component captures the workload and then stores this information in the local metabase (LM). Additionally, the DBA (Database Administrator) may alternatively provide a file containing previously workloads, if he/she has that information or is able to extract it from the DBMS log.

3.2 Step 2: Identifying time-consuming queries and efficient index structures

The solution proposed to identify the time-consuming queries is based on the benefit heuristic, initially proposed in [Salles and Lifschitz 2005]. The notion of benefit quantifies the gain in using of a specific index structure i to process a query q.

Definition 3.1 *Benefit.* Let $B_{i,q}$ the benefit provided by the index structure i for processing a query q. The benefit of using i to run q is computed as follows:

$$B_{i,q} = max\{0, cost(q) - cost(q,i)\},$$

where, $cost(q)$ represents the cost of running the query q without the use of index i and $cost(q,i)$ represents the cost of running the query q using the index i.

Now we can formally define the concept of time-consuming queries.

Definition 3.2 *Time-Consuming Query.* A query q is considered "time-consuming" if and only if: *(i)* $RT_q > t$, where RT_q is the response time of q and t is a constant (parameter), *(ii)* $F_q < k$, where F_q is the number of executions of q divided by the size of the observation period, given in months, and k is a constant (parameter), and *(iii)* there is at least one index structure i such that $B_{i,q} > EC_{C_i}$, where EC_{C_i} is the estimated cost of creating the index structure i.

In order to identify time-consuming queries and the most appropriate indexes for each one of these queries, PIM uses an adaptation of the strategy proposed in [Monteiro et al. 2008]. The mechanism proposed in [Monteiro et al. 2008] uses the concept of Hypothetical Execution Plan (for short, HP). The key idea behind the concept of HP is to identify indexes which could bring benefits to a given query. Thus, after obtaining the real execution plan of query q (i.e., the execution plan generated by the DBMS native query optimizer), the captured real plan is traversed for searching for operations, which do not use indexes, such as full scans on tables. The goal is to replace such operations in hypothetical execution plans by equivalent physical operations which make use of indexes, such as index scans.

Hypothetical execution plans may have real or hypothetical indexes. A real index structure physically exists in disk, while a hypothetical index structure does only exist in PIM metabase. Indexes identified during the execution of this step are considered the most appropriate to accelerate the execution of the analyzed query. PIM selects hypothetical indexes whose benefit to the analyzed query q

($B_{i,q}$) is greater than its creation cost (EC_{C_i}). The set of indexes selected for a query q is called I_q. This task is performed periodically and automatically by PIM.

The set of time-consuming queries is automatically built by PIM as follows. For each SQL query q captured from LM (see Figure 1), PIM retrieves $q's$ real execution plan (generated by the native query optimizer), which is stored in LM as well. Thus, PIM is able to construct a hypothetical plan of q. By using Definition 3.2, PIM classifies q as a time-consuming or not. Next, the PIM stores in LM: *(i)* the information that q is a time-consuming query and *(ii)* the set I_q. Note that this task is carried out automatically by the PIM.

Nevertheless, DBAs may provide to PIM the set of time-consuming queries and the most appropriate indexes for each query.

3.3 Step 3: Selecting appropriate prediction models for time-consuming queries

In this step, for each time-consuming query q, the PIM component captures the execution history of q. Thereafter, from an external tool, PIM gets the data distribution curve and finds the best statistical distribution for representing the execution history of q (i.e., normal, binomial, poisson , etc.). From the statistical distribution the PIM infers the best prediction model for q (linear regression, MLP, RBF, for example). Thus, PIM may create and evaluate different prediction instances (configurations) of the chosen prediction model to find out which instance gives the best results, regarding the metrics correlation coefficient and root mean squared error. The instance with best results is associated to q, in order to be used to predict the next executions of q.

3.4 Step 4 : Predicting time-consuming query execution

At this point, time-consuming queries and the set of appropriate indexes for each one of those queries have already been identified. Besides, the most efficient prediction model for each time-consuming query has been chosen. Thus, PIM is able to instantiate the chosen prediction model for each time-consuming query.

PIM uses the prediction model instance for a given query q to forecast the next time q is going to run again. For this, PIM component captures from LM (local metabase, see Figure 1) the n last executions of q (n depends on the time window used by predication model instance) and predicts when q will run again by applying the prediction model on n. PIM sends a request to DDL Generator (Figure 1) to schedule the creation of indexes associated to q (in I_q). This request is composed of the following triple: $< q, I_q, N_q - t >$, where N_q is the predicted date for the next run of q and t is an integer. The algorithm presented in Figure 2 details this activity.

