

Implementation of an Agent Architecture for Automated Index Tuning

Rogério Luís de Carvalho Costa, Sérgio Lifschitz,
Maíra Ferreira de Noronha and Marcos Antonio Vaz Salles
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio) - Brazil
{rogcosta,sergio,maira,mvsalles}@inf.puc-rio.br

Abstract

It has been important to extend database management systems to support new requirements of applications and administration. We focus in this paper on the automatic tuning feature, particularly manipulation of indexes. The architectural approach considered here is based on software agents. We briefly present the heuristics involved in the decision upon creation, dropping and the exact moment to proceed with either action. Then we discuss our agent-based database architecture, with its layers and coupling within an actual DBMS. Finally, we explain the way our hypothetical indexes are integrated in PostgreSQL and how they can be used in practice for automated index tuning.

1 Introduction

Database tuning is a quite complex activity, usually carried out by database administrators [18]. For current systems, these fine manipulations may be needed for hardware configuration, physical design and query specifications, among others. At each new edition of commercial DBMSs, additional operational parameters appear to be adjusted. Therefore, tuning has become very important and, at the same time, more expensive, as highly specialized professionals are needed.

It is part of the tuning process to perceive that a given system resource is not efficiently used, diagnose its causes and take corrective actions. Database systems would, ideally, be configured in such a way that no tuning activities would ever be needed. All the resources would be always placed and adjusted in the best possible way. When there are mandatory changes, however, the DBMS would then be able to execute them automatically. This capacity of perception and automatic adjustment is known as self-tuning.

Due to the high complexity of DBMSs' current imple-

mentations, self-tuning solutions are still very restricted. We need to adapt the DBMSs' architecture in order to achieve an actual automatic behavior (e.g. [5]). In this paper we consider the use of software agents to deal with self-tuning and DBMSs operational requirements.

The software agents' research area aims at building systems that deal with heterogeneous, distributed and dynamic environments [19]. When agents are present, systems can detect the external environment where they are inserted and react in different ways according to the existing system configuration .

We are mainly concerned here with a particular kind of autonomic behavior: automated index tuning. We present an approach that enables the automation of the complete index self-tuning process, requiring no human intervention. Our approach matches agents systems and DBMSs in a feasible architecture. Indeed, we use a software agent coupled with the DBMS optimizer to manage indexes.

Our agent architecture is implemented within PostgreSQL [16], an open source DBMS. We show that, given some particular heuristics for decision upon creation and dropping of indexes, the agents enable autonomic management in a very effective way, through the use of hypothetical indexes integrated with the DBMS optimizer. A hypothetical index exists only in the database schema and is not actually materialized [11]. We also evaluate the quality of estimates when compared to optimizer values for actual indexes.

In the next Section some previous works directly related to ours are given. Then, we briefly show the heuristics used within the self-tuning engine to support automated index management in Section 3. In Section 4 we present ways software agent systems may interact with database systems and our specific choice. Then, in Section 5, we detail implementation issues regarding our self-tuning agent and its coupling with the PostgreSQL source code. In Section 6 we discuss hypothetical indexes, which enable the feasibil-

ity of our approach. Section 7 lists our main contributions, ongoing and future work.

2 Related Work

There are many approaches to automatic index selection described in the literature. They mostly focus on constructing tools with detailed rules that encode the knowledge of good database design. Due to space limitations, we will comment briefly some of the most important works related to this paper. For a more thorough discussion of work done on self-tuning for relational database systems, see [14].

In [10] the proposed index selection tool makes use of the optimizer's cost estimates to compare the gains of alternative hypothetical designs. This is a key observation because it avoids asynchrony between the cost models used by the tool and the system. The work of [11] proposes to extend the optimizer's interface to allow the experimentation of alternative hypothetical index sets.

