Chapter LXXX
Self-Tuning Database Management Systems

Camilo Porto Nunes  
*Federal University of Campina Grande, Brazil*

Cláudio de Souza Baptista  
*Federal University of Campina Grande, Brazil*

Marcus Costa Sampaio  
*Federal University of Campina Grande, Brazil*

**INTRODUCTION**

Computing systems have become more complex and there is a plethora of systems in heterogeneous and autonomous platforms, from mainframes to mobile devices, which need to interoperate and lack effective management.

This complexity has demanded huge investments to enable these systems to work properly. It is necessary to invest on software acquisition and installation: management, administration, and update. These costs compound the *Total Cost of Ownership* (TCO), which tends to increase exponentially according to the software complexity.

Information Technology (IT) focuses mainly on providing information services in order to achieve simplicity, agility, large access to information, and competitiveness. *Database Management Systems (DBMS)* are part of the IT infrastructure in large, medium, and even small enterprises.

According to LIGHTSTONE (2003), the complexity of a DBMS enhances the difficulty in administrating it, as there are too many tasks a Database Administrator (DBA) should consider. Thus, more people are needed to be trained to efficiently manage this software. As a result, the maintenance costs increase and the problem is not properly solved. Performance, time-to-market, throughput, robustness, availability, security, are some concerns that a DBA should consider.

Suppose that, due to an unexpected event (e.g., too many clients accessing the DBMS), the average load in a database server changes quickly and the query response times may become unacceptable. The DBA must, rapidly (a) detect the problem (unacceptable response time); (b) find the root cause
of the problem (e.g., too many swap operations); and (c) tune the server to maintain response time goal (e.g., to increase memory size).

These three activities—symptom detection, diagnosis, and tuning—if executed manually by a DBA, can take a long time. Worse still, the DBA could be absent by the time the symptom arises.

On the other hand, autonomic computing has become an important research theme, aiming to reduce the manual efforts of administration of complex systems. Regarding DBMSs, they should be capable to react automatically to negative events, under the form of adaptive continuous tuning (LIGHTSTONE, 2003). For example, when detecting a load change, with risk of degrading the global performance of the system, a DBMS could automatically adjust some query execution plans; change its memory pool configuration; or still adjust data on disk (through index creation, data clustering, table partition, and so on) in order to maintain its overall performance. DBMSs with some capabilities of adaptive continuous tuning are called self-tuning ones.

This chapter addresses the issue of self-tuning DBMS. In the remainder of the chapter we present a background on this topic, followed by a discussion focusing on performance, indexing, and memory issues. Then, we highlight future trends and conclude the chapter.

**BACKGROUND**

Manual DBMS tuning requires a collection and analysis of some performance metrics by the DBA. Examples of such metrics include hit ratio in data cache, number of I/O executed in a given table, etc. After collection and analysis of these metrics, the DBA may decide whether it is necessary to adjust the DBMS. If an intervention is due, then the DBA needs to know which components are likely to be adjusted and what are the impacts of such adjustments.

In order to adjust the DBMS to improve performance, the DBA may act in several components such as memory buffers, index creation, SQL tuning, etc. In this chapter we focus on index and memory tuning, due to their importance in the overall performance of the DBMS.

**Memory buffers** consist of memory areas which are used for a given purpose. DBMS uses them, for example, to store the query execution plans, for both data and SQL statement caching, as an area for data sorting, and for storing session data. The tuning of these buffers may improve the performance of the submitted queries. For example, by increasing the memory buffer size, the number of physical I/O operations may be reduced, which results in performance gains.

**Index maintenance** is one of the most important tasks in DBMS administration. Hence, it deserves special attention due to its importance in the whole tuning process. An index may help the execution of a query submitted to the DBMS by reducing the number of accesses to disk. Nonetheless, during data update, insertion and deletion, an index may decrease the DBMS performance, as index reorganization may be necessary in such operations.

Once DBMS tuning involves many parameters, components and metrics, it is very difficult for humans to efficiently complete this task. Therefore, there has been a great effort from both academia and industry in providing automatic solutions for database administration.

**Automatic Diagnosis of Performance Problems**

The main motivation to automate the task of diagnosing performance problems concerns the complexity and importance for the subsequent decision making to solve the problem. The diagnosis, when executed manually, usually depends on DBA’s experience, because of the great number of metrics involved. Then it is difficult and ex-
pensive to proceed with analysis and correlation of such metrics in order to find the root cause of the problem.

