Design and Implementation of a Global Self-tuning Architecture

Anolan Yamilé Milanés and Sérgio Lifschitz

[1]Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
PUC-Rio – RJ – Brasil

{anolan,sergio}@inf.puc-rio.br

Abstract. *Self-tuning is a feature that enables the automation of DBMS functions. The main goal is to keep a good performance while minimizing the DBA interaction with the system. This work proposes an architecture for DBMS automatic tuning. We consider here tuning as a global issue, given that changes of a single parameter can affect the performance of other operations. Besides describing the architecture details, we discuss its integration within PostgreSQL in a implementation based on software agents. This way we provide a more flexible approach to include automatic tuning features in a DBMS.*

Resumo. *A auto-sintonia é uma característica que automatiza as operações de um SGBD visando manter um bom desempenho enquanto a interação do DBA com o sistema é reduzida. Este trabalho propõe uma arquitetura para sintonia automática de um SGBD. A tarefa de sintonia é tratada aqui como um problema global, dado que alterações de um parâmetro podem ter reflexos no desempenho de outras operações. Além da descrição dos detalhes da arquitetura, discutimos também sua integração ao PostgreSQL em uma implementação baseada em agentes de software. Dessa forma temos uma abordagem mais flexível para a inclusão de atividades de sintonia automática em um SGBD.*

1. Introduction

Current Database Management Systems (DBMS) offer a set of tuning knobs and configurable parameters that enable modifications, known as database tuning activities, in the actual assignment of hardware and software system resources. It is an advanced task that requires creativity, knowledge and persistency from Database Administrators (DBA).

Most recent commercial DBMS versions include some sort of automatic features (e.g., [Dias et al. 2005]). These systems include autonomic, self-manageable or self-tuning features, all these terms referring to the same basic idea: as DBMS are getting more complex and its management is quite difficult even for database gurus, most of the tuning tasks should be automatically adjusted, with none or very low human intervention.

There are many research works aiming at solving some of the main database tuning activities in a (semi)automatic way. A common approach is to focus on isolated problems and specific solutions, the so called local self-tuning strategies [Lifschitz et al. 2004]. However, system components interactions and different workloads may degrade the overall performance, and a global tuning approach should rather be considered. This brings us to the idea of a global tuning system (e.g.

[Barrera III 1993, Weikum et al. 1994, Costa et al. 2003, Dias et al. 2005]), where particular tuning activities at each available resource may be coordinated with others.

Sometimes performance problems are caused due to local tuning actions that are executed without this global view of the system. Indeed, when all the indices pages are present in main memory, queries response time are usually considerably improved. If one wants also to reduce contention and the number of disk accesses, the DBA could increase the size of the buffer. Both actions would take most of the available memory area, affecting operations such as sorting and hashing. As a consequence, query execution plans would limit the optimizer choices, leading to bad performances.

The main contribution of this paper is the proposal of a global self-tuning architecture, in two versions (centralized or distributed), that enforces the requirements for automatic systems in the database context. Additionally, we have studied the implementation of our ideas embedding the architecture into PostgreSQL [Postgresql], an open-source relational DBMS widely used that has no self-tuning features in its official distribution.

Our prototype used software agents as the main components of the architecture. This way it may be possible to include new self-tuning mechanisms in a independent manner, with the inter-relationships managed by the system. This is one of the main differences of our approach, as previous works usually deal with self-tuning components integrated in a given DBMS context. The integration of software agents and database systems in order to increase their adaptability and flexibility, is also one of the main interests of our research group [Lifschitz and Macêdo 2004, Costa et al. 2003, Lifschitz and Salles 2005].

This paper is organized as follows: in the next section, we discuss some general approaches for self-tuning, focusing on works directly related to one presented here. Then we propose our architectures, stating also some of the requirements that self-tuning systems should observe in Section 3. The implementation issues and a case study with PostgreSQL are explained in Section 4. We conclude listing the main contributions, ongoing and future work in Section 5.

2. Related Work

It is very difficult to characterize the performance behavior of a DBMS as a whole [Weikum et al. 2002]. Therefore, most previous works have given more attention to tuning issues related to DBMS individual components. This allows researchers to gain insight into the factors that affect performance of specific DBMS modules. It serves as a basis to the investigation (and validation) of the interrelationships among components.

Relatively few works have been done to evaluate how different components that compete for resources affect each other's performance. Practical tuning experience indicates that understanding such correlations can be quite challenging [Shasha 1996].

Works present in the literature can be divided into two main research directions: *global* self-tuning and *local* self-tuning [Lifschitz et al. 2004]. Local self-tuning tries to solve specific database tuning problems, such as index selection and management, physical data placement, multiprogramming level, buffer page replacement policies and others. Global self-tuning, on the other hand, focus on the construction of systems that can automatically maintain an equilibrium among the resource use made

by different DBMS components in order to achieve the best overall performance and throughput. There are some proposals aiming to build global self-tuning database systems [Martin et al. 2004, Chaudhuri and Weikum 2000]. Although these promise great benefits, relatively few works implement and evaluate such complex systems.