Tools for indexes suggestion implemented in Microsoft's SQL Server are discussed in [3, 4], as part of the AutoAdmin project. The objective of the index suggestion tools is to generate an index set for a given input workload, obtained by the DBA. The workload is broken into single statement inputs and candidate indexes for each statement are generated. These are, then, arranged into configurations and their costs are evaluated by the query optimizer. A greedy algorithm extends the number of statements and index configurations considered until a best index configuration is determined to the workload as a whole. As not all the possible indexes evaluated exist in the database, a separate module enables the simulation of hypothetical indexes. Additional work on index selection heuristics done by the same authors is documented in [2].

The work of [15] suggests that the index selection heuristic should be tightly integrated with the optimizer. The optimizer itself is extended with an index suggestion mode. This means that, before the optimization of a given query, hypothetical indexes for all relevant columns are generated. Then, the indexes picked by the optimizer are recommended for the query. The single query recommendations serve as input to an index selection heuristic that tries to find the best index configuration for a given workload.

There still exists some other commercial tools (e.g. [8, 20]) that are capable to recommend indexes for a workload. However, all of these previous proposals do not address the automation of the complete cycle of workload collection, index selection and actual data structure creation or destruction. Indeed, they need the intervention of the database administrator, who not always has all the information and tools needed to characterize the system's workload in an efficient manner.

Therefore, the final decision of index creation or destruction requires human intervention. Our work studies a possible solution to these problems through the use of a software agents architecture. The idea is to completely automate the index tuning process. The architectural approach that makes this feasible will be detailed in the next sections.

3 Heuristics Overview

As the main focus of this paper is on implementation issues, we briefly describe in this section the basic heuristics that allow the total automation of index choice, creation and destruction, during the DBMS's normal operation. This is done through the use of a self-tuning agent called *Self-Tuning Agent Based on Differences* that is coupled with the DBMS.

The basic database objects used are the hypothetical indexes. These exist only in the database catalog and are not materialized [11, 4]. Hypothetical indexes are used by the optimizer to recommend an execution plan that would use such an index if it was physically created. Analogously, for indexes that should be dropped, the optimizer should be extended to be able to find an execution plan that disregards actual indexes present in the database.

Our agent senses each query submitted to the DBMS, creates several hypothetical indexes that can be interesting to the query, and selects the indexes that bring the greatest benefits as measured by the optimizer's cost estimates. The indexes chosen are then considered as candidate indexes to our differences evaluation procedure.

The enumeration of candidate indexes has been studied in other works in the literature [10, 11, 3, 15]. We have chosen to implement the SAEFIS ("Smart column Enumeration for Index Scans") heuristic detailed in [15] because of its simplicity and efficiency. The SAEFIS heuristic has also been implemented in DB2 Universal Database.

In our model, for one given operation submitted to the DBMS, the query optimizer generates the best plan with the best real configuration, that is, it only uses objects materialized in the database.

Then, the self-tuning agent enumerates hypothetical indexes that may speed-up the operation. When such indexes do not exist, the execution of the operation continues normally. When good hypothetical indexes exist, these are created in the database's schema and a plan is generated according to a hypothetical configuration.

The costs of the best plan according to the real configuration (C_R) and of the best plan according to a hypothetical configuration (C_H) are compared and a decision on creation/destruction of an index is taken.

```

C = CR - CH ;
If C > 0, then
  If C > CC, then
    Create Indexes & Execute the query;
  Else
    Execute the query;
    CP = CP + C;
    If CP + C >= CC then
      Create indexes & Reset CP;
    End if;
  End if;
Else
  CP = CP + CRI ;
End if;

```

Figure 1. Queries and previous operations

Query evaluation

When evaluating queries, a factor C is defined as the difference between the cost of the best plan according to the real configuration and the cost of the best-established plan according to hypothetical configurations. When positive, we compare it with the cost of creation of the necessary indexes used in the hypothetical configuration, called CC (creation cost). If C is lower than CC , then it will be advantageous to create the indexes before query execution. On the other hand, an evaluation of previous operations will be carried through, in accordance with the stored statistics. This procedure is represented in Figure 1.