Some related works (BERNOIT, 2005; DAGEVILLE, 2006) are based on two postulates: (1) it should be possible to define units of measure for different performance metrics which are either compatible or additive; and (2) both the many phases of processing user requests in charge of DBMS components and the many database resources used may be disposed in a hierarchy, so that drill-down / roll-up operations may be executed.

**DBTime** is an important concept in tuning, as it measures the time spent by a DBMS when processing user requests. Nonetheless, it does not include time spent in the intervening layers such as the network and middle tiers. **DBTime** includes tasks such as connecting to the DBMS, parsing SQL statements, optimizing SQL statements, executing SQL statements and fetching query results. The meaning of this concept is the measurement of the total amount of work done by the DBMS, and the **DBTime** rate is the DBMS load average. Figure 1 illustrates this concept.

Notice that DBMS (sub)components like ‘Query Parser’ and ‘Query Optimizer’ consume DBMS resources that include both hardware resources like CPU and I/O devices and DBMS resources like index, join algorithms, etc. The question to be answered is: in a given time interval, which DBMS components and hardware resources are being overloaded, in average?

To answer this question, the hierarchy of tasks for processing user requests as well as the underlying hierarchy of resources should be explored. A DBMS then could be modeled like an oriented graph, the nodes of which are either tasks or resources. More precisely, the root node represents the DBMS with its total **DBTime** rate; the second level represents the users connected to the DBMS during the time interval, with their portions of consumed **DBTime** rate; the other intermediary nodes represent either tasks or resources, with their respective percentual of the DBMS **DBTime** rate; finally, the leaf nodes are always atomic resources. Figure 2 presents such graph.

Considering that the DBMS has been correctly and extensively instrumented to obtain precise timing information — saying in another way, all components and resources have been correctly monitored —, the automatic detection and diagnosis of a performance problem, with the help of a **DBTime** graph consists of two steps: (1) to identify the leaf node(s) having a proportion of DBMS **DBTime** above a specified threshold; and (2) to roll up the oriented graph from the leaf node(s) with problematic **DBTime**. These identified nodes could indicate the root cause of the performance problem; soon after, by rolling up the

---

**Figure 1. DBTime**

<table>
<thead>
<tr>
<th>Session Connect</th>
<th>Query Submission</th>
<th>SQL Parsing</th>
<th>Query Optimization</th>
<th>Query Execution</th>
<th>Fetch Result</th>
<th>Disconnect</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1 (T_{u1})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User 2 (T_{u2})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User N (T_{un})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**DBTime:** \[ T_{u1} + T_{u2} + \ldots + T_{un} \]
Self-Tuning Database Management Systems

A diagnosis of the problem can be presented to the DBA. Inversely, if the DBA wants to know where is the bottleneck for a resource or component (intermediary nodes), the DBMS can find the answer by drilling down the graph from the resource or component until to reach the cause node.

AUTOMATED INDEX TUNING

Indexing tuning is not a trivial task. Frequently the DBA does not have enough time to analyze index metrics, whether it is interesting to create a new indexing or remove an existent one. Hence, automatizing this process seems to be a very good approach.

The basic idea consists of submitting to the DBMS a set of statements, named workload. The self-tuning index manager will analyze the workload, according to criteria such as query filters, tables and attributes, table joins, index selectivity, and so on. Then, two decisions may arise: 1) the hint to the DBA to create or delete some indexes; or 2) to automatically create or remove indexes, that is, without DBA intervention.

To find out whether a given index may improve performance in an execution of a SQL statement, the index manager may use the index statistics or virtual indexes. The procedure is done as follows: for each SQL statement from the workload, the index manager analyzes its structure and selects a set of indexes, known as candidate index set, which may give a gain in performance. Then the information about the candidate indexes are inserted in the DBMS catalogue, but their respective data structures are not created (that is why they are called virtual indexes). This is done in order to enable the virtual indexes to be taken into account by the query optimizer. If during the generation of the query execution plan, a virtual index is invoked, then this index will be either suggested to be created by the DBA, or automatically created.

There are some variations of the previous procedure due to side effects which may be generated. One of them is related to local optimization, once those indexes are analyzed individually for each statement in the workload submitted to the DBMS. This may result in an index providing an excellent performance for a specific SQL statement, but unacceptable slow for other SQL statement.

