Index Self-tuning with Agent-based Databases

Rogério Luís de Carvalho Costa
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
Rua Marquês de São Vicente 225
Rio de Janeiro, RJ 22453-900 - Brasil
e-mail: rogcosta@inf.puc-rio.br

and

Sérgio Lifschitz
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
Rua Marquês de São Vicente 225
Rio de Janeiro, RJ 22453-900 - Brasil
e-mail: sergio@inf.puc-rio.br

Abstract

The use of software agents as Database Management System components lead to database systems that may be configured and extended to support new requirements. We focus here with the self-tuning feature, which demands a somewhat intelligent behavior that agents could add to traditional DBMS modules. We propose in this paper an agent-based database approach to deal with automatic index creation. Implementation issues are also discussed, for a built-in integration architecture.

1 Introduction

The software agents' research area aims at building systems that deal with heterogeneous, distributed and dynamic environments [26]. The agent technology has been used in many complex environments [3, 25]. 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. Agents are used mainly in applications where reactivity is very important [5].

For database systems, a possible approach is to make use of the agents' technology to add a reactive capacity to the DBMSs, that enables autonomous reconfiguration and extensibility. In general, DBMSs are known as passive systems that become active only in response to requests from end users or application programs. Extensions due to new systems requirements has created multiple extended DBMS classes, such as active database systems [23]. Active databases and software agent systems are quite similar as both are used on reactive applications [3]. Active rules are usually limited to deal with database objects and transactions, while agents may apply to the whole system environment.

There are many application domains for which Database Management Systems (DBMS) must be extended and configured [22]. In this paper we propose the use of software agents to deal with self-tuning and DBMSs operational requirements. This issue is considered one of the most important current research topics in databases [2].

The database tuning task consists of fine manipulations aiming at obtaining better performances of applications by means of an efficient use of the available computational resources. It is one of the main maintenance tasks of a database administrator. Commercial DBMSs offer a number of operational parameters that can be adjusted. Tuning can also be done in hardware configuration, physical design and query specifications.

A good prompt for the tuning process is to "think globally, fix locally" [21]. This means that we must understand the functioning of the entire system, but only carry through adjusts in specific points each time. Some factors have made the tuning process more complex, as in the case of the parallel machines and systems. These bring new questions such as the data allocation in multiple disks. Moreover, at each new edition of commercial DBMSs, additional operational parameters appear to be adjusted. So, tuning becomes even more important and, at the same time, more expensive, as highly specialized professionals are needed [2, 10].

One of the basic tuning activities is system monitoring. Statistics related to the effective use of the processors, the number of I/O operations and the execution time of basic operations must be collected, among others. The tuning process contemplates both the perception that a given system resource is not efficiently used and the comprehension why this happens, what is not a trivial task.

Database systems would, ideally, be configured in such a way that no tuning activities would ever be needed. All the resources would be placed and adjusted in the best possible way. When modifications are imperative, the DBMS would then be able to execute them automatically. This capacity of perception and automatic adjustment is known as self-tuning.

We are mainly concerned here with index self-tuning for database systems. Due to the high complexity of DBMSs' current implementations, self-tuning solutions are still very restricted. In order to achieve an actual self-tuning behavior, we need to rethink DBMSs' architecture [10]. We propose here an approach that matches agents systems and DBMSs in a feasible architecture. The agent architecture chosen is based on a object-oriented framework for building agent systems proposed in [15].

It should be noted that some of the existing approaches for index self-tuning, mostly in commercial DBMSs, are based only on index suggestion for specific workloads, leaving to the database administrator the decision on choosing the right representative workload and of the index creating. In our work we propose an index self-tuning complete process with automatic creation of indexes in a agent-based database architecture.

In the next Section we give more details on software agent systems and the way they may interact with database systems. We discuss some existing works related with semi-automatic index tuning and propose a self-tuning engine on an agent-based database architecture. An application focused on automatic index self-tuning is presented in Section 3. Then, in Section 4, we discuss implementation and architectural issues. Finally, we conclude in Section 5 listing our contributions and comment on future and ongoing work.

2 Agent-based Databases and Self-tuning

There exist multiple definitions for the agent term [5, 14, 18, 26] and only the autonomous characteristics have come to a consensus. Indeed, this is a central point related to the agency concept. We consider here basically the definition given in [26]: 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 that will be implemented here is based on a object-oriented framework for building agent systems proposed in [15]. This framework defines a layered architecture (Figure1) which identifies each agent function and can be used for both simple and complex agents.

Each layer communicates only with the layers that are located above and below it, with exception of the Mobility and Sensory layers, that communicate with others agents/environment's regions to execute their functions. This is the main reason why this layered architecture was chosen: each layer (except the extreme ones) depends only on its neighbors, making easier the implementation of new functionalities.