The evaluation of the previous operations is extremely simple. It consists on deciding if an index should be created or not by analyzing the time that could have been saved in the last queries in consequence of the existence of such index. The creation decision is taken when the total costs that could have been already saved, in case that index existed, reaches the forecasted index creation cost. This is represented in Figure 1, where C_P is the stored value representing the total cost's reduction if the related index had already been created.

When C_H is equal or bigger than C_R , the query is executed according to the real configuration. If the optimizer always finds the best plans, C_H would never be bigger than C_R , as C_H would represent the optimal plan. We should consider that the indexes that are used in the query execution plan, in the real configuration, are reducing the query execution costs. So, we should add to C_P another factor, C_{RI} , which is the cost of execution if the indexes used in the real configuration were dropped.

Update evaluation

The case of updates and exclusions is a little more complex than the one of queries: actions that could be taken include create and/or destroy an index.

We call CD the destruction cost of an index. When C is

```

C = CR - CH ;
If C > 0, then
  If C > CC + CD, then
    Create proposed indexes;
    Drop indexes proposed to be dropped;
    Execute the query;
  Else
    If CHC > CC then
      Create proposed indexes & Execute the query;
      Reset CP;
      Exit;
    End if;
    If CHD > CC then
      Drop proposed indexes & Execute the query;
      Reset CP;
      Exit;
    End if;
    Execute the query;
    CPC = CPC + CHC ;
    CPD = CPD - CHD ;
    If CPC >= CC then
      Create index & Reset CPC;
    End if;
    If |CPD| >= |CC| then
      Drop index & Reset CPD;
    End if;
  End if;
Else
  CP = CP + CRI ;
End if;

```

Figure 2. updates and previous operations

negative or equal to zero, the actions to be taken are similar to those taken in the query operations evaluation, just described. On the other hand, when C is positive, we should evaluate if C will be greater than the sum of the indexes creation costs and the indexes destruction costs ($CC + CD$). If it is, then we create/destroy the indexes and carry through the query. If C is lower than $CC + CD$, then we should obtain C_{HC} that is the cost of execution of the operation if the suggested indexes are created but no index is dropped. C_{HC} should be compared to CC . If C_{HC} is lower than CC the indexes are created and the update/delete is executed. If C_{HC} is not lower than CC , then we should obtain C_{HD} . C_{HD} is the cost of execution of the operation if the suggested indexes are destructed but no index is created. If C_{HD} is lower than CD the indexes are dropped and the update/delete is executed.

The other way around, we only make an evaluation of previous operations. The evaluation of previous operations for the case of an update/delete is composed of two parts: (i) for the indexes which were proposed to be created, C_P is increased of the value of C_{HC} ; (ii) for the indexes which were proposed to be dropped, we update the value of C_P deducting C_{HD} . When C_P is positive and $|C_P|$ reaches the cost of indexes creation, these are created. When C_P is negative and $|C_P|$ reaches the cost of indexes creation, these are destroyed. The procedure for update/delete operations is represented in Figure 2.

4 Agent-based Databases

There are multiple definitions for the agent term but consensus exists only for the characteristics of autonomy. Indeed, this is a central point related to the agency concept. We consider here basically the definition given in [19]: an agent system is a computational system that lives in a given environment, being able to execute autonomous actions over this environment in order to achieve its goals.

The agent architecture implemented here is based on an object-oriented framework for building agent systems proposed in [12]. This framework (Figure 3) defines a layered architecture which identifies each agent function and can be used for both simple and complex agents. We shall discuss how each of the layers presented were implemented for index tuning in Section 5.

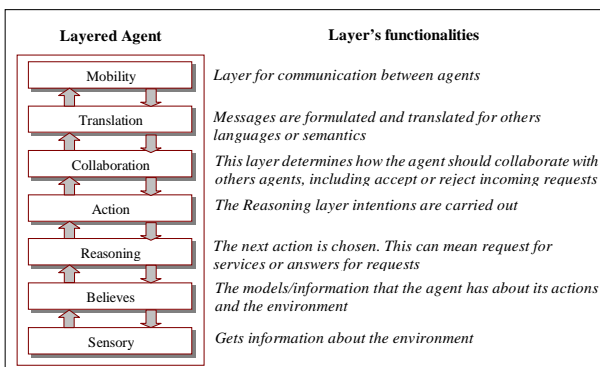


Figure 3. Layered architecture for agents

Some works investigate the use of software agents to extend DBMSs functionalities in general [1] and also specifically for self-tuning [9]. We have observed in our implementation that the agent abstraction effectively permits the introduction of self-tuning functions in the system with small impact in DBMS components and architecture.