Figure 2. A DBTime oriented graph
Self-Tuning Database Management Systems

That is, the index local analysis does not assure a global optimization in the workload. In the following we discuss some research works which proposes self-tuning indexing techniques.

SATTLER (2005) propose an access-balanced index structure, Simple Adaptable Binary Tree (SABT), instead of traditional data-balanced ones. Their experimental evaluation has evidenced that the benefit of SABT is greater for data accessed more frequently than for data accessed less often. Essentially, a SABT has an adjustable size, i.e. the number of nodes can be increased or decreased during runtime, and it is deeper for data which are accessed more often to minimize page access. Moreover: (1) the index is sparse; (2) the leaf nodes are page containers, which hold pages of tuples with keys of the value range specified by the tree; (3) page containers keep local page access statistics and (4) the lookup in the tree works as usual.

The crucial issues of reorganizing and balancing is resolved by observing if a page container is more or less accessed; in other words, if its access counter is above or below a certain threshold. In the first case, if there are free nodes, a new node with the median key from the container, and the container is split into two according new containers. By judiciously configuring the free node pool variable, the tree can always grow, when necessary. On the other hand, shrinking can be done the same way. Finally, these reorganization as well as statistics updates are performed during lookup operations.

The main drawback of this approach seems to be the fact that binary trees are not the optimal choice for a DBMS index structure. Other open issues include concurrency (hot spots), implications for secondary indexes and multidimensional indexing schemes, and operations based on access-balanced index structures.

Due to space limitations, we now shortly comment some other related works. LIFSCHITZ (2005) and COSTA (2005) present an architecture based on agents to implement the automatic management of indexes. DAGEVILLE (2004) highlights the Oracle 10g component which is responsible for SQL tuning and it recommends the creation of indexes to improve the performance of the top queries. SCHNITTNER (2006) implements an index manager for the PostgreSQL DBMS and it follows the same ideas discussed in this section. AGRAWAL (2005) proposes an index manager for the SQL Server 2005 which aims optimal optimization of a workload submitted to the DBMS. SATTLER (2003) presents a solution for indexing automatic management which does not need DBA intervention.

Automatic Memory Management

Memory is considered one of most used resources used by DBMSs. Obviously, this is due to its fast access when compared to disks. For this reason, DBMSs try to automatically place in the memory the most frequently used objects to minimize disk read/write operations (MULLINS, 2005).

Nonetheless, several kinds of data compete for memory including application data, execution plans of frequent queries, stored procedures, concurrency locks and log buffers. Furthermore, some memory areas should be allocated and reserved for special types of processing, such as data sorting, join operations, temporary objects and so on.

Once the memory used by DBMSs is limited, adjustments in any memory buffer unavoidably will affect other buffers. Hence, memory tuning is a tricky task.

The memory allocation problem can be defined as follows: given a workload and a fixed and limited amount of memory, it is necessary to determine the size of each specific buffer area in order to optimize overall performance. This would demand from the DBA a strong expertise. Moreover, sometimes DBA’s experience is not enough to succeed in memory optimization. The
reason for that is that the time spent by DBA to make an analysis of the workload parameters and, soon after, to decide which adjustments should be made can be quite long. Thus, it may compromise the quality of the DBMS tuning.

Due to these difficulties, it is necessary to implement a DBMS autonomous component that is able to manage memory buffers automatically, efficiently and effectively.

There are several research works on automatic memory management. XI (2001) presents an analytic model, using chains of Markov, for prediction of the hit ratio of the data cache of a DBMS. POWLEY (2005) describes an autonomous component for administration of the PostgreSQL memory buffers. The rationale for that component consists of monitoring the average time of I/O requests; in case the I/O resource suffers an increment of 5% in its average value, an algorithm of memory buffer configuration is immediately executed.

BROWN (1996) has proposed a technique which is used to assist response time requirements for different classes of transactions submitted to a DBMS. The solution consists of allocating data used by different classes of transactions in different data caches (fragments). This approach assumes the proportional hypothesis between buffer miss ratio and response time. Later on, this technique was improved with the incorporation of a model for estimating buffer hit ratio. The new model is based on the Theorem of the Concavity which states: “... the hit ratio, as a function of the allocation of memory buffer, is a concave function”. Hence, a policy for buffers allocation may be mathematically optimum.