Previous works have proposed the implementation of global self-tuning for computational systems in general, and particularly for database systems [Lifschitz et al. 2004]. In [Barrera III 1993], the author suggest a process where a single agent with global system knowledge is responsible for analyzing the data collected by sensors and stored in a common repository. The result of the analysis is the main input for actions that are executed.

In [Hellerstein 1997] there is a description of the ATS - Automated Tuning Systems - whose operation is essentially the same as the one proposed by [Barrera III 1993]. Besides observations about the technical challenges for building such systems, the authors intend to solve the ATS integration problem to guarantee the peer-to-peer performance level. The main idea is to use an ATS coordinator that balances the requirements from each ATS from the lower level, with its system global view.

The implementation of an agent-based system for a generic self-tuning is given in [Bigus et al. 2000]. This generality is obtained by a model of the controlled system that may be rebuilt as soon as the controlled system characteristics change. This model permits the mapping of the expected values of service levels into control parameters.

When we focus at the database application domain, there are some different strategies for constructing global self-tuning systems. First, systems specially designed are implemented in such a way that the system itself contains enough knowledge about performance models with respect to its interaction with the environment. However, as already mentioned in [Chaudhuri and Weikum 2000], these systems need to deal with the complexity of integrating self-tuning mechanisms with the system code base. A prototype that follows this direction has been explained in [Harizopoulos and Ailamaki 2003]. As far as we know, no practical systems exist using this approach.

Another possibility would be to study the system as if it was a black box. The main advantage here is that global self-tuning components could be coupled to existing systems with no need to interfere in their code. However, the self-tuning component needs some behavior modeling for the system being observed [Bigus et al. 2000]. Furthermore, we must take into account some mechanisms needed to monitor the system (observation) and to make tuning actions (control) externally accessible. Following similar ideas we can cite the Quartermaster system [Martin et al. 2002], which aims at global self-tuning in an electronic commerce system.

In [Costa et al. 2003], the basic idea is to integrate local self-tuning services using an agent-based architecture that aims at database indices self-tuning. The global self-tuning model proposed includes a repository where the agent's already executed operations are stored, in order to avoid their re-execution if in the past these operations have led to a worse performance. The proposed model already stated that agents communication may guarantee a combined action that is balanced with respect to the whole system.

All the existing proposals use the feedback control loop [Weikum et al. 1994] (or a similar idea) as the basic control mechanism. This enables the system output continu-

ous control, as the function that relates the configuration and workload parameters with performance is usually unknown.

More recently, commercial DBMS are giving more attention to the idea of global self-tuning, as in the case of Oracle 10g [Dias et al. 2005]. They have included a specialized component for automatic diagnosis that take into account the total database throughput, rather than only local performance problems. However, the idea of using the *database time* as the unit to compare different performance and administration problems may not capture many common situations that arise in practice.

The work in [Benoit 2003] have discarded possible solutions for this problem with typical optimization techniques. Due to the intrinsic complexity of this problem, even approximate mathematical models are very hard to be generated. This is the main reason for engineering-based proposals (e.g. [Weikum et al. 2002]) that aim at simplifications for the whole tuning process. The architectures proposed in this paper, to be described next, follow these ideas.

3. Architecture

We propose in this section an architecture for the implementation of a global self-tuning feature in a DBMS. We first recall the main steps that self-tuning approaches go through. Then, we list some important observations that must be considered by any global self-tuning architecture to be effective in practice. Finally, we briefly present our architecture. A complete and detailed presentation may be found in [Milanés 2004].

Self-tuning stages

The self-tuning steps do not vary considerably from one proposal to another. Indeed, in [Hellerstein 1997] the main phases are detection, diagnosis, action and evaluation. In [Barrera III 1993] the suggested steps include the performance expectation definition, performance monitoring, analysis and action. There are only three steps in [Weikum et al. 1994]: observation, prediction and reaction. We have previously worked with slightly different four-steps generic self-tuning process [Costa et al. 2003]:

1. *Information Retrieval (IR)*: when system observation occurs or, specifically, the observation of the system region where self-tuning is active;
2. *Situation Evaluation (SE)*: according to measures obtained in IR stage and with metrics related to the level of system performance, the performance is evaluated and the need of adaptations is defined;
3. *Possible Alterations Enumeration (PAE)*: when it is detected that a given component is not answering adequately, the possible alterations are enumerated;
4. *Alterations Accomplishment (AA)*: from the enumerated alternatives, the self-tuning engine can make alterations in multiple DBMS components;

It is important to note that the choice of the evaluation metric for the SE stage is a difficult step in the self-tuning process, as the performance evaluation criteria is very dependent on a particular situation. The domain of possible alternatives is, generally, large and complex. Failing to consider a relevant metric could harm the solution. On the other hand, considering every possible alternative implies in an overhead to the system.