There exists, in Figure 1, two basic streams of information: one from the Mobility layer to the Sensory layer, and the other bottom-up in the opposite direction. The first one can occur when a message is received from another agent while the latter can occur when the agent uses the information captured by the Sensory layer to bring up to date its Believes, which are used to make decisions and execute some actions.

Agents and Database Systems

In [3] active databases are compared with agent systems. That work states that the integration of both technologies would even increase the complexity of the systems. It would be imperative to develop debugging

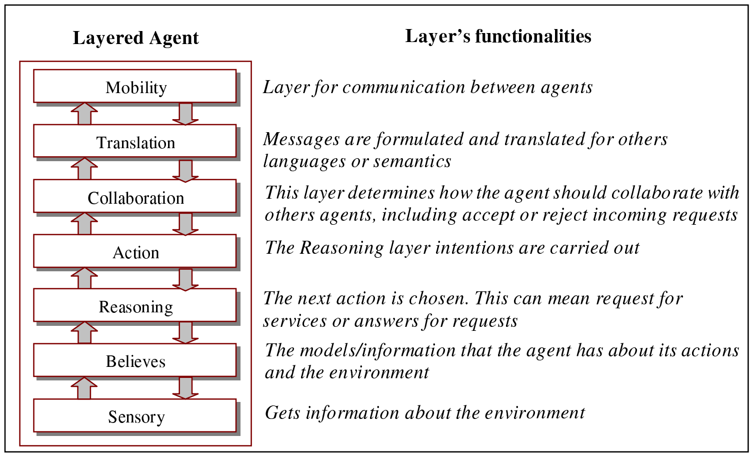
Layered Agent
Layer's functionalities
Mobility
Translation
Collaboration
Action
Reasoning
Believes
Sensory
Layer for communication between agents
Messages are formulated and translated for others languages or semantics
This layer determines how the agent should collaborate with others agents, including accept or reject incoming requests
The Reasoning layer intentions are carried out
The next action is chosen. This can mean request for services or answers for requests
The models/information that the agent has about its actions and the environment
Gets information about the environment

Figure 1: Layered Architecture for Software Agents

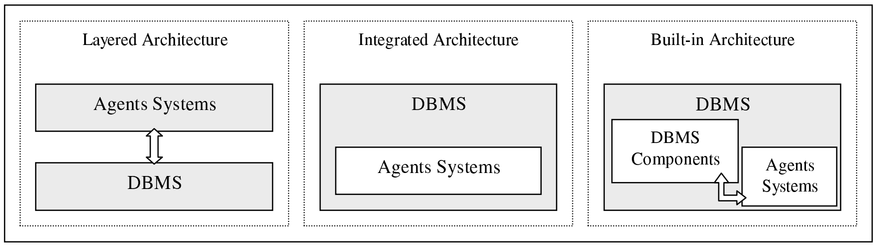

Figure 2: Architectures for the integration of Agent Systems and DBMS

tools as the control becomes more difficult. The focus was in a high level abstract comparison of both paradigms, with no direct consequences. However, a relevant point mentioned is that an important barrier to the integration of both technologies is the lack of methodologies for building plans and rules.

In [17] 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 (we omit here a detailed discussion due to space limitations). 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 will consider in this paper the built-in architecture to implement the self-tuning feature in an agent-based database architecture. This was also the case for a previous work [17] in which an agent-based database was built to further tailor the workload balancing process during parallel joins. We will discuss implementation aspects of this previous work and the one proposed here later in Section 4.

In this section, we describe some self-tuning previous studies related to the our work here. We present an architecture where most DBMSs self-tuning approaches fit and show how it can be implemented by the use of software agents. We enhance, still, the importance of the communication between diverse self-tuning agents for the establishment of a true integrated system that makes possible a full and effective DBMSs self-tuning process.

2.1 Self-Tuning in Databases

The self-tuning feature have been studied in many different areas. Transactions tuning and memory management are studied in [24]. In [16] a self-tuning engine is presented for the data allocation problem in parallel systems. In [10] the authors claim that deep architectural changes must be done in databases to enable actual self-tuning processes. A framework for tuning a physical database design is given in [20].

The work presented in [11] generates index sets from the monitoring of DBMS activities in a determined time interval. The indexes selection is achieved by considering indexes "benefit graphs". These structures represent the indexes sets that are presented and chosen by the optimizer, together with their execution costs. Each of the indexes sets selected by the optimizer is viewed as a graph node. The construction of a benefit graph for all the alternatives of indexes sets possibly tested and chosen, for each query, is unfeasible. Thus, a graph is generated only for those sets that will be chosen according to following criterion: given an initial set P of considered indexes and a subset C, of P, containing indexes chosen by the optimizer, $|C|$ subgraphs are generated, starting with $P - C_i$, where $1 < i < |C|$. Two heuristics are also considered: (a) if a set of n indexes induces a benefit S, each index will generate a benefit of S/n; and (b) the maximum value of a index benefit in a benefits graph is considered the index benefit for the query. The query optimizer must, then, be capable to generate an execution plan taking into account indexes that are not actually present in the database catalog[1].