If one opts not to use software agents, changes to include system data collection and autonomic system modification functions must be done directly to existing or newly added DBMS components. This means trying to alter the architecture of an already complex system to include new functionality for which it was not originally designed. The complexity of this task must not be underestimated. Indeed, some argue that a more compelling path would be to completely rethink DBMS architecture and implement simpler, RISC-style systems [5]. In our work, we show that it is possible to take a less radical approach to coding self-tuning functionality by using software agents to set apart existing DBMS components and novel tuning algorithms.

In [13] three integration architectures between agents and DBMSs are proposed: Layered, Integrated and Built-in. Each one of the three integration architectures has ad-

vantages and disadvantages. The Layered architecture is the one implemented in most existing approaches but is also the one where less functionalities are supported. In the Integrated architecture the maximum agency level is obtained, as agents systems replace all (or almost all) of the DBMSs' components. However, building such an integrated system is extremely complex. The Built-in architecture enables the reuse of DBMSs' existing components. The degree of extension of DBMSs' functionalities depends on the coupling level between agents and components.

We have considered the built-in architecture to implement the self-tuning feature in an agent-based database architecture, as will be presented next.

5 Self-tuning Agent Implementation

We discuss in this section the way our agent was integrated with the DBMS's code and some important choices made during the implementation that made our solution feasible.

Agent Layers

Our agent interacts with the DBMS to obtain the queries and updates submitted to the system. For each statement processed, the agent updates its beliefs on index accumulated costs and evaluates new indexing possibilities. This evaluation may involve interactions with the system's optimizer and enumeration of hypothetical indexes. After indexing alternatives are examined, indexes may be created or destroyed. The agent executes these actions by making calls to the DBMS's internal components.

We have followed the agent architecture presented in Figure 3 to implement our index self-tuning agent. A detailed object-oriented design for the layers is discussed in [7]. We have mapped the desired index agent's functionalities to each of the layers as presented below:

1. *Sensory*: this layer is responsible for obtaining information from DBMS's components. The agent must receive each statement processed by the system's optimizer as well as cost, table and actual index information for the statement. In order to achieve this, we propose that DBMS code be instrumented to send information to the agent after statement optimization but before execution. We can implement this kind of information collection in an efficient manner using a notification mechanism. This represents less overhead when compared to the alternative of using polling to obtain system information. Additionally, if we standardize information gathered from the DBMS, we can ease the integration of the index tuning agent with different database systems.

2. *Believes*: this layer stores information on database tables, actual and candidate indexes. This information is necessary to calculate index selection heuristics implemented in the *Reasoning* layer.
3. *Reasoning*: this layer evaluates agent's beliefs and enumerates possible courses of action. The *Self-Tuning Agent Based on Differences* uses two heuristics. The first heuristic performs hypothetical indexes enumeration and chooses candidate indexes for each statement. In order to simulate the existence of hypothetical indexes in the database, the *Reasoning* layer invokes action execution procedures available at the *Action* layer. The candidate indexes found are stored on the *Believes* layer of the agent. The second heuristic then decides which indexes should be created and which should be destroyed. Actual index modification actions are executed through calls to the *Action* layer. Refer to Section 3 for a brief explanation of the index selection heuristics used by our agent.
4. *Action*: this layer carries out the modifications chosen by the *Reasoning* layer. For the index tuning agent, the layer must offer capabilities for hypothetical indexes simulation and actual indexes creation/destruction. As an implementation of these functions requires direct interaction with the DBMS, the procedures on this layer shall fire effectors specifically coded to the DBMS used.