However, it is in the PAE stage that the bigger overhead to the system is generated. Some alternatives are elaborated and their benefits/costs are calculated. This process is

similar to and as costly as the process of plan generation done by conventional query optimizers. Not always the best alternative will be among the feasible ones.

The difference among proposals and the number of steps is related to considering (or not) this PAE stage as part of the self-tuning engine. Depending on the procedure chosen for the execution at the SE stage, those two stages could be integrated.

Some Requirements

Besides considering the important stages for an actual self-tuning feature, we list here some important issues that should be considered for any architecture development. These are facts that may become requirements for system implementations:

- Today's commercial systems contain multiple tuning knobs. Therefore, we should avoid storing all metrics that describe the reaction of the system for all adjusting parameters, and for all possible workloads;
- Some works describe complete local self-tuning processes, from problem detection up to the tuning itself. However, these works make some considerations upon the DBMS state [Shasha and Bonnet 2003] that may not be always valid;
- Most proposals do not take into account other system components, including those that may cause similar skew on tuning metrics. This means that the integration of local self-tuning modules may lead to unexpected effects. Indeed, a good example is mentioned at [Weikum et al. 1994], where data placed unevenly on disks may imply slow access, and this implies higher locking periods. When data contention gets to a critical level, their Multi Programming Level (MPL) control mechanism would stop accepting new transactions and canceling some of the current ones. This would cause a CPU overhead due to many transaction's restarts.
- The complexity of diagnosing a database performance problem and its actual tuning justifies their separate execution in a system, in a concurrent execution. This has been discussed in [Costa et al. 2005], where the SQL queries capturing task is parallelized with the creation and dropping of indexes and the analysis process. While an internal agent is responsible for SQL submissions, other agents are executed outside the system, in order to keep the overhead at a low level;
- The action execution capabilities of a given local self-tuning component may be useful for other system components. Indeed, the mechanism that controls transaction admissions into the system, related to the data contention control, can be used to control the maximum number of concurrent system transactions.
- The task of determining the performance problem cause is easier if the tuning tasks are executed one at a time, separated by intervals that depend on how much time the system takes to react to a given adjustment, and the monitoring time interval.

The underlying idea for our proposal, to be described next, is to integrate local self-tuning engines within a global self-tuning architecture in such a way that this guarantees a flexible approach for individual automatic tuning mechanisms cooperate in a *social software community*. This enables the construction of a system that may diagnose performance problems as well as schedule actions to correct them, while keeping a stable situation. We might take advantage of the several existing works in local self-tuning to obtain a global balance of the system.

The architecture design is motivated on the previous studies in global self-tuning. The authors in [Weikum et al. 1994] argue for the need of a feedback control loop. In [Barrera III 1993] there is a discussion on the need for a component database in charge of gathering each component expectations of use and resulting performance. The paper states that analysis and expectations (or user goals) should be concentrated in a single agent and database. Similar architectures were proposed in [Martin et al. 2002, Wiseth 2003].

The main differences brought by our proposal include a distributed architecture, the implementation using software agents and the reuse of local self-tuning components.

Architecture Proposal

Our architecture consists basically of a set of components that control many different DBMS modules. There is also a knowledge base that stores all the information concerning the DBMS state. Moreover, the architecture keeps track of the components actions, user goals and DBMS component interactions. All accesses to the knowledge base are done through an Historical component.

Other local self-tuning components might be incorporated to the systems infrastructure. The combination of each component tasks enables a coordinated work towards global self-tuning. For example, the queries collector [Costa et al. 2005] for an automatic index mechanism would be helpful for the whole system to obtain information about the submitted workload.

The relationship among components in our architecture depends on the system control distribution degree, which can range from a *centralized* approach to a completely *distributed* strategy. These differ basically in the way decisions upon automatically tuning are taken, either by one or simultaneously by multiple components.

The centralized approach follows the basic manual process that is executed by a DBA regarding actual database systems. That is, the administrator receives some information about performance problems and starts its analysis. His evaluation is based on collected metrics, system documentation, data provided by some external tool and also, his own experience. Then, the DBA determines what would be the main reasons for the problem and decides to execute some adjustments, while observing the reaction of the system. He needs to be ready to undo some modification if there is any negative impact for the system.

Our architecture, depicted in Figure 1, permits the execution of this complete method replacing the DBA with a software agent. In this centralized case, there is a Coordinator component that decide upon tuning actions.

The whole process here consists of a central component that decides upon tuning actions when it receives the information from other components that some metric is far from the expected values. In this approach, the components directly associated to the available resources participate mostly as information collectors but also act when local self-tuning services are invoked by the coordinator.