In [1, 7, 8, 9] tools for indexes suggestion implemented in Microsoft's SQL Server are discussed, as part of the AutoAdmin project. Particularly, in [1], the indexes suggestion tool is presented with a materialized views suggestion tool. The objective of the index suggestion tools is to generate an index set considered for one determined input workload. A workload is supplied as input. This workload could be build from a log archive. Given a workload W with n operations, the tool generates n workloads, each one with a single operation, partitioning the initially submitted load. Then, it chooses the best configuration for each one of these loads, independently. Finally, it considers all the indexes belonging to one of the configurations chosen for the diverse workloads as candidates for W as a whole. From a workload with only one operation we have the selection of candidate indexes - in this stage only simple indexes are considered. Then, a greedy algorithm considers diverse configurations for each query. These configurations are submitted to the query optimizer (the existence of an index only worth if it is effectively used in a query plan execution). However, not all the possible indexes exist in the DBMS catalogue. Therefore, another module exists that makes it possible to the query optimizer to choose hypothetical indexes during elaboration of a query execution plan.

The selection of the input workload (or system time period observation) is the main difficulty for tools as the ones discussed. This happens because the database administrator does not always have all the information on what will actually be affected by specific operations when defining a workload [6]. Index suggestion engines do not take care of automatic adjustment of the DBMS, being highly dependent on decisions and actions made by the system or database administrator. They do not represent, then, a real self-tuning engine.

2.2 Self-tuning stages

In [24] are identified three main stages in the self-tuning process. Here we define four stages that are part of a generic self-tuning process:

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

It is important to stand out that the choice of the evaluation metric, in the SE stage, is difficult step in the self-tuning process, as performance evaluation criteria is very dependent on the particular situation . The domain of possible alternatives is, generally, numerous and very complex. However, it is in PAE stage that the bigger overhead to the system is generated. Some alternatives are elaborated and their benefits/costs are

[1]We will call Hypothetical Indexes to indexes that do exist in database catalog but are treated by the query optimizer as if they exist. The ones that belong to the database catalog will be called Real Indexes.

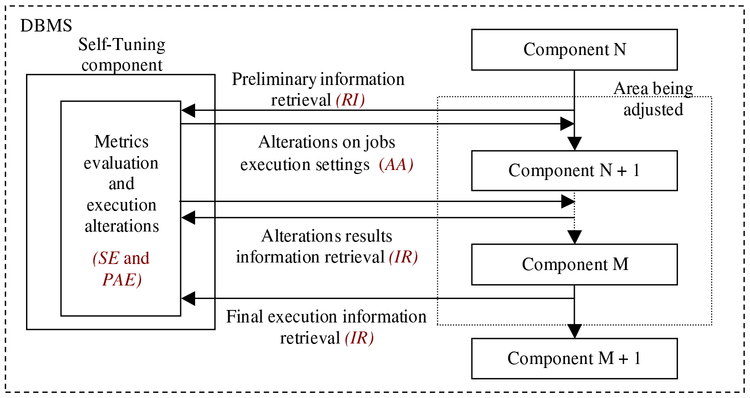
DBMS
Self-Tuning component
Metrics evaluation and execution alterations
(SE and PAE)
Preliminary information retrieval (RI)
Alterations on jobs execution settings (AA)
Alterations results information retrieval (IR)
Final execution information retrieval (IR)
Component N
Area being adjusted
Component N + 1
Component M
Component M + 1

Figure 3: Component's Local Self-Tuning Architecture

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.

It is important to note that, in some cases, this PAE stage is not considered in a self-tuning engine. This happens because in the IR stage it can be decided that it is not necessary to modify the system or that it will not be possible to reach better solutions. Then, PAE and AA will not be executed. On the other hand, some engines need to enumerate the alternatives before deciding whether or not to modify the system. Therefore, the PAE stage as well as IR and SE stages will always occur in these cases, while the last stage may not occur.

2.3 Local self-tuning model

In Figure 3, we propose an architecture to accomplish the self-tuning tasks discussed previously. The architecture's goal is to improve component's operations performance. This is made by a local component which is specialized in capture a determined task.

This structure observes a set of DBMS's components with intention to capture a low performance. Then, information is sent to diverse components listing alterations in determined task's execution and collects information of the results obtained. The self-tuning component can, also, use other DBMS's components to simulate the intended modifications. In this case, it collects results in an intermediate stage, before all the involved components in the operation have concluded its execution. This enables it to restart the process with new parameters. When the intermediate results fetched are the expected ones, it allows the complete execution of all the operations.

It can be noted that the IR stage can occur more than once in the self-tuning process. Therefore this is the predominant self-tuning stage. For the extreme case where no tuning activity is needed only IR and SE stages (or, still, PAE, as argued Section 2.2) would occur.

2.4 Self-Tuning Global Model and Persistence

It is generally worthy, however difficult, to make self-tuning component learn with decisions previously made. This is important because what seems good for a DBMS's component can harm others and, consequently, harm the whole system performance. Moreover, as self-tuning is an automatization process, it is important to gain experience. Some self-tuning engines need to store, in a consistent way, data and statistics about decisions taken, as represented in Figure 4. In this figure the possibility of communication between self-tuning components is also represented. This communication can be very useful, even if the system becomes more complex.

For example, consider two self-tuning components, one taking care of the indexes existence and another that takes care of memory's allocation and page's substitution policy. The creation or exclusion of an index can intervene, for example, in the choice of a join operation execution method. This could modify, for example, the memory area for sort operations. If the index self-tuning component informs the memory's

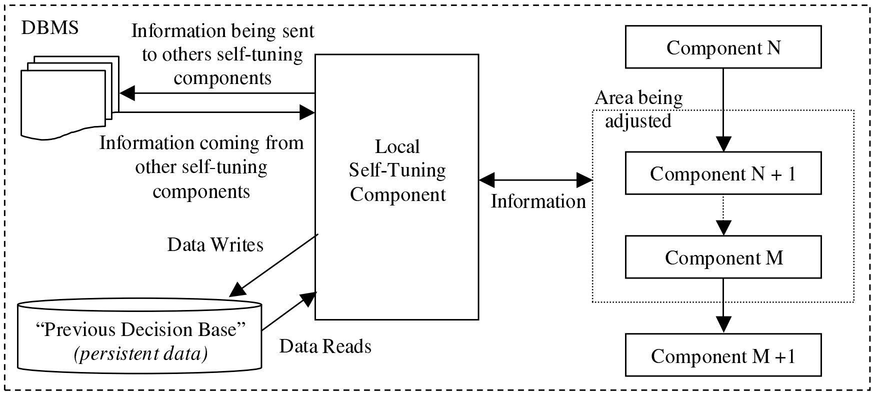
DBMS
Information being sent to others self-tuning components
Information coming from other self-tuning components
Local Self-Tuning Component
Data Writes
Data Reads
"Previous Decision Base" (persistent data)
Information
Area being adjusted
Component N
Component N + 1
Component M
Component M +1

Figure 4: Self-Tuning global model

component on decisions taken, alterations could be made preventively, before a problem occurred.

2.5 Self-Tuning Agent

The self-tuning component must possess:

- Autonomy: capacity of act/react to reach an objective without any intervention;
- Reactivity: by capturing what occurs in the environment, it is the capacity of responding to changes so that its objectives are reached;
- Pro-activity: to act in order to reach its objectives, anticipating changes in the environment;
- Sociability: capacity to interact with other participants of the environment;

Beyond these, the possibility of learning also exists: a decision in a given moment can be different from a decision made previously, according with the consequences measured in the first decision. These are exactly the features of intelligent software agents [25, 26].

Therefore we claim that an agent could act as a self-tuning component. We will give in Section 4 an agent architecture that will satisfies these features of a DBMS's self-tuning component.

2.6 Self-Tuning with Multiple Agents

A DBMS is a very complex system for a stand-alone agent to take care of all system's self-tuning tasks. In practice, we need a multiple agents system, which cooperate to achieve their particular goals.

In a agent-based database system, we could create different self-tuning agents specialized on different DBMS's tasks. Interaction techniques enable jobs exchanges between tuning agents of different DBMS's areas.

The interaction could be only for information flow on observed situations. Let us consider that an agent perceive that the response time for a join operation is very high. This agent sends then a message to the index and memory agents and to another one responsible for join operations. The index agent tests indexes creation. The memory agent can modify the memory size or modify the buffer page's substitution policy. The last one can qualify the hash join method and a new execution plan for the join operation can be generated.

This type of action is exclusively reactive and it does not make use of the agent's pro-active capacity. In this situation, the ideal would be that the index had been created during the elaboration of the query execution plan. It would have to take into account the hash join technique and possible substitutions in the buffer's page replacement policy. A self-tuning agent with total autonomy and pro-activity for the creation/destruction of indexes will be presented in the next Section.

```
$C = C_R - C_H$;
If $C > CC$, then
        Create index
        Execute the query
Else
        Execute the query
        Previous operations evaluation
End-if;
```

a- Algorithm for queries evaluation

```
If $C_P + C > CC$ , then
        Create index
        Delete $C_P$
Else,
        $C_P = C_P + C$
End-if;
```

b- Previous operations evaluation

Figure 5: Queries and previous operations evaluation

3 Indexes Self-Tuning Based on Differences

In this section we describe a method that allows the total automatization of the choice, creation and exclusion of indexes. Process is done during DBMS's normal operation, without the intervention of the administrator, through the use of a self-tuning agent.

The Self-Tuning Agent Based on Differences will use hypothetical indexes. We will not detail the indexes choice engine, as this problem is already sufficiently known. We focus our attention will be given to the process that controls the creation and the exclusion of the indexes, exactly the most critical point of the problem. The whole process will be done through a agents based architecture.

3.1 General model

The main difficulty in the implementation of an automatic project of indexes creation and exclusion refers to the choice of the moment where indexes must be created/destroyed, since such operations imply in costs that cannot be forgotten. For one given operation submitted to the DBMS, the query optimizer generates the best plan with the best real configuration (it only uses objects materialized in the database) and its execution cost. Then, the self-tuning component will decide if hypothetical indexes may participate in an execution plan whose execution cost is lower than those of the real configurations. When such indexes do not exist, the execution of the operation continues normally. When good hypothetical indexes exist, these are submitted to the optimizer and a plan is generated according to a hypothetical configuration.

The costs of the best plan according to real configuration (C_R) and of the best plan according to hypothetical configuration (C_H) are compared and a decision on creation/destruction of an index is taken. A factor C is defined as the difference between the cost of the best plan according to real configuration and the cost of the best-established plan according to hypothetical configuration.

3.2 Queries costs evaluation strategy

For a query operation, when C_H is bigger than C_R, the query is executed according to real configuration. On the other hand, in case that C factor is positive, we compare it with the cost of creation of the necessary indexes used in hypothetical configuration, called CC (creation cost). If C is lower than CC, then it will be advantageous to create the indexes before the query execution. These will be created and the query executed. On the other hand, if C is higher than CC the query will be executed without the index creation and, then, an evaluation of previous operations will be carried through, in accordance with the stored statistics. This procedure is represented in Figure 5a.

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 5b, where C_P is the stored value representing the total cost's reduction if the related index had already been created.

```
$C = C_R - C_H$;
If $C > 0$
        If $C > CC$, then
                Create index
                Execute the operation
        Else,
                Execute the operation
                Previous operations evaluation
        End-if
Else,
        $C_P = C_P - C$;
        If $|C_P| = CC$, then,
                If index exists, destroy index
                Execute the operation
                Delete $C_P$
        End-if
End-if
```

Figure 6: Engine of costs evaluation for updates/exclusions

3.3 Updates/exclusions costs evaluation strategy

The case of updates and exclusions is a little more complex than the one of queries: actions should be taken no matter C is positive or negative. When C is positive, the actions to be taken are similar to those taken in the query operations evaluation. It means we must evaluate if C will be greater that the indexes creation cost, when we create them and later we carry through the query. The other way around, we only make an evaluation of previous operations, similar to the one proposed in Section 3.2.

When the existence of the index is not beneficial, if C is negative, we must also evaluate the information stored about the indexes in the hypothetical configuration[2] . Then, we update the value of C_P deducting $|C|$. Thus, we made the indexes creation more difficult, since they cause performance degradation.

However, it is still possible the index exists and is harming the update performance. In this situation, we reduce from C_P the value of $|C|$. When $|CP|$ reaches the cost of indexes creation, these are destroyed. The procedure is represented in Figure 6. Previous operations evaluation will be the one depicted at Figure 5b.

4 Implementation Issues

In this section we will present some issues related to the implementation of a self-tuning agent. We will show an agent architecture that makes possible to implement all the self-tuning characteristics presented in previous sections.

To implement the self-tuning agent we have chosen a built-in agent-based database architecture. It is the one that better fits in our approaches as we do not intend to direct change DBMSs components functionalities, only extend them.

The layered agent presented in Section 2 can be used as we can attribute all the self-tuning process stages described in Section 2.2 to agent's layers, as shown in Figure 7.

The Sensory layer will be responsible for Information Retrieval. The Reasoning layer does the Situation Evaluation and Possible Alterations Enumeration. This layer will be the one where the greater processing activities will take place. The Action layer will receive an intentions plan from the Reasoning layer, and will transform it into attitudes to be carried through and will also pass these to the Translation layer, which will transform the attitudes into commands that the DBMS's components can understand. These are passed to the layer Mobility that executes them, finishing with the Alterations Accomplishment phase. The Believe and Collaboration, Translation and Mobility layers provide the functionalities of learning and communication between components, respectively.

The global self-tuning system can be implemented by the use of Collaboration, Translation and Mobility layers, what allows better tuning results. The architecture presented in Figure 8 presents this cooperation.

[2]Until this point all the hypothetical configurations had more indexes than the real ones.

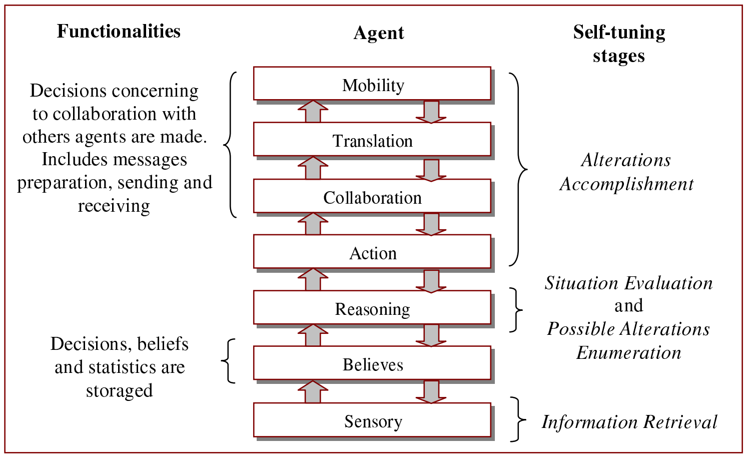
Functionalities
Agent
Self-tuning stages
Decisions concerning to collaboration with others agents are made. Includes messages preparation, sending and receiving
Mobility
Translation
Collaboration
Action
Reasoning
Believes
Sensory
Alterations Accomplishment
Situation Evaluation and Possible Alterations Enumeration
Decisions, beliefs and statistics are storaged
Information Retrieval

Figure 7: The self-tuning agent layer's functionalities

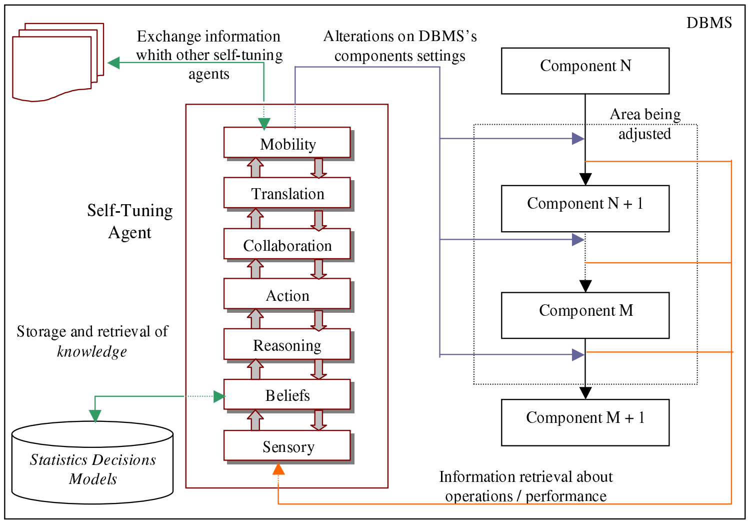
DBMS
Exchange information whith other self-tuning agents
Alterations on DBMS's components settings
Component N
Area being adjusted
Mobility
Translation
Collaboration
Action
Reasoning
Beliefs
Sensory
Self-Tuning Agent
Storage and retrieval of knowledge
Statistics Decisions Models
Component N + 1
Component M
Component M + 1
Information retrieval about operations / performance

Figure 8: DBMS's Self-Tuning Architecture with cooperation between agents

Although systems with some agents cooperating between themselves bring a series of advantages, they also bring some implementation challenges.

A self-adjust multiple-agents systems allows that the existing studies on self-tuning are implemented separately; each developed engine is transformed into an agent. Thus, the existing studies would have part of the four inferior layers ready, as well as the stream of data that has broken of Sensory in route to Mobility, until the Action layer. It would be enough that, for the integration of the components, the interaction policy between the agents were defined. These would be implemented in the three upper layers and the stream of information would be from the upper layer to the bottom one.

These definitions are not trivial. First, interfaces for the communication between agents must be defined. Then, each agent will have to know which agent must be notified on its actions, intentions or observations ("think globally, fix locally"). Moreover, in a system with some agents, each can need to keep Believes on itself, but, also, on the states and actions of the other agents [3].

In [17] an implementation of an agent-based database is briefly described. The agents were placed in Minibase [19], a public domain database management system, in a built-in architecture. They were used to act in parallel join load balancing in a ARCOJP Architecture [12] implementation.

The *Self-Tuning Agent Based on Differences* that implements the engine described in the Section 3 fits very well in the layered agent architecture. The main functions of each layer are listed below:

- Sensory: Measures carried through in the DBMS, such as, assembled queries submitted and execution plans, with its costs and used indexes;
- Believe: It controls the storage and backup of C_P;
- Reasoning: It verifies the necessity of index creation (it verifies existing indexes). Generation of possible alterations based on indexes suggestion. Plan of action: immediate/posterior indexes creation/destruction with statistics update, or continuation of the DBMS operations without any interference;
- Action: It transforms the decisions taken from Reasoning into a series of logical commands. For example: for a request of costs for one given hypothetical configuration, it detects the need of activation of the hypothetical index and submission of the query to the optimizer with the new active hypothetical configuration;
- Collaboration: Possibilities of collaboration with other agents are analyzed. Responses to other agents requests are evaluated;
- Translation: It translates commands for other agents languages and for DBMS's calls for components procedures;
- Mobility: Communication by sending messages between agents in different places and calls for DBMS's components procedures;

We can state that the information used in the evaluation processes of the previous operations, nominated values of C_P, are stored as acquired knowledge. The Believes layer will do its manipulation.

A possible objection here would be the eventual need of extra storage space to contain information of all the indexes tested for the system. Some statistics will remain much time without being brought up to date. In case there is a physical limitation for the storage space, the information that is least recently used could be being extinguished without causing great losses.

The Self-Tuning Agent Based on Differences will be available to cooperate with others agents receiving and analyzing two types of requests: one for storage space organization and another for index creation.

As long as this agent is pro-active for indexes creation (whenever the agent believes that the existence of determined index is good/harmful for the system's performance, it creates/destroys the index at the right moment), when a request of indexes creation/destruction is received, the agent already evaluated the possibility of indexes creation/destruction indexes for the attributes in question. There are three possible responses:

- No index is beneficial for the query in question and either no index will be created for a creation request, or no index will be destroyed, for a destruction request;
- The indexes creation or destruction is already determined and will happen at an adequate moment - for creation or destruction requests, respectively;
- Indexes already had been created/destructed, taking care of the request before it was done;

As a complement, aiming at a better understanding of the query or update operation cycle in a DBMS in the presence of the self-tuning agent, the life cycle is illustrated in Figure 9. There are some blocks separated and identified by the most important layers in each block. When an operation is submitted to the DBMS and its execution plan is generated (according to real configuration), the agent acts with the DBMS's operations before code generation. Statistics are accessed by Believes. Reasoning can decide that the real plan is the best one to be executed and, then, makes with that Mobility initiates the code generation operation (breaking the cycle). In case that Reasoning needs more information on execution plans of a hypothetical configuration, Mobility will request that such plan is assembled. In this case, Sensory will get information on the generated plan, closing the cycle. Believes will access statistics on this hypothetical configuration. Reasoning will have, now, new information to use in its decision-making. This cycle will happen again until Reasoning decides what to do, which will be passed to the upper layers until Mobility. This will interact with the DBMS's components, executing the definitive actions and going off the code generation.

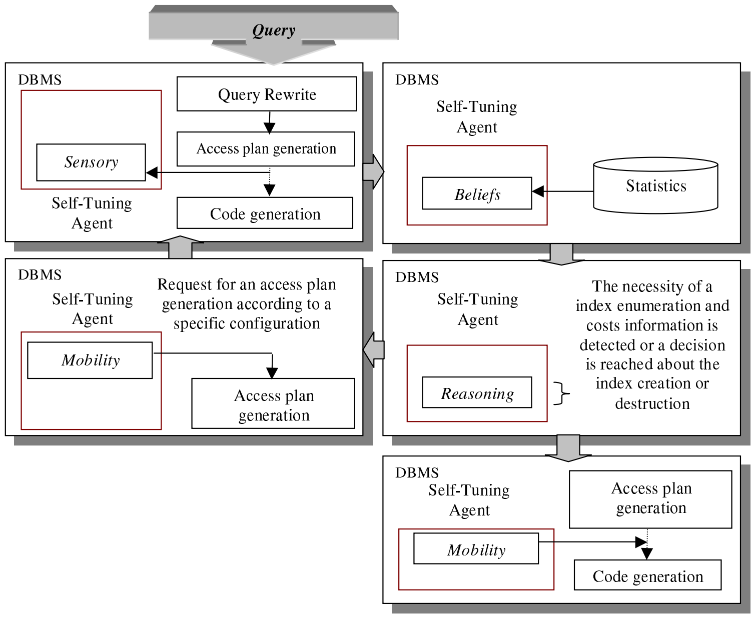
Query
DBMS
Query Rewrite
Access plan generation
Sensory
Code generation
Self-Tuning Agent
Beliefs
Statistics
Request for an access plan generation according to a specific configuration
Mobility
Access plan generation
The necessity of a index enumeration and costs information is detected or a decision is reached about the index creation or destruction
Reasoning

Figure 9: The cycle of a query/update operation

5 Conclusions

In this work we have defined the stages of a self-tuning process and we have showed how to use an intelligent software agent to accomplish it. We apply this agent in the DBMS's context and show the need of some cooperative agents. We also define precisely a DBMS's self-tuning architecture. The works described in Section 2 represent a small part of the operations to be carried through for each indexes self-tuning agent defined in Section 4. It can be observed, therefore, that they are not properly self-tuning engines, but, tools that help the database administrator to decide upon tuning the system.

We define the structure of an indexes self-tuning agent in accordance to the architecture proposed. An engine that enable the automatic indexes creation/destruction, from the evaluation of the indexes creation/destruction costs and the benefits (or not) of the indexes presence is presented. In this process the indexes creation (destruction) costs that do not exist (and also for the existing ones) in the DBMS are considered.

As main contributions we can mention the idea to use an agent-based database architecture implement the self-tuning process and the proposal of a generic architecture, validated by a detailed case study. As future works there are some possibilities already being investigated. It is interesting to study communication interface between self-tuning agents their cooperation policy. Also we intend to implement this architecture as it was done in [17] and be able to make a detailed performance study, comparing them with the possibilities offered by the commercial DBMSs.

References

[1] Agrawal, S.; Chaudhuri, S. and Narasayya, V., Automated Selection of Materialized Views and Indexes for SQL Databases, Procs 26rd VLDB Intl Conference, 2000, pp 496–505.

[2] Bernstein, P.; Brodie, M.; Ceri, S.; Dewitt, D.; Franklin, M.; Garcia-Molina, H.; Gray, J.; Held, J.; Hellerstein, J.; Jagadish, H.; Lesk, M.; Maier, D.; Naughton, J.; Pirahesh, H.; Stonebraker, M. and Ullman, J., The Asilomar Report on Database Research, ACM SIGMOD Record 27(4), 1998, pp 74–80.

[3] Bailey, J. A.; Georgeff, M.; Kemp, D. B.; Kinny, D. and Ramamohanarao, K., Active databases and agent systems - a comparison, Procs 2nd Intl Workshop on Rules in Database Systems, LNCS 985, 1995, pp 342–356.

[4] Bigus, J. P.; Hellerstein, J. L. and Squillante M. S., Auto Tune: A generic Agent for Automated Performance Tuning, Procs 5th Intl Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology, 2000, pp 33–52.

[5] Bradshaw, J.: Software Agents, MIT Press, 1997.

[6] Costa, R.L.C. and Lifschitz, S., An Agent-Based DBMS Architecture for Index Self-Tuning, Monografia da Ciência da Computação 28/01, Departamento de Informática - PUC-Rio, 2001 (in portuguese).

[7] Chaudhuri, S. and Narasayya, V., An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server, Procs 23rd VLDB Conference, 1997, pp 146–155.

[8] Chaudhuri, S. and Narasayya, V., AutoAdmin "What-if" Index Analysis Utility, Procs ACM SIGMOD Intl Conference on Management of Data, 1998, pp 367–377.

[9] Chaudhuri, S. and Narasayya, V., Microsoft Index Tuning Wizard for SQL Server 7.0, Procs ACM SIGMOD Intl Conference on Management of Data, 1998, pp 553–554.

[10] Chaudhuri, S. and Weikum, G., Rethinking Database System Architecture: Towards a Self-tuning RISC-style Database System, Procs 26rd VLDB Intl Conference, 2000, pp 1–10.

[11] Frank, M.; Omiecinski, E. and Navathe, S., Adaptive and Automated Index Selection in RDBMS, Procs Intl Conference on Extending Data Base Technology, 1992, pp 277–292.

[12] Freitas, F., Lifschitz, S. and Macêdo, J.A.F.: ARCOJP: An Architecture for Comparing Joins in Parallel, Procs XVI Brazilian Symposium of Databases (SBBD), 2001, pp 286–300.

[13] Fayad, M., Schmidt, D.C. and Johnson, R.: Implementing Applications Frameworks: Object Oriented Frameworks, Wiley & Sons, 1999.

[14] Gilbert, D.: IBM Intelligent Agent Strategy, IBM Corporation. 1995.

[15] Kendall, E. Krishna, P. Murali, P. Chirag and Suresh, C.B. Implementing Application Frameworks. Wiley. 1999.

[16] Lee, M.L.; Kitsuregawa, M.; Ooi, B.C.; Tan, K. and Mondal, A., Towards Self-Tuning Data Placement in Parallel Database Systems, Procs ACM SIGMOD Intl Conference on Management of Data, 2000, pp 225–236.

[17] Lifschitz, S. and Macêdo, J.A.F., Agent-based Databases and Parallel Join Load Balancing, XXVII Latin Conference on Informatics (CLEI), 2001, 15pp CD-ROM Procs, pp 47 Abstracts Procs.

[18] Nwana, Hyacinth; Software Agents: An Overview, Cambrige University Press; 1996.

[19] Ramakrishnan, R.; The Minibase Software, em Database Management Systems; apndice A, McGraw-Hill, 1998, pp 692–695.

[20] Rozen, S. and Shasha, D.; A Framework for automating Physical Database Design, Procs 17th VLDB Conference, 1991, pp 401–411.

[21] Shasha, D., Tuning Databases - A Principled Approach, Prentice Hall, 1992.

[22] Stonebraker, M. Object-Relational DBMSs The Next Great Wave; Morgan Kaufmann; 1996.

[23] Widom, J. and Ceri, S. Active Database Systems. Morgan Kaufman Publishers. 1996.

[24] Weikum, G.; Hasse, C.; Mnkeberg, A. and Zabback, P., The Confort Automatic Tuning Project, Information Systems 19(5), 1994, pp 381–423.

[25] Wooldridge, M. and Jennings, N. R., Intelligent Agents: Theory and Practice, Knowledge Engineering Review 10, 1995, pp 115–152.

[26] Wooldridge, M., Intelligent Agents, chapter in Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence, ed. Gerhard Weiss, The MIT Press, 1999.