It is important to note that the indexes creation are always schedulled to t days before N_q. Thus, the higher the value of t the greater the chance of having a correct prediction, ie, the indexes are indeed created before the query execution. On the other hand, the higher the value of t the greater the overhead of running update on the indexes due to possible update operations on the table where the index is defined. Furthermore, PIM component monitors the workload submitted to DBMS in order to discover if a given query q (whose execution was predicted and whose indexes belonging to I_q have been created proactively) has already been performed. It is also important to decide the time for which the index structures should stay materialized, since indexes structures consume disk space and may suffer from the update overhead problem. In the proposed approach, indexes are removed in one of the following events: *(i)* when the query q already been executed (then the indexes in I_q can be dropped) or *(ii)* k days after N_q. The algorithm showed in Figure 3 ilustrates these two situations.

Algorithm 1 APIM_P Heuristic: Query Prediction

```
1: for each captured query q ∈ LM do
2:    if q is a time consuming query then
3:       L_candidates ← the candidate index list used in the hypothetical plan generated for q
4:       for each index i, such that i ∈ L_candidates and B_{i,q} > EC_{C_i} do
5:          I_q ← i
6:          L ← the last n executions of q day and month, for example, where n is the number of time-windows
7:          N_q ← the expected date for the next run of q //This date is obtained using a specific prediction instance
            taking the set L as input
8:          send a create index request to DDL Generator component containing < q, I_q, N_q − t >
9:          insert < q, I_q, N_q > in the Local Metabase
10:      end for
11:   end if
12: end for
```

Fig. 2. PIM_P Heuristic: Query Prediction

Algorithm 2 PIM_M Heuristic: Query Monitoring

```
1: for each tuple < q, I_q, N_q >∈ LM do
2:    if q already runs OR the current date > N_q + k then
3:       send a drop index request to DDL Generator Component containing I_q
4:    end if
5: end for
```

Fig. 3. APIM_M Heuristic: Query Monitoring

Algorithm 3 EA Heuristic: Index Maintenance

```
1: for each received request r do
2:    if r is a create index request containing < q, I_q, N_q > then
3:       for each index i ∈ I_q do
4:          schedule the creation of i to the night preceding N_q − t >
5:       end for
6:    else if r is a drop index request containing I_q then
7:       for each index i ∈ I_q do
8:          drop i
9:       end for
10:   end if
11: end for
```

Fig. 4. EA Heuristic: Index Maintenance

3.5 Step 5: Index creation/drop in an automatic and proactive manner

During the execution of this step, each index creation request (containing $< q, I_q, N_q - t >$) received by the DDL Generator component is sent to the Scheduler (Figure 1), which in turn schedules the index creation to the night before $N_q - t$, where t is an integer value. This task is depicted in Figure 4. In the experiments, we have used different values of t. It is important to highlight that the Scheduler component tries to schedule index creation actions when the DBMS workload is in a low level.

4. EVALUATION

In order to evaluate the proposed approach we will present and analyzes an execution scenario to illustrate how PIM works. The idea is to describe and analyze how each step implemented by PIM works.

4.1 Step 1: Monitoring and capturing the workload

In this step, the queries execution history in the target DBMS is captured by PIM. There are two possibilities for executing this procedure: *(i)* PIM may access the local metabase (LM), which contains all queries captured by WO component, and *(ii)* PIM may read a file containing the queries executions history, which is provided by a DBA. In the second case, the DBA has to analyze the target DBMS transaction-log file, which stores all SQL clauses of all transactions submitted to the DBMS, and extract the required information (SQL clause, query execution plan, costs and datetime). In this work we have usee the log file as a source of historical records of executed queries. This choice was motivated mainly by the fact that the local metabase (LM) does not have yet a significant amount of information.

Thus, we analyzed log files of a real database used in a multinational corporation, whose identity may not be revealed for confidentiality reasons. We analyzed the records of the transactions executed, between 2008 and 2010, in a production database, used by an OLTP application. This database consists of 1531 tables and has a size of 1.5 TB. Information regarding executed queries, query execution plans, costs as well as execution date and time were extracted from the log files, imported by PIM and stored in the LM.