The central component is the only one that has a global vision and, therefore, it may determine which is the real cause of a given performance problem. This coordinator component is the only one with enough power to decide upon executing new tasks, exactly

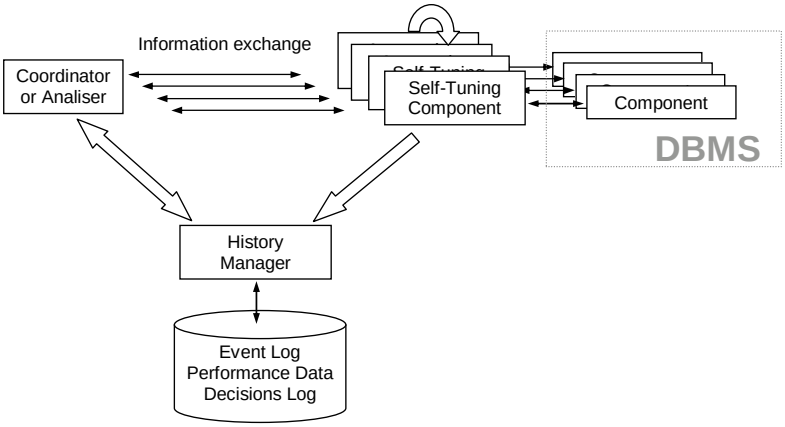


Figure 1. Global self-tuning Architecture

the way it is proposed in [Barrera III 1993]. In order to analyze the whole situation, it needs to understand every existing variable, as well as the specific features that the available local (self-)tuning mechanisms offer to solve the problems.

In our distributed version, on the other hand, the control and knowledge of the system architecture are distributed through the system components. In Figure 1 we would have now an Analyzer component (instead of a Coordinator) that participates of the global self-tuning architecture sharing the decision process with all other components. Differently from the centralized approach, here the solution process will be based on the negotiation between the system components and not on the decision of a unique component.

The distributed answer to a given performance problem is described as follows:

1. During the monitoring step, each sensor collects performance metrics;
2. Data collected are either sent to the Historical component, that offers a registry service specific for this, or they are kept by the agent, which would have to offer some path for access by other agents. As there are a great number of metrics, these cannot freely circulate inside the system.
3. These data collected are compared to the expected values. If a performance problem is detected, a tuning process must be started.
4. The planning step aims at determining the problem's cause, and the elaboration of an action (or intention) plan to solve it. Decisions taken by the analyzer have greater priority, as it has exclusive access to information in the knowledge base related to restrictions and conflicts. The local self-tuning components related to this problem can elaborate proposals that may be considered as soon as they do not violate these restrictions.
5. The actions enumerated at the result plan are scheduled to be executed by the corresponding components. Actions execution requests appear as service requests. Each task execution is reserved to the local self-tuning component that has suggested the solution selected to be executed (but the component could also assign it to any other component that offers the service).

Let us consider an example to illustrate our general methodology. During the monitoring phase, a sensor collects average throughput data of the system, which are sent to the repository. When the average value expected for this metrics is less than expected,

an alarm message is sent to the system. The analyzer sends, then, a message announcing a ”low throughput”. The components that may be related to this problem detect that a planning session is about to start.

Each component invoked by the message may offer its own solution to the problem. A component implemented as in [Heiss and Wagner 1991] would reduce the MPL as soon as the throughput is also reduced, and will restart increasing it when the throughput stops to go down. On the other hand, the module responsible for buffer pages replacement control would modify the current policy when long sequential scans are detected. The index management component would suggest the dropping of the index that might be causing the higher level of locking. And the analyzer, following our model, would suggest to lower the MPL or a temporary interruption on accepting new transactions.

In this case, both the MPL and the analyzer agents are proposing an action involving the same parameter, generating a conflict. Considering that similar situations (that have led to performance problems) were not present at the knowledge base, all but the MPL control action out of the suggested proposals would be executed. The solution of this particular conflict should take into account that the analyzer agent has access to the system's model. Thus, he should always win in a conflict situation. The MPL agent is not liberated because the decision MPL agent behavior could lead to oscillations. However, the action layer will probably execute this action, with the analyzer agent controlling it.

The main drawbacks of the centralized architecture include its limited expansibility, restricted autonomy for its components and strong dependency on the Coordinator. However, it is easier to implement, when compared to the distributed version. The advantages of the distributed-decision architecture are due to the fact that there are situations in which the self-tuning mechanisms, isolated, do not offer better solutions. However, the diagnosis and planning processes imply that the analyzer decisions should always have the highest priority. There is no clear winner among these two proposals, but the main underlying idea is the same: integrate local self-tuning features in such a way that collaboration between them may bring more stable tuning solutions for practical situations. In the next section we propose and explain how we have implemented some of these ideas.

4. Implementation

We briefly describe here (further details may be found in [Milanés 2004]) the way we have implemented a prototype that instantiates the centralized architecture proposed here (although planned, the distributed architecture has not been yet implemented due to its intrinsic complexity). The prototype consists of a self-tuning system embedded in an actual relational DBMS. The main idea is to show the feasibility of our architecture, discuss the advantages and main drawbacks of our solution and to evaluate the multi-agent system influence on the DBMS performance.