The *Collaboration*, *Translation* and *Mobility* layers are responsible for dealing with communication among agents in distant societies, that may interact with distinct protocols, languages and message formats. The implementation of these layers might be helpful for dealing with multiple agents, possibly distributed in various databases. These various agents can negotiate in order to achieve global tuning decisions that take into consideration the trade-offs among possible adjustments. In this paper, we study the implementation of an individual index tuning agent integrated with a centralized database. Therefore, the *Collaboration*, *Translation* and *Mobility* layers will not be further discussed.

Integration with PostgreSQL

Figure 4 depicts how the *Self-Tuning Agent Based on Benefits* was integrated with PostgreSQL. We use a Built-in architecture, in which the agent is compiled along with other DBMS's components.

We had to make the agent interact with PostgreSQL's server process model. The DBMS follows a client/server structure. Each client process is connected to exactly one server process. As the quantity of connections is not known at the outset, one master process (*postmaster*) is created

when the database is started. For each connection request, a new server process (*postgres*) is created. All *postgres* processes communicate with each other using semaphores and shared memory in order to ensure data integrity.

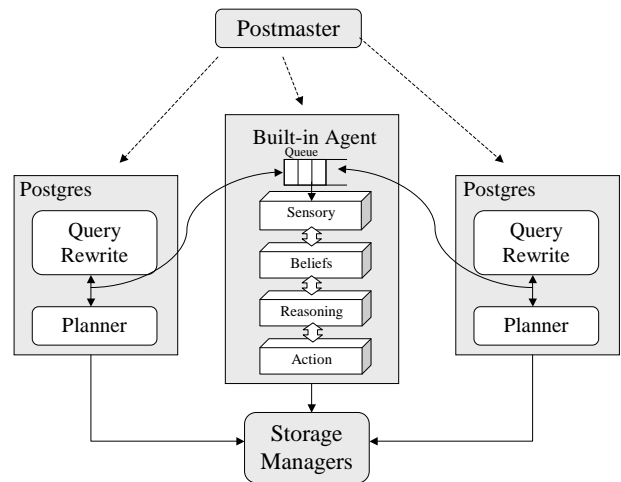


Figure 4. Agent and PostgreSQL

Every *postgres* process contains instrumentation code that interprets the statements being processed and records column usage information to a shared queue. The agent senses information present in this queue, updates its beliefs, evaluates index selection heuristics and, if necessary, executes actions. These actions can make invocations of internal database routines.

A few challenges had to be overcome to accomplish this level of integration. While the agent has been coded in C++, PostgreSQL is completely written in C. Therefore, we had to create interface functions and structures in order to compile the agent and the database system together [6].

Another important issue was process synchronization. The self-tuning agent is a server process that must have access to information present in the query trees of each statement submitted to the optimizer. Therefore, we have introduced an operating system message queue that holds the query information that should be considered by the agent to apply its heuristics.

As the agent is compiled along with the database system, it has access to all DBMS functions. When an action must be carried out in the system, such as index creation or destruction, the agent's *Action* layer invokes interface routines that call the appropriate DBMS's internal functions. This has been possible because we have initialized the agent process as a regular *postgres* process is initialized. Therefore, the agent process is transparently capable to deal with all shared DBMS structures, such as the lock table and the catalog cache.

6 Hypothetical Indexes Implementation

In order to properly simulate the pros and cons of an index configuration, we have extended the optimizer's interface to recognize the concept of hypothetical indexes. We have created new SQL statements in PostgreSQL to allow the manipulation of hypothetical indexes. There are two commands to create and drop hypothetical indexes and also a third command that is used to generate a query plan that acknowledges the existence of both hypothetical and actual indexes.

[CREATE,DROP] HYPOTHETICAL INDEX

In order to support these commands, we have modified some of PostgreSQL's modules. First, we altered the system's parser to recognize the new syntax. For the *create hypothetical index* statement, we have included a parameter in the query tree indicating that the index to be created is a hypothetical one. When the query tree is sent to execution, a reference to the index is created on the system catalog, but the index is not actually materialized. A new column has been added to the catalog to inform if the index is hypothetical or not. As we will see, the *explain* and the *explain hypothetical* statements will use this new column to distinguish hypothetical indexes from actual ones.