4.2 Step 2: Identifying the time-consuming queries

Initially, this step aims to identifying a set of time-consuming queries Q. Next, for each query $q \in Q$, it is need to discover the set of more appropriate index structures (to reduce the response time of q), denoted by I_q. This information can be obtained in two different ways: i) automatically by the PIM component, which can infer this data from the LM, and ii) the DBA can provide a file with this information to be imported by PIM. In this running example we chose to provide a file containing: i) a set of time consuming queries and ii) for each query, the set of most appropriate indexes. This file was exported to PIM. In order to create this file we process and analyze the history of the queries executed between 2008 and 2010, with the purpose of: i) select (filter) only time-consuming queries and ii) decompose the datetime value of each query execution register (retrieved from the database log) in two distinct values: day (1-31) and month (1-12). To filter time-consuming queries we have selected queries with: $RT_q > 9000s$, $F_q < 5$ (q is executed between one and four times per month, on average). Moreover, the queries should have irregular distribution. By irregular distribution we mean that such queries are not executed in regular periods of time. These values were chosen aiming at to generate a small but representative query sample, which was used to validate the proposed approach and the use of prediction models. Thus, the following queries were selected:

(1) Warning debt query: This query displays the users who are defaulting with the company and have been warned of the debt.
(2) Defaulting query: This query returns the number of accounts in arrears and debt value, showing users who are in debt and the value of the debt.
(3) Default debt report query: This query displays a report of all debts in the current month related with the debits in the previous month.
(4) Inactive customers query: List the customers checked as inactive.
(5) Debt parcel summary query: This query lists the customers who have scheduled their debts in installments.

Table I. The Most Appropriate Index for each Time Consuming Query

Query	Table	Column
Query 1	Warning	warning_type
Query 2	Payment	customer_id
Query 3	Bill	bill_type
Query 4	Customer	fl_active
Query 5	Parcel	parcel_type

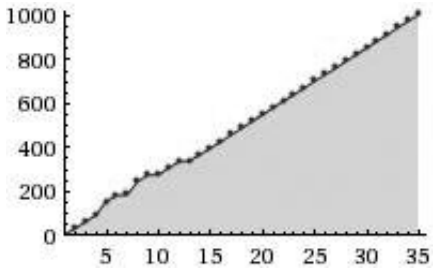

Fig. 5. Distribution Curve of Query 1.

Next, for each time-consuming query $q \in Q$, PIM executes the algorithm proposed in [Monteiro et al. 2008] in order to find the most appropriate index for q (I_q). Thereafter, a file containing Q and I_q for each $q \in Q$ is created. This file was exported to PIM. The key attribute on the set I_q are illustrated in Table I.

4.3 Step 3 and Step 4: Selecting the most appropriate prediction model and Predicting time-consuming query execution

For each time-consuming query, the most appropriate prediction model should be defined. This task can be automatically done by PIM component or the DBA may provide this information to PIM. Now, for each time-consuming query $q \in Q$, the execution history of q, stored in the local metabase, is accessed by PIM. Thereafter, the captured execution history is passed to an external statistical tool for obtaining the data distribution curve for q and for finding the best statistical distribution for representing the execution history of q (normal, binomial or poisson). Figure 5 shows the distribution curve for Query 1, wich follows a normal distribution. From the statistical distribution, the best prediction model for query q (linear regression, MLP or RBF) can be inferred. For example, since Query 1 has a normal distribution, an appropriete prediction model to Query 1 may be linear regression. Next, different prediction instances (configurations) of this model are created and evaluated to find out which instance gives the best prediction results, evaluating the correlation coefficient and the root mean squared error. The instance with best results is associated to the query q, in order to be used to predict the next executions of q.

In this work we evaluated three different prediction models: linear regression, multilayer perceptron (MLP) with time windows and radial basis function (RBF). For each prediction model we have evaluated several prediction instances. In order to instantiate each prediction model, four different formats for input and output data have been evaluated.

The first format utilizes classifiers. Thus, the idea is to build the network input by means of the variables day and month, while the output consists of a pair containing a day and a month separated by a hyphen. Whenever classifiers are used, it is necessary to declare all possible values for input and output of the neural network, which is called the nominal specification. In the second format, the network input is formed by the variables day and month and the output is day of year, i.e., a value between 1 and 365. In this case is not necessary to declare all the possible input and output values. Nevertheless, input and output data type should be specified. In the third format, the network input and output parameters are represented by day of year, i.e., a value in the interval [1, 365]. Finally,

Table II. Parameter Settings Used in Training and Validation Phases of MLP Prediction Instances.