Agent-based databases

As expected, the implementation of our architecture is quite complex. We need a software abstraction that offers adaptability and reactivity, among others. Despite their many distinct definitions, the software agents paradigm was chosen due to its main features, which may be useful for any system automation [Bigus et al. 2000, Wooldridge 1999].

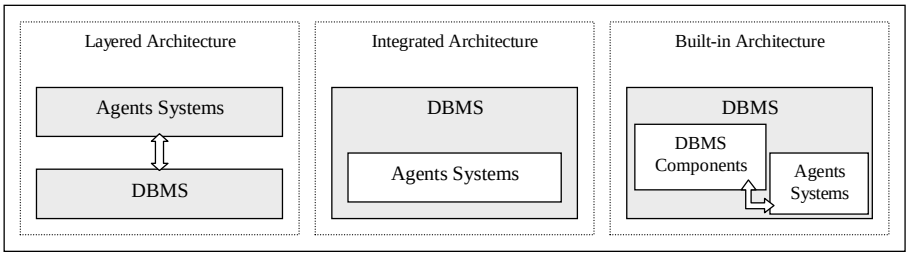


Figure 2. Architectures for the integration of Agent Systems and DBMS

In [Lifschitz and Macêdo 2004] three integration architectures between agents and DBMSs are proposed: Layered, Integrated and Built-in (Figure 2). Each one of the three integration architectures has advantages 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.

Due to its complexity, we could not consider the implementation the Integrated architecture. Also, the Layered architecture would not allow us to manipulate PostgreSQL internal parameters that are not exposed by the API. Therefore, we consider in this work the Built-in architecture to implement the self-tuning feature. This was a straightforward solution in the direction of the research done by our group (e.g. [Costa et al. 2003]) and also an acceptable and feasible solution regarding efficiency.

Agents implementation

In [Costa et al. 2003] the authors considered the agent construction architecture proposed by [Kendall et al. 1999] (see Figure 3), based on a framework for designing and implementing software agents. This architecture is organized in vertical layers and enables, in a relatively simple manner, to build pro-active and reactive agents. It is possible to map the self-tuning process steps, suggested in [Costa et al. 2003] to this model's layers. And these, on the other hand, can be mapped directly to our main proposal of this paper, the four global self-tuning process steps: observation, analysis, planning and action.

The architecture shown in Figure 3 has 2 main flows, from the top layer to the bottom layer and vice-versa. The bottom layer (Sensory) monitors the environment and updates the beliefs. This activates a procedure at the Reasoning level. This procedure may generate either plans to be executed at the Action layer or messages to be prepared and sent by the Collaboration layer. The other layers must deal with the message translation and agents mobility.

Once at the receptor agent, the process goes in the reverse direction. The top layers get the messages and eventually pass them until they reach the Reasoning layer. The beliefs may be updated or not, or update the Sensory layer, generating new plans. In this configuration, perception and action are each involved with at most one layer.

This system must enable the insertion of agents as their main components, built

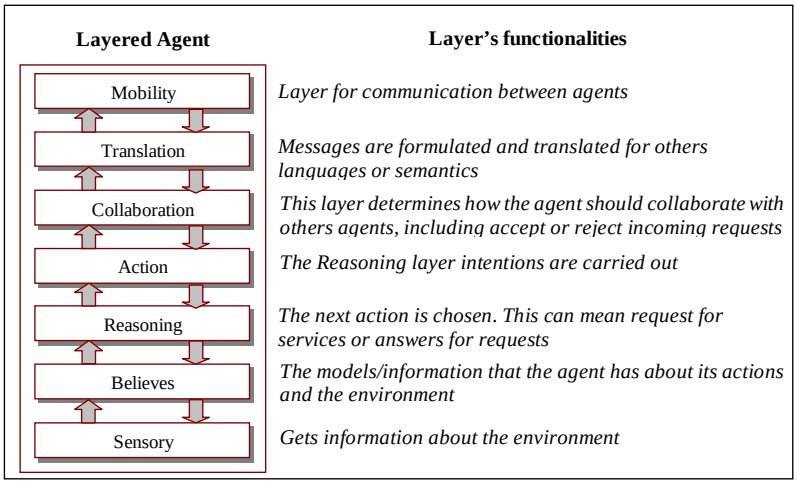


Figure 3. Layered Architecture for Software Agents

with respect to a given *template*. This way it is possible to minimize the programming effort for each new component and enforce some standardization. The agents are meant to work in collaboration with others but should be able to run also in isolation. This feature would enable a step-by-step inclusion of self-tuning mechanisms within the DBMS.

This template, following the ideas proposed by [Kendall et al. 1999], was also used in previous works of our group (e.g. [Costa et al. 2003]). For the current architecture proposed here, we had to incorporate the Collaboration layer and the concurrency management. We consider here that our agents use a common ontology and mobility issues are not important. Therefore, our template does not include the Translation and Mobility layers.