MARTIN (2006) presents an approach to map database objects into different data caches with the goal of maximizing the database transaction throughput. The algorithm takes into account the access patterns and inherent characteristics of database objects for a given workload. Similar objects are then grouped in the same buffer pools. The amount of memory allocated to each buffer pool is determined according to the size of the database objects that will reside in the respective buffer pool. The approach uses data mining techniques to analyze and discover access patterns for database objects.

To summarize, state of the art on automatic memory management presents some open issues:

1. Current techniques have focused only on buffer pool but they have not addressed other important memory areas such as those used for statements, sorting, and temporary data;
2. Lack of a holistic approach which would address not only memory but also other DBMS components such as indexes, disks, and optimizer;
3. There are few works on open-source database systems, once many contributions focus on DBMS proprietary architecture;
4. Lack of standards in the monitoring process; and
5. Lack of a framework which may monitor heterogeneous DBMS.

FUTURE TRENDS

There are still open issues related to the ability of a DBMS to execute DBA tasks automatically. Scientists are seeking alternatives to offer larger autonomy to DBMSs. According to LIGHTSTONE (2003), besides the characteristics discussed up to now, future DBMSs will have to incorporate some essential characteristics of autonomic computing (KEPHART, 2005) to reduce their maintenance and administration costs. This tendency aims, in a holistic way, to turn DBMSs capable to automatically accomplish any task, besides those related to performance administration. That is, DBMSs, instead of self-tuning, would become self-manageable.
The first characteristic to be incorporated is the continuous adjustment and adaptation of a DBMS face to changes of any type. More specifically, the DBMS should monitor itself to detect changes occurred in the workload and to adjust parameters so that some of its components may adapt to the changes. Query optimizers should automatically search for new execution plans, memory buffers should be reallocated to support an abrupt change of workload, and the physical database design should be altered in an automatic way to create new indexes, to remove undesirable indexes, to partition tables, to materialize views, etc.

The second characteristic that a DBMS should have is the capacity of discovering the workload type, for example, if it is either OLTP or OLAP. As it is known, these two workload types have quite different characteristics, demanding different database design and implementation.

The automatic detection and integration of new sources of data is also an important issue. Many human and financial resources have been spent to integrate different sources of corporation data. In this sense, it is highly recommended that DBMSs be able to discover where new sources of data are and how to integrate them into the corporative database.

Concerning the physical storage, a DBMS should be able to optimize disk usage in order to reduce data storage costs. The aim is to save physical space when this is restricted (e.g. by removing indexes or avoiding materialized views), or then to use the available disk free space (e.g. by creating indexes and materialized views).

DBMSs in the close future should be able to automatically react to eventual exception conditions, such as high disk occupation, or lack of enough memory. Also, functions such as sub-minute automatic failover (LIGHTSTONE, 2003) allow for preventing to detect situations in which there will occur exceptions: this way, DBMSs will become pro-active stricito sensu.

Last but not the least, DBMSs, besides explaining the reason of their decisions, should interact with DBA to receive his feedback, so that DBMSs can improve their decisions.

CONCLUSION

This chapter has addressed the importance of self-tuning DBMS. We have highlighted the main difficulties for a DBA to properly manage a DBMS currently, due to the large complexity of the latter. Then, the main issues on self-tuning DBMS — performance, indexing and memory — have been discussed. Lastly, future trends on self-tuning DBMSs have been examined: the main verification is that self-tuning DBMSs should evolve for self-manageable DBMSs.

Self-tuning databases are a very hot research topic which has been investigated by large database companies and academy. It is part of the next generation of computer systems based on autonomic computing principles.

REFERENCES


Self-Tuning Database Management Systems


KEY TERMS

**Cache Hit Ratio:** Is a ratio of buffer cache hits to total of requests, that is, the probability that a data block will be in memory on a subsequent block re-read.

**Cache Miss Ratio:** A ratio of the number of times a DBMS cannot find a given data in the memory.

**Database Administrator (DBA):** Professional responsible for DBMS administration.

**Database Management Systems (DBMS):** Software responsible for management of data in a corporation.

**Index:** Data structure which enables fast access to data stored in disks.
**Self-Tuning Database Management Systems**

**Memory Buffer:** Area of memory which is allocated for a specific destination, such as data caching, session caching, program caching, sorting, and temporary area.

**Self-Manageable Database:** DBMS which is able to self-management, that is, it is able to execute the DBA’s tasks automatically.

**Self-Tuning Database:** DBMS which is able to automatically administrate and tune its performance.