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

and

Marcos Antonio Vaz Salles
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: mvsalles@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 architecture to deal with automatic index creation. Implementation issues are also discussed, for a built-in agents and DBMS integration architecture.

1 Introduction

The software agents' research area aims at building systems that deal with heterogeneous, distributed and dynamic environments [33]. The agent technology has been used in many complex environments [4, 32]. 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 autonomy and reactivity are very important [6].

In general, Database Management Systems (DBMSs) are known as passive systems that become active only in response to requests from end users or application programs. A possible approach is to make use of the agents' technology to add a reactive capacity to the system, that enables autonomous reconfiguration and extensibility.

Extensions due to new systems requirements have created multiple extended DBMS instances, such as active database systems [30]. Active databases and software agent systems are quite similar as both may be used on reactive applications [4]. However, active rules are usually limited to deal with database internals, like its objects and transactions, while agents may apply to the whole system environment.

There are many application domains for which Database Management Systems (DBMS) could be extended and configured [28]. 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. However, data and transactions are continuously evolving and a DBMS initial configuration may not anymore be effective. When modifications are imperative, the DBMS should then be able to execute them automatically. This capacity of perception and automatic adjustment is known as self-tuning. In this paper we propose the use of software agents to deal with self-tuning and DBMSs operational requirements.

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 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" [27]. 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 for parallel machines and systems, that bring new questions such as the data allocation in multiple disks. Moreover, at each new release of commercial DBMSs, additional operational parameters appear to be adjusted. So, tuning becomes even more important and, as well, more expensive, due to the need of highly specialized professionals [3, 12].

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 [12]. 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 [17].

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. This paper extends the work presented in [8], showing revised heuristics for an index selection agent's decision process and discussing an object-oriented design for the implementation of this agent in a real DBMS.

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 4. Then, in Section 5, we discuss implementation and architectural issues. Finally, we conclude in Section 6 listing our contributions and comment on future and ongoing work.

2   Agent-based Databases

There exist multiple definitions for the agent term [33, 6, 16, 23] 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 [33]: 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.

2.1   Software Agents

The agent architecture that will be implemented here is based on an object-oriented framework for building agent systems proposed in [17]. This framework defines a layered architecture (Figure 1) 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 other agents/environment 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.
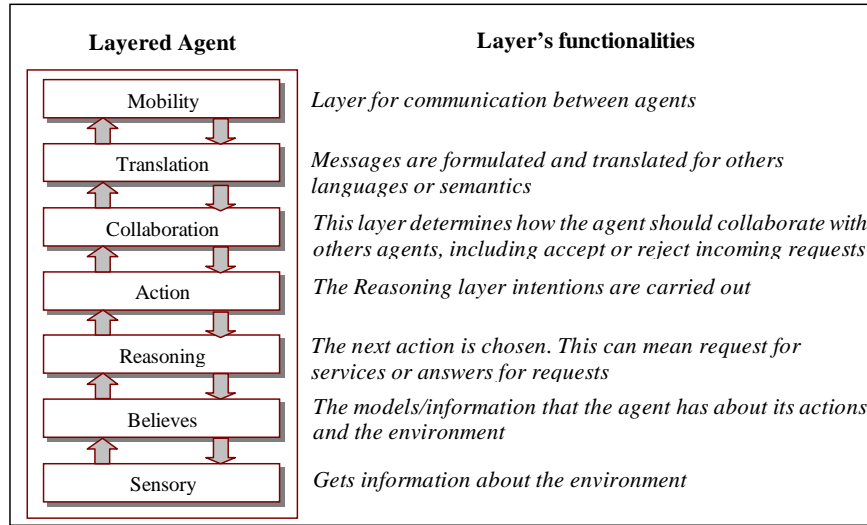
Figure 1: Layered Architecture for Software Agents

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.

2.2 Agents and Database Systems

In [4] 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 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.