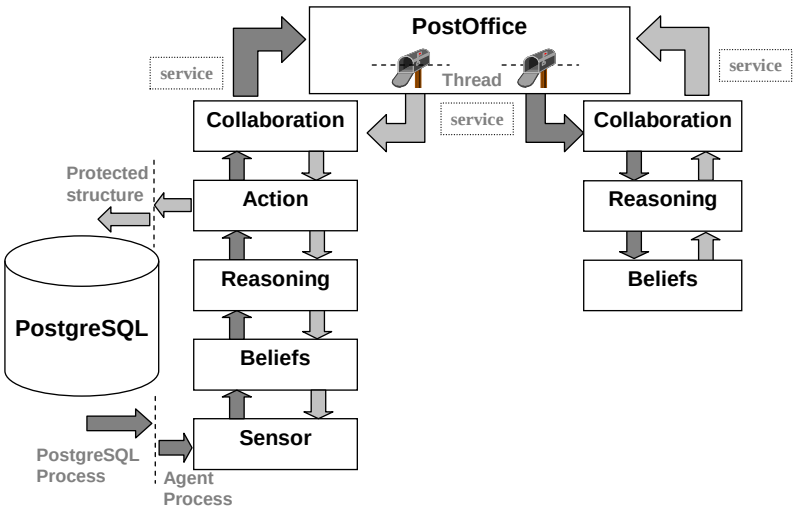


Figure 4. Self-tuning system interactions

We show in Figure 4 a communication process in our system, involving a single DBMS process and two agents. It is worth to note that agents requests are exchanged as services. One of the agents has only the Beliefs, Reasoning and Collaboration layers. This could be the Coordinator agent in our centralized architecture. There is no need for a

Sensor layer because the environment this agent captures comes through all Collaboration layers. Also, the Action layer is not present as all the actions to be executed are service requests to other system agents, that are sent by Collaboration layers. The information traffic between the Agent System process and the PostgreSQL processes may involve protected structures.

In this particular case, the Coordinator agent must decide upon which actions have to be executed, always as service requests. This decision is based upon information received from other agents. An example of service would be a message from the *Buffer Page Replacement* agent to the *Historical* agent asking register the new state "MRU = true".

When a given agent comes to the Agent System, its registration is stored in a list kept by the *AgentSystem* object, which is unique to the system. At the same time, the agent's Collaboration layer is initiated and the agent is registered at the *PostOffice*. Each action execution capability for the self-tuning system is specified during the agent configuration and also registered at the *PostOffice*. This way they become available to other agents in the system.

Case Study

In order to check whether or not our proposed architecture is feasible in practice we have embedded it inside a real DBMS. A first problem was the choice of this DBMS, as we need to develop code involving metric variables as well as control functions and parameters. Most commercial DBMS publish some API that enable access to system's functions. However, these do not enable internal code modification, as expected. Another related problem is that commercial systems already have some self-tuning features and these could affect our tests. Last but not least, this research has been done in close collaboration with [Lifschitz and Salles 2005], where a local index self-tuning strategy could be used in our work as one of the specific components.

Therefore, we have chosen to work with an open source DBMS. We have analyzed some different systems, in details only two of the most popular and robust ones, which are MySQL [MySQL] and PostgreSQL [Postgresql]. Both have versions for different operating systems and are widely used in many different current applications. Neither have self-tuning features implemented, at least not yet published ones. We have decided to work with PostgreSQL for basically two reasons. First, we needed well organized code and also large, detailed and good documentation. PostgreSQL, at least at the time this implementation work was done, was definitely much better than MySQL in these aspects. Second, as already mentioned, it was being used already by other people in our group. Thus, we could in the future add new local tuning features in a much easier way.

The coding language for all implementations was C++, and the main reason for this was the fact that PostgreSQL is actually written in C. The compiler was gcc version 3.2.2. The hardware platform we have considered was based on an Athlon 1.6GHz with 128MB RAM under RedHat Linux version 8 operating system.

DBMS interaction

The PostgreSQL code was modified in order to include a new parameter for command line execution that starts the self-tuning feature (file postmaster/postmaster.c). When this op-

tion is enabled, the internal agents are instantiated and included in the agent system. The interaction between PostgreSQL and the self-tuning system is done through an interface (postgresInterface.cpp) with functions that make calls to creation (and destruction) methods from the agents system. The interface contains also functions that execute actions on the DBMS and those invoked by the DBMS to notify about an event. The self-tuning system use simple text files as information repositories for saving the information that must be analyzed in order to detect violation of the performance expectations and also check the validity of a given action when it was already executed in a similar context.

We have tried to implement our global self-tuning feature in PostgreSQL in such a way that it would not interfere in regular DBMS activities. That is, when the self-tuning feature is disabled, the DBMS runs as there was no modifications. Even though, our agents are embedded [Lifschitz and Macêdo 2004] in the DBMS, and there is resource consumption even when they execute in separate processes.