Instead of adding a new column to the system catalog, we could have chosen to create a user specified table to represent hypothetical index information, as was done in [4]. This option could bring some benefits in terms of contention on the system catalog, since every hypothetical index creation demands a row exclusive lock in the catalog. Exclusive locks may have a negative impact on system performance because the agent tries to enumerate hypothetical indexes concurrently with the processing of normal user queries. Using separate tables for hypothetical information is also interesting if more complex simulations are to be done with the optimizer, such as multiplying the cardinality of user tables by scaling factors.

In spite of such potential advantages, creating separate user specified tables makes the impact on database server code greater. The DBMS's code must be altered in many distinct points in order to make the optimizer aware of where to get its statistics and schema information from.

Another important factor to consider is that actual index creations acquire a shared lock on the underlying table, thus avoiding the execution of concurrent updates on the table. Hypothetical indexes are not materialized and, therefore, we have used less restrictive lock levels for the creation of our simulated structures. We only block concurrent data definition language operations.

To drop an hypothetical index, we have included in the system a *drop hypothetical index* command. As we have

chosen to represent hypothetical indexes in the system catalog, this new statement is very close to the conventional *drop index* statement. The main difference in the routines is, once again, the acquisition of a less restrictive lock level on the underlying table. This similarity of the hypothetical indexes logic to the logic used by actual indexes is an implementation advantage of using the system catalog to store the hypothetical indexes metadata.

EXPLAIN HYPOTHETICAL

The *explain hypothetical* statement is an extension of the conventional *explain* command [10]. We have included a parameter in the *explain* statement query tree to indicate to the optimizer if hypothetical configurations should be evaluated. In *explain hypothetical*, the resulting query plan generated by the optimizer may include both hypothetical and actual indexes.

The modifications that had to be made to the optimizer were very localized. Basically, when the system tries to obtain optimization information about a relation, hypothetical indexes are also included if and only if the optimizer is working in the *explain hypothetical* mode.

One important concern that applies to the creation and use of hypothetical indexes is statistics management [4, 15]. Some statistics approximations were necessary to generate cost estimates for hypothetical indexes. In PostgreSQL's cost model, indexes make use of the following statistical components: selectivity, correlation, number of tuples and number of pages.

Selectivity and correlation estimates in PostgreSQL do not involve specific index information, so generic routines can be used both for hypothetical and actual indexes yielding the same results. Therefore, no extensions were necessary to estimate these parameters.

We have approximated the number of tuples in an index by the number of tuples in the underlying table. This is always true for dense indexes, which is the type of indexes that we deal with in PostgreSQL. We have followed the same policy for the number of pages in an index, approximating it by the number of pages in the underlying table. For most index creation scenarios, this will yield conservative cost estimates for index scans, as a table tends to be larger than any one of its individual indexes. On the tests we have conducted, the cost values estimated for hypothetical indexes and its materialized counterparts have been reasonably close (see the discussion in the next subsections).

As the implementation we made is general, we can use the *explain hypothetical* command combined with any of PostgreSQL's DML statements. Hypothetical indexes are a basic infrastructure to index selection. Therefore, our implementation of hypothetical indexes in PostgreSQL not only enabled the coding of the *Self-Tuning Agent Based on*

Benefits, but also permits that other developers create index selection tools similar to the ones proposed in [3, 15] for commercial databases.