Configuration	Hidden Layers	Learning Rate	Training Time
1	5, 10, 20	0,1	500
2	5, 10, 20	0,3	300
3	5, 10, 20	0,1	2000
4	5, 10, 20	0,3	1000

Table III. Parameter Settings Used in Training and Validation Phases of RBF Prediction Instances.

Configuration	Number of Clusters
1	2
2	4
3	6
4	8
5	10
6	12
7	20

the fourth format is similar to the previous one, however it uses as input and output an integer value that represents a day in the observation period, i.e., a day between the years 2008 and 2010. Thus, the network input and output values assume a value ranging from 1 to 1095 (365 x 3).

In order to build instances for MLP and RBF models, the four different input/output formats have been employed. For each one we have tested three different time windows, namely 2, 3 and 4 time windows. Combining a input/output format with a given time window, a prediction instance in the case of MLP or RBF models has been generated. For linear regression the time window parameter is not applied and the fourth input/output format has been employed.

For the MLP network, we have adapted the basic algorithm to enable the execution of dynamic tasks, since MLP does not support this property. To do this, we have added a time delay in the network input. Thus, the network input has values in a time-ordered sequence, where in each iteration the input set is initiated with the value that is subsequent to the initial value in the previous sequence.

As a result, for each of the five selected queries, we have used 2 neural networks (MLP and RBF), 4 different configuration setting and 3 time windows (2, 3 and 4), totaling 120 neural networks instances. On the other hand, for each select query, there is only one linear regression instance. Therefore, 125 predication model instances have been evaluated. To implement, train and validate each neural networks instance, the WEKA (Waikato Environment for Knowledge Analysis) tool has been applied. WEKA is composed of several neural network (and data mining) algorithms and techniques. To implement, train and validate each linear regression instance we have implemented a Java application.

We have evaluated each MLP instance by changing the values of the following neural-network-specific parameters, Hidden Layers, Learning Rate and Training Time. The first parameter defines the hidden layers of the neural network. This parameter consists by a list of positive integers, one for each hidden layer representing the number of neurons in this layer, separated by commas. Thus, the value "5,10,20" means 3 layers with 5 neurons at the first layer, 10 neurons at the second layer and 20 neurons at the third layer. The second parameter is the neuron weights and the third parameter is the number of epochs used to train the network. The MLP instances were tested according to the parameter combination described in Table II.

We have tested the RBF prediction instances by changing the values of parameter Number of Clusters. This variable represents the amount of clusters that the RBF instance will use. The RBF prediction instances were tested following the settings described in Table III.

Thereafter, PIM had created the prediction instances, the training phase for each prediction instance is triggered. As already mentioned, for training the MLP and RBF instances, we have used WEKA. For linear regression instances we have implemented a Java application.

Table IV. The Best Prediction Instances for Time-Consuming Queries.

Query	Prediction Model	Configuration	RMSE
Query 1	LR	60% Training, 40% Validation	2,6
Query 2	RBF	input/output format = 4, Windows = 4, Clusters = 20	9,42
Query 3	RBF	input/output format =4, Windows= 3, Clusters =8	42,48
Query 4	MLP	input/output format =4, Windows= 3, Configuration 3	22,95
Query 5	RBF	input/output format=4, Windows=2, Clusters=20	33,89

To evaluate the results provided by MLP and RBF instances in the validation phase we used the cross-validation method with 10 partitions. This method is widely used in scenarios where the goal is prediction. It requires partitioning the data into 10 subsets. One of those subsets is used for the training phase, while the other nine are used for validation. In the case of linear regression instances, 60% of the historical data were used to the training phase and 40% for the validation phase.

To identify which prediction instance gives the best prediction results for a query q, the correlation coefficient and the root mean squared error metrics are used by PIM. The correlation coefficient (Equation 1) can assume values between -1 and 1. The signals are used to indicate a positive or negative linear correlation. Values close to -1 indicate a strong negative linear correlation between predicted and real values, while values close to +1 indicate a strong positive linear correlation. Values between 0.30 to 0.70 indicate positive or negative moderate correlation.