The internal events detection is done through *checkpoints* or *software probes*. These are instructions that are inserted in a program code aiming at collecting and storing data, or even a measuring routine. To avoid possible interference problems due to measurements, our implemented policy states that the PostgreSQL process should only give the information needed and then return as the corresponding sensor executes in another concurrent process.

In order to check the feasibility of our proposed architecture in a practical situation, we have initially implemented two local self-tuning mechanisms: one for automatic change of buffer management page replacement policy and another that deals with thrashing and data contention, related to the Multi Programming Level.

Page Replacement Policy

We have added to the PostgreSQL code a function that implements the MRU buffer page replacement policy. This option is not yet included in the official distribution of PostgreSQL. The basic difference to the existing LRU strategy is that MRU replaces the most recently used pages before. Older information is kept for faster access.

This is certainly a good strategy for information that is used for database requests only once. Indeed, in our implementation, when a given query execution plan includes a sequential scan, the sensor would be notified by the corresponding PostgreSQL process. The agent, then, may evaluate if the action of changing the buffer page management policy is worth. In this case, it would execute the corresponding function that we have added to the PostgreSQL code.

Data Contention Thrashing

The variable *conflict ratio* indicates the number of locks hold by active transactions with respect to the total number of assigned locks. These values are registered with the introduction of *checkpoints* at the PostgreSQL file that has all the locking confirmations (lock.c), which actually correspond to the interface ags_notify_lock() function calls that notifies these values. Each transaction's end is also notified. This event activates the (transaction) admission control mechanism, which may exclude transactions off the queue when the conflict ratio is too high.

The sequence of events is basically the following: as soon as the MPL control agent receives a PostgreSQL data, it determines the new value for the conflict ratio. This value is, then, sent to the historical agent to be stored in the centralized repository. It is also evaluated to detect if there is any risk of data contention thrashing. When there is a violation, the coordinator agent is notified. It starts immediately an analysis of the system state aiming at diagnosing the actual cause of the problems. The whole idea is to avoid the situation where the tuning action not only does not solve the problem but, rather, makes the system performance worse. Later, when the possible solutions to the performance problem are enumerated, they are evaluated with respect to the available information on already executed actions. Finally, the chosen actions are forwarded to the corresponding agents as service requests, so the performance problem would automatically be solved.

Implementation issues

This implementation work consisted basically on the C++ agent framework implementation and the necessary customizations to embed instantiated agents from this framework into an actual DBMS. We have considered initially previous versions we had available [Lifschitz and Macêdo 2004, Costa et al. 2005]. These are single agent implementations and we have added concurrency management and the collaboration layer.

It is worth to remember that there is no concurrency support in C++. The use of *Message Queues* has avoided the need of implementing semaphores to guarantee exclusive access, and also it has made it easier to code waiting conditions. As it was done in [Kendall et al. 1999], we have made extensive use of design patterns.

The collaboration layer was a hard task since the centralized collaboration is not well documented in [Kendall et al. 1999] and the distributed version is not complete. Thus, we have implemented the centralized collaboration version using the *Mediator*, as suggested in [Kendall et al. 1999], that is the basis of *PostOffice*. Besides supporting asynchronous messages, it includes also synchronous traffic due to the specific needs of the coordinator agent.

With the basic structure of the software agents already built, all we need to do is to extend the *Sensor*, *Belief*, *Reasoning* and *Action* classes (or only those that the agent really needs) in order to instantiate each agent that will participate in the system.

The insertion of agents into the PostgreSQL code has been a great challenge. Indeed, PostgreSQL is a robust and complex system, that has to be exploited and studied in details if one wants to embed code as our agents system into it. In our prototype the agents are executed in different *threads* inside the postmaster process.

A better solution would assign the agents system to a separate process, isolating it from the DBMS. In case of errors with the agents system only, the DBMS processes would not suffer and even debugging would be easier. However, this is far from being a trivial task, as it is necessary to reproduce the non-documented PostgreSQL procedure that is executed each time a new process is created. This solution was eventually implemented in [Costa et al. 2005].

5. Conclusions

We have presented in this paper a study for the design and implementation of global self-tuning architectures for database systems. The main contributions of our work are: (i)

the enumeration of facts and requirements for this global self-tuning approach (ii) the description of an architecture that integrates local self-tuning features, and (iii) the actual implementation with a software agents paradigm within a real DBMS, which enables great flexibility for incremental addition of new modules.

The practical results obtained has shown the feasibility of our agent-based self-tuning architecture. Despite all the complexity of such systems, we claim that the architecture proposed in this paper is a viable alternative to more complex approaches that have been proposed in the literature. One main difference of our work when compared to previous proposals is that the local self-tuning components (here implemented as software agents) may work either cooperatively or independently.