Hypothetical Indexes Evaluation

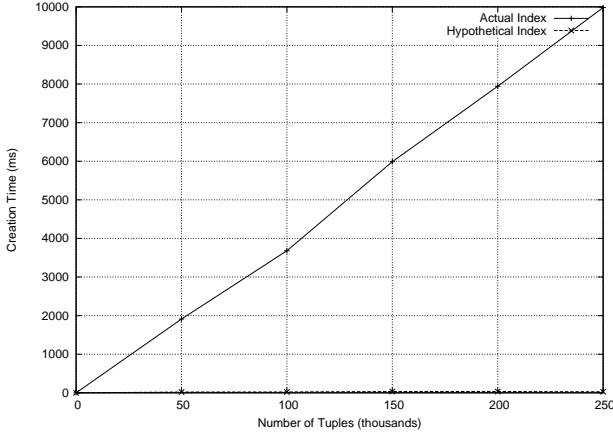


Figure 5. Hypothetical vs actual indexes

We have made some experiments to validate the performance gains involved in using hypothetical indexes and also to evaluate the quality of the estimates made compared to optimizer estimates for actual indexes. In our first experiment, we have investigated the time difference between creating a hypothetical and corresponding actual indexes. This difference in time justifies the usage of hypothetical indexes as a simulation tool for large databases.

We have populated an example table, called *sales*, with growing numbers of tuples and have defined an index to be created on the table (due to space limitations, please refer to [17] for more details on the creation of the test scenarios). For each size of the sales table, we have taken ten time measurements for hypothetical and actual index creation. The average of those ten measurements corresponds to a data point in Figure 5. Time measurement was accomplished through the use of the *timing* option of PostgreSQL’s *psql* client [16]. Both client and server were run on the same machine and, therefore, no network delays were introduced in the values obtained.

As we can see in Figure 5, actual index creation times grow linearly with table size. Hypothetical indexes, on the other hand, have creation times that remain practically constant and equal to a few hundredths of a second in the hardware we have used. A database user or administrator can use hypothetical indexes to evaluate the potential benefits to queries with very little computational costs inflicted to the system.

Another important point is the estimation quality that the optimizer achieves when hypothetical indexes are used. It

Query Type	Query Text
Point query	select * from sales where s_num = 100;
Range and aggregation query	select s_prod_num, s_date, sum(s_value) as total from sales where s_value > 1500000 and s_date between '20040101' and '20040131' group by s_prod_num, s_date;
Ordering query	select * from sales order by s_num;

Table 1. Samples queries

must be comparable to the quality achieved for actual indexes. Therefore, we have made additional experiments to validate if the optimizer would find the same query plans in the presence of hypothetical indexes and their actual counterparts.

We have selected some types of queries that make use of indexes (for a classification, see [18]). For these queries, we have used the optimizer to calculate the cost to process the query with a convenient hypothetical index and the cost to process the query with the same actual index.

Table 1 shows a sample of some of the queries we have experimented with. As we have approximated the number of pages in a hypothetical index by the number of pages in the underlying table, we have picked queries that scan different numbers of pages from the index.

The first query is a point query, that is, a query that accesses only one tuple from the table. The sales number attribute is a sequential identifier for sales entry in the system. An index on this attribute can bring benefits to the query. For this query, cost estimates made with hypothetical and actual indexes were identical. The number of pages estimated for the hypothetical index has no impact as only one page from it will be accessed.

The second query makes a range selection and then an aggregation. An index on the date and value attributes of the sales table could speed the query up. For this query, the differences in estimates for hypothetical and actual indexes were small and equal to 0,10%. Once again, only a few pages of the index should be accessed and thus the approximation made for the number of pages has little impact.

Last, the third query obtains all the tuples from the table ordered by the sequential identifier of the sale. There is a high correlation between the order of the tuples in the table and the order of the sequential identifier (clustering). Therefore, an index can be used in such setting to eliminate the need to process a sort operation to resolve the *order by*

clause of the query. For this query, we had a greater cost difference, equal to 26.94%. As the query scans all the pages in the index, the estimates made for the number of pages are significant. Yet, the query plans chosen were exactly the same, both with hypothetical and actual indexes. This consistently happened in other experiments with the same characteristics.

7 Conclusions

In this work we have presented an engine that enables automatic indexes creation and destruction for DBMSs. The engine presented is based in an integration between software agents and DBMS's components.

Our reasoning model uses heuristics to choose and automatically create or destroy indexes during normal DBMS operation. We have implemented the proposed heuristics in a layered software agent coupled with PostgreSQL. We have presented some of the implementation issues in this paper, such as the extension of PostgreSQL to include hypothetical indexes and the process synchronization necessary to integrate the software agent with the DBMS.

As main contributions of our work, we can cite (i) an extension of PostgreSQL to include the notion of hypothetical indexes, and (ii) an actual implementation of a self-tuning agent integrated with PostgreSQL.

As ongoing work, we can mention the conduction of a detailed performance evaluation of the index tuning agent [17]. In the future, we plan to refine the implementation of hypothetical indexes in PostgreSQL in order to improve estimation quality. Finally, a longer-term goal is to investigate the construction of new local self-tuning agents and their integration in a global self-tuning architecture.

References

- [1] J. V. D. Akker and A. Siebes. Enriching Active Databases with Agent Technology. In *Proceedings of the First International Workshop on Cooperative Information Agents (CIA-97)*.
- [2] S. Chaudhuri, M. Datar, and V. Narasayya. Index selection for databases: A hardness study and a principled heuristic solution. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1313–1323, 2004.
- [3] S. Chaudhuri and V. Narasayya. An efficient, cost-driven index selection tool for microsoft sqlserver. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 146–155, 1997.
- [4] S. Chaudhuri and V. Narasayya. Autoadmin “what-if” index analysis utility. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 367–377, 1998.
- [5] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1–10, 2000.
- [6] S. Clamage. Mixing c and c++ code in the same program. <http://developers.sun.com/tools/cc/articles/mixing.html>.
- [7] R. Costa, S. Lifschitz, and M. Salles. Index self-tuning with agent-based databases. *CLEI Electronic Journal*, 6(1):22 pages, 2003.
- [8] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic sql tuning in oracle 10g. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1098–1109, 2004.
- [9] Y. Diao, F. Eskesen, S. Froehlich, J. L. Hellerstein, L. Spainhower, and M. Surendra. Generic, on-line optimization of multiple configuration parameters with application to a database server. submitted to Distributed Systems Operations and Management, 2003.
- [10] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Transactions on Database Systems*, 13(1):91–128, 1988.
- [11] M. Frank, E. Omiecinski, and S. Navathe. Adaptive and automated index selection in rdbms. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 277–292, 1992.
- [12] E. Kendall, P. Krishna, P. Murali, C. Pathak, and C. Suresh. A framework for agent systems. In M. Fayad, D. Schmidt, and R. Johnson, editors, *Implementing Application Frameworks: Object-Oriented Frameworks*, pages 113–154. John Wiley & Sons, 1999.
- [13] S. Lifschitz and J. A. F. Macêdo. Agent-based databases and parallel join load balancing. In *Proceedings of the Latin Conference on Informatics (CLEI)*, 2001.
- [14] S. Lifschitz, A. Y. M. Milanes, and M. V. Salles. State of the art in self-tuning relational database systems (in portuguese). Technical report, Departamento de Informática, PUC-Rio.
- [15] G. Lohman, G. Valentin, D. Zilio, M. Zuliani, and A. Skelley. Db2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 101–110, 2000.
- [16] Postgresql dbms. <http://www.postgresql.org>.
- [17] M. Salles. Autonomic index creation in databases (in portuguese). Master's thesis, Departamento de Informática, Pontificia Universidade Católica do Rio de Janeiro (PUC-Rio), 2004.
- [18] D. Shasha and P. Bonnet. *Database Tuning: Principles, Experiments and Troubleshooting Techniques*. Morgan Kaufmann, 2003.
- [19] M. Wooldridge. Intelligent Agents. In G. Weiss, editor, *Multiagent Systems - A Modern Approach to Distributed Artificial Intelligence*, chapter 1, pages 27–78. The MIT Press, Cambridge, Massachusetts, 1999.
- [20] D. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. Db2 design advisor: Integrated automatic physical database design. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1087–1097, 2004.