The root mean squared error (RMSE) is a quadratic scoring rule, which measures the average magnitude of the error. The difference between forecast and corresponding observed values are squared and then averaged over the sample. Next, the square root of the average is taken. Since the errors are squared before they are averaged, the RMSE gives a relatively high weight to large errors. Thus, RMSE is most useful when large errors are particularly undesirable. It is important to note that the results may vary depending on parameter values. The instance with best results w.r.t. correlation coefficient and root mean squared error is associated with the query q, in order to be used to predict the next executions of q. Table IV shows the best prediction instance for each time-consuming query. Figure 6 show RMSE for the best predicition instance for each analyzed prediction model (LR, MLP and RBF) in the Query 1 context.

$$r = \frac{n\sum x.y - (\sum x)(\sum y)}{\sqrt{[n\sum x^2 - (\sum x)^2].[n\sum y^2 - (\sum y)^2]}} \quad (1)$$

To compute prediction by means of linear regression, the following equation has been adopted:

$\hat{S}(t) = a + bt$

The time t is an independent variable. $\hat{S}(t)$ represents the estimated value of $S(t)$ and is variable with t. Parameter a is the interceptor-t (value of $\hat{S}(t)$ for t=0) and b is the strech slope. Observe that a and b are regression coefficients and are calculeted as follows:

$b = \frac{\sum (t_i - \overline{t})(S_i - \overline{S})}{\sum (t_i - \overline{t})\hat{}2}$

$a = \frac{1}{n}(\sum S_i - b\sum t_i) = \overline{S} - b\overline{t}$

So far, for each time-consuming query q, PIM has already choose the best prediction model and the best prediction instance for this model. For that reason, it is able to apply the prediction instance to forecast the next execution of q.

4.4 Step 5: Index creation/drop in an automatic and proactive manner

After each prediction step is executed for a query q, the PIM component sends a message to DDL Generation requesting the creation of indexes belonging to I_q.

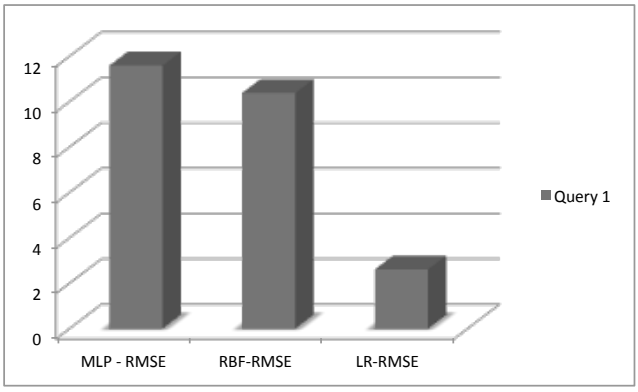

Fig. 6. LR, MLP and RBF RMSE for Query 1.

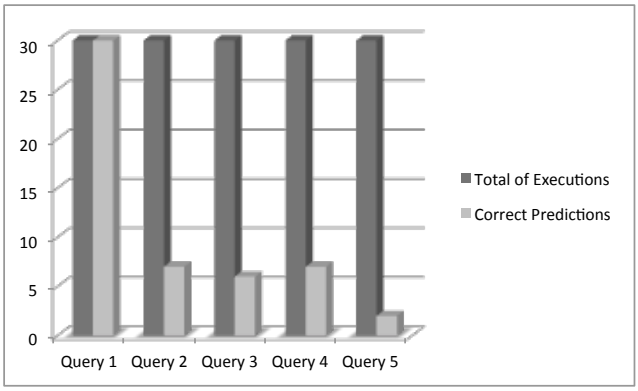

Fig. 7. Number of Correct Predictions.

4.5 Experimental Results

In order to quantitatively evaluate the proposed approach, we have performed a series of 30 predictions for each of the five time-consuming queries. For each query, we have used the best prediction instance, previously identified. To set the time window between the indexes creation and removal we have used $t = 3$ (Figures 2 and 4) and $k = 4$ (Figure 3). The Figure 7 illustrates the hit ratio (correct predictions) for each query.

From Figure 7, one can infer that when a suitable prediction model is identified for a given query execution, the results are promising, such as the case of Query 1. Furthermore, even when the prediction accuracy is low the approach provides gain. This is because, for heavy hitter queries (lasting more than 8 hours), one correct prediction (for example, reducing 5 hours in the response time of Query 3 in Table V) is enough to compensate incorrect predictions, since the overhead of an incorrect prediction is low, as we argue next. Additionally, we have investigated 3 prediction models, but may others could be evaluated and easily added to the proposed framework.