We are currently implementing other local self-tuning components and we are studying ways to evaluate a distributed approach. We are also interested in exploring other negotiation strategies for components (agents or not). The idea here is to obtain a better autonomy degree. We intend to investigate next how insertions and queries to the knowledge database may be optimized. Moreover, we want to study performance metrics choices for the SE stage and technics for minimizing the IR stage cost.

References

Barrera III, J. S. (1993). Self-tuning systems software. In *Procs Workshop on Workstation Operating Systems, IEEE Computer Society*, pages 194–197.

Benoit, D. (2003). *Automatic Diagnosis of Performance Problems in Database Management Systems*. PhD thesis, School of Computing, Queen's University.

Bigus, J. P., Hellerstein, J. L., Jayram, T. S., and Squillante, M. S. (2000). Auto tune: A generic agent for automated performance tuning. In *Procs International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Systems (PAAM)*, pages 33–52.

Chaudhuri, S. and Weikum, G. (2000). Rethinking database system architecture: Towards a self-tuning risc-style database system. In *Procs Intl. Conf. Very Large Databases (VLDB)*, pages 1–10.

Costa, R., Lifschitz, S., de Noronha, M., and Salles, M. (2005). Implementation of an agent architecture for automated index tuning. In *Procs IEEE Intl. Workshop on Self-Managing Database Systems (SMDB)*.

Costa, R., Lifschitz, S., and Salles, M. (2003). Index self-tuning with agent-based databases. *CLEI Electronic Journal*, 6(1):22pp.

Dias, K., Ramacher, M., Shaft, U., Venkataramani, V., and Wood, G. (2005). Automatic Performance Diagnosis and Tuning in Oracle. In *Online Procs Biennial Conf. on Innovative Data Systems Research (CIDR)*, pages 84–94.

Harizopoulos, S. and Ailamaki, A. (2003). A case for staged database systems. In *Online Procs Biennial Conf. on Innovative Data Systems Research (CIDR)*.

Heiss, H. and Wagner, R. (1991). Adaptive load control in transaction processing systems. In *Procs Intl. Conf. Very Large Databases (VLDB)*, pages 47–54.

Hellerstein, J. L. (1997). Automated tuning systems: Beyond decision support. In *Procs Computer Measurement Group*, pages 277–292.

Kendall, E., Murali Krishna, P., C.V., P., and Suresh, C. (1999). *A framework for agent systems*, chapter 6, pages 113–154. John Wiley & Sons. Implementing Application Frameworks: Object-Oriented Frameworks at work.

Lifschitz, S. and Macêdo, J. (2004). *Agent-based Databases and Parallel Join Load Balancing*, chapter 6, pages 69–88. Sistemas de Información e Ingeniería de Software: Temas Selectos - Editora Centro Estudios Informatica, Univ. Simon Bolivar (Merida) Venezuela.

Lifschitz, S., Milanés, A., and Salles, M. (2004). State of the art in self-tuning relational database systems (in portuguese). Technical report, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio). Tech Report PUC-RioInf.MCC35, english version in preparation.

Lifschitz, S. and Salles, M. (2005). Autonomic index management. In *Procs IEEE Intl. Conf. Autonomic Computing (ICAC)*, pages 304–305.

Martin, P., Powley, W., and Benoit, D. (2004). Using reflection to introduce self-tuning technology into dbmss. In *Procs Intl. Database Engineering and Applications Symp. (IDEAS)*, pages 429–438.

Martin, P., Powley, W., Li, H., and Romanufa, K. (2002). Managing database server performance to meet QoS requirements in electronic commerce systems. *International Journal on Digital Libraries*, 3(4):316–324.

Milanés, A. (2004). An architecture for global self-tuning databases using agents (in portuguese). Master's thesis, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio).

MySQL. MySQL. http://www.mysql.com, accessed in July 2005.

Postgresql. PostgreSQL. http://www.postgresql.com, accessed in July 2005.

Shasha, D. (1996). Tuning databases for high performance. *ACM Computing Surveys*, 28(1):113–115.

Shasha, D. and Bonnet, P. (2003). *Database Tuning: Principles, Experiments and Troubleshooting Techniques*. Morgan Kaufmann.

Weikum, G., Hasse, C., Mönkeberg, A., and Zabback, P. (1994). The comfort automatic tuning project. *Information Systems*, 19(5):381–423.

Weikum, G., Mönkeberg, A., Hasse, C., and Zabback, P. (2002). Self-tuning database technology and information services: from wishful thinking to viable engineering. In *Procs Intl. Conf. Very Large Databases (VLDB)*, pages 20–31.

Wiseth, K. (2003). Oracle database 10g, the world's first self-managing, grid-ready database arrives. *Oracle Magazine*, pages 28–37.

Wooldridge, M. (1999). Intelligent agents. In Weiss, G., editor, *Multiagent Systems: A modern Approach to Distributed Artificial Intelligence*, chapter 1, pages 27–78. The MIT Press, Cambridge, MA, USA.