The extension of the agency level of an active database is investigated in [2]. They use an active database system based on autonomous objects, each of which has data and goals to achieve. The extension of the agency level is obtained by changing objects to agents. This way the agent abstraction can be used to model applications and implement DBMS functionalities. Each DBMS component could be an agent with own strategies and goals. This work was maybe the first one to argue in favor of an agent-based database. Nevertheless, for the best of our knowledge, this architecture was never implemented.

In [22] an agent implementation model based on a DBMS layered architecture is proposed. They claim it is a good idea since the system would get flexibility with agents and robustness with a DBMS. All agent operations are mapped to DBMS operations, i.e., the DBMS scheduler manages every possible conflict. The cooperation among agents is executed through a nested transaction model. Each subtransaction is executed as regular DBMS transactions. The goal of the proposed model is to permit the representation of a layered architecture for building agents which considers these nested transactions.

Another important work describes a formal approach to agent-based federated DBMS design [29]. A federated DBMS is seen as an agent federation, with multiple agents as mediators. For each DBMS component, there is a mediator to access it. In a similar way, for each federated DBMS, there exists an agent which is responsible for constructing and maintaining the federation. Considering the decentralized approach, only the mediators communicate between themselves. Each mediator agent has the knowledge of all other agents that are part of the federation. The paper also discusses the fundamental properties of each agent.

Finally, we can cite the work in [18], where the authors extend the active capacity of a relational DBMS with an approach based on agents. The extension is done by the incorporation of a mediator between the DBMS and its users. The agent is responsible for rule creation, event notification and action firing. This is possible because the agent provides an user interface where E-C-A (Event-Condition-Action) rules may be
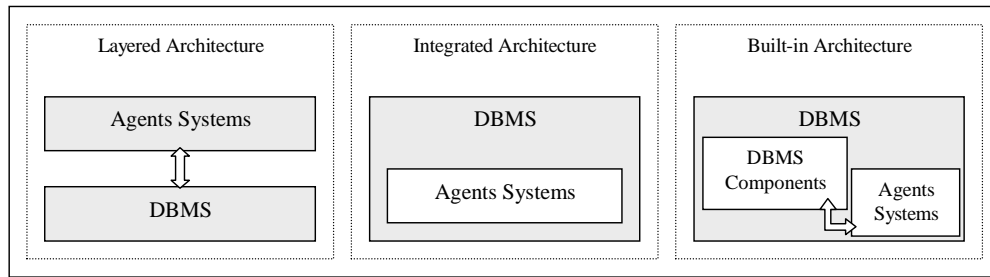
3

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

created, as well as events. Although they use the term agent, this work does not define the agency properties. An "agent" is presented as a multi-threaded program that defines an interface between users and the DBMS.

Some other details of these and other related works can be found in [21]. There we describe some practical aspects of agents and database integration. We will see next some proposed architectures to make agents and database systems work together.

2.3 Agent-based Database Architectures

In [20] 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 [20] 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 5.

3 Self-tuning

We describe here some self-tuning previous studies related to the our work. 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.

We will give in this section the main stages of an automatic adjustment engine and its insertion in the context of traditional DBMSs. The self-tuning process has been mainly studied in the evaluation of particular problems. We will discuss the importance of making the self-tuning engine a global process to the DBMS operation and not a specific process for a given problem or component.

3.1 Self-Tuning in Databases

The self-tuning feature has been studied in many different areas. Transactions tuning and memory management are studied in [31]. In [19] a self-tuning engine is presented for the data allocation problem in parallel systems. In [12] 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 [25].

The work presented in [13] generates index sets from the monitoring of DBMS activities in a determined time interval. The indexes selection is achieved by considering index "benefit graphs". These structures represent the index sets that are presented and chosen by the optimizer, together with their execution costs. Each of the index sets selected by the optimizer is viewed as a graph node. The construction of a benefit

graph for all the alternatives of index 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 [9, 10, 11, 1] 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. This workload could be built 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 is 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 [7]. 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.

3.2 Self-tuning stages

In [31] three main stages are identified 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 a 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 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.

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

---

[1]Hypothetical Indexes are indexes that do not physically exist in the database but are treated by the query optimizer as if they did [13]. The ones that belong to the database will be called Real Indexes.