Table V illustrates the gains, which the proposed approach provides for each correct prediction. This table shows for each query: response time in the absence of index structures (RT), response time in the presence of appropriated indexes (RTI), the time needed to create the appropriated indexes (ICT) and the gain (G) provided by the approach for each correct prediction, which is obtained as follows: G = RT – (RTI + ICT). It is important to note that the gain provided by the proactive indexes creation for Query 3 is of 17100s, i.e., almost five hours. Thus, for this particular query its execution time is reduced by almost five hours in each correct prediction. Figure 8 shows the execution

Table V. Gain Provided by the Proposed Approach for each Correct Prediction.

Query	RT without Indexes	RT with Indexes	Indexes CT	Gain
Query 1	9360 s	4560 s	1200 s	3600 s
Query 2	14040 s	5820 s	3520 s	4700 s
Query 3	28800 s	8400 s	3300 s	17100 s
Query 4	16200 s	6600 s	2160 s	7440 s
Query 5	19200 s	6400 s	4080 s	8720 s

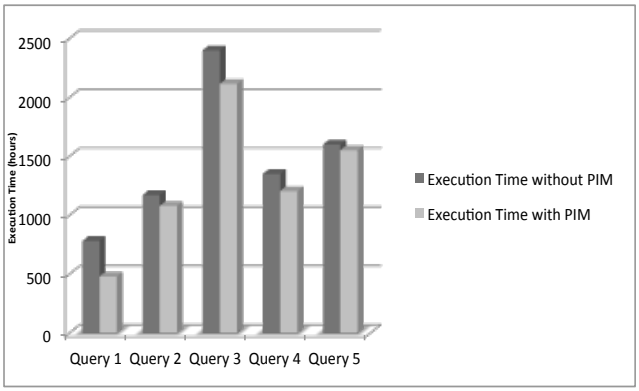

Fig. 8. Execution Time for the Serie with 30 Predictions.

time, in hours, for 30 predictions in two different scenarios: with and without PIM.

It is worthwhile to mention that the key idea behind the proposed approach consists of making the index creation/drop action asynchronous w.r.t. the query execution prediction. Creating index structures during DBMS workload peaks may compromise the database performance. In order to overcome this problem, the Scheduler (Figure 1) monitors the current DBMS workload to find the best moment to create index structures, that is, the instant which the DBMS workload is in a low operation level. Additionally, indexes creation are scheduled to t days (usually by night) before the predicted date (see Figures 2 and 4). For that reason, the execution time with PIM, depicted in Figure 8, does not include the time of building the indices.

Still, dropping an index structure consists of removing the index information from the DBMS metabase (catalog). In other words, a drop index operation involves no cost, since no physical operation is carried out. For that reason, the execution time with PIM mode in Figure 8 does not include the time to drop indexes. Therefore, the proposed approach presents low overhead for index creation or drop when a prediction fails. Note that the "without PIM" mode in Figure 8 indicates that all PIM's components are turned off.

5. CONCLUSIONS AND FUTURE WORK

In this work, we have presented a mechanism, denoted PIM, for proactive index management based on the use of prediction models. The key goal of the proposed mechanism is to identify time-consuming queries and index structures which may reduce the response time of such queries. Additionally, PIM forecasts based on a prediction model when time-consuming queries is going to be executed. By doing this, PIM may create index structures to reduce the response time of a consuming-time query q before the time t which has been predicted to run q. After q is executed, PIM drops indexes created to execute q.

Therefore, index are automatically created and dropped by PIM in a proactive manner. Experiments show that our method presents low overhead, can be effectively deployed to predict time-consuming query execution and provides performance gains, even if the created index are dropped as soon as

the predicted query runs. For instance, the experimental results show that using PIM the Query 3's execution time has been reduced by almost five hours (Table V).

We want to conclude this article by highlighting that the results of the experiments prove the feasibility and effectiveness of PIM. Besides, the developed mechanism can also be configured to operate in a semi-automatic manner, generating alerts on planned consultations, but leaving the decisions for DBAs.

REFERENCES

Agrawal, S., Chaudhuri, S., Kollar, L., Marathe, A., Narasayya, V., and Syamala, M. Database tuning advisor for microsoft sql server 2005. In *Proceedings of the International Conference on Very Large Data Bases*. Toronto, Canada, pp. 1110–1121, 2004.

Alagiannis, I., Dash, D., Schnaitter, K., Ailamaki, A., and Polyzotis, N. An automated, yet interactive and portable db designer. In *Proceedings of the 2010 international conference on Management of data*. New York, NY, USA, pp. 1183–1186, 2010.

Bruno, N. and Chaudhuri, S. An online approach to physical design tuning. In *Proceedings of the IEEE International Conference on Data Engineering*. Los Alamitos, CA, USA, pp. 826–835, 2007a.

Bruno, N. and Chaudhuri, S. Online autoadmin: (physical design tuning). In *Proceedings of the ACM SIGMOD International Conference on Management of Data Conference*. Beijing, China, pp. 1067–1069, 2007b.

Bruno, N. and Chaudhuri, S. Interactive physical design tuning. In *International Conference on Data Engineering*. Long Beach, CA, USA, pp. 1161–1164, 2010.

Chaudhuri, S. and Narasayya, V. Self-tuning database systems: A decade of progress. In *Proceedings of the International Conference on Very Large Data Bases*. Vienna, Austria, pp. 3–14, 2007.

Dageville, B., Das, D., Dias, K., Yagoub, K., Zait, M., and Ziauddin, M. Automatic sql tuning in oracle 10g. In *Proceedings of the International Conference on Very Large Data Bases*. Toronto, Canada, pp. 1098–1109, 2004.

de Carvalho Costa, R. L., Lifschitz, S., de Noronha, M. F., and Salles, M. A. V. Implementation of an agent architecture for automated index tuning. In *Proceedings of the International Workshop on Self-Managing Database Systems*. Washington, DC, USA, pp. 1215–, 2005.

Jimenez, I., LeFevre, J., Polyzotis, N., Sanchez, H., and Schnaitter, K. Benchmarking online index-tuning algorithms. *IEEE Data Engineering Bulletin* 34 (4): 28–35, 2011.

Luhring, M., Sattler, K.-U., Schmidt, K., and Schallehn, E. Autonomous management of soft indexes. In *Proceedings of the IEEE International Conference on Data Engineering*. Washington, DC, USA, pp. 450–458, 2007.

Maier, C., Dash, D., Alagiannis, I., Ailamaki, A., and Heinis, T. Parinda: an interactive physical designer for postgresql. In *Proceedings of the 13th International Conference on Extending Database Technology*. New York, NY, USA, pp. 701–704, 2010.

Monteiro, J. M., Lifschitz, S., and Brayner, A. A non-intrusive tool to automatic index maintenance. In *Demo Session of the Brazilian Symposium on Database*. Campinas, Brazil, 2008.

Morelli, E. M. T., Monteiro, J. M., Almeida, A. C., and Lifschitz, S. Automatic reindexation in relational dbmss. In *Proceedings of the Brazilian Symposium on Databases*. Fortaleza, Brazil, 2009.

Salles, M. A. V. and Lifschitz, S. Autonomic index management. In *Proceedings of the International Conference on Autonomic Computing*. Seattle, WA, USA, pp. 304–305, 2005.

Sattler, K.-U., Geist, I., and Schallehn, E. Quiet: Continuous query-driven index tuning. In *Proceedings of the International Conference on Very Large Data Bases*. Berlin, Germany, pp. 1129–1132, 2003.

Sattler, K.-U., Schallehn, E., and Geist, I. Autonomous query-driven index tuning. In *Proceedings of the International Database Engineering and Applications Symposium*. Washington, DC, USA, pp. 439–448, 2004.

Schnaitter, K., Abiteboul, S., Milo, T., and Polyzotis, N. Colt: continuous on-line tuning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data Conference*. Chicago, IL, USA, pp. 793–795, 2006.

Schnaitter, K., Abiteboul, S., Milo, T., and Polyzotis, N. On-line index selection for shifting workloads. In *Proceedings of the International Workshop on Self-Managing Database Systems*. Istanbul, Turkey, pp. 459–468, 2007.

Schnaitter, K. and Polyzotis, N. Semi-automatic index tuning: Keeping dbas in the loop. *Proceedings of the VLDB Endowment* 5 (5): 478–489, 2012.

Schnaitter, K., Polyzotis, N., and Getoor, L. Index interactions in physical design tuning: modeling, analysis, and applications. *Proceedings of the VLDB Endowment* 2 (1): 1234–1245, 2009.

Valentin, G., Zuliani, M., Zilio, D. C., Lohman, G. M., and Skelley, A. Db2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of the IEEE International Conference on Data Engineering*. San Diego, CA, USA, pp. 101–110, 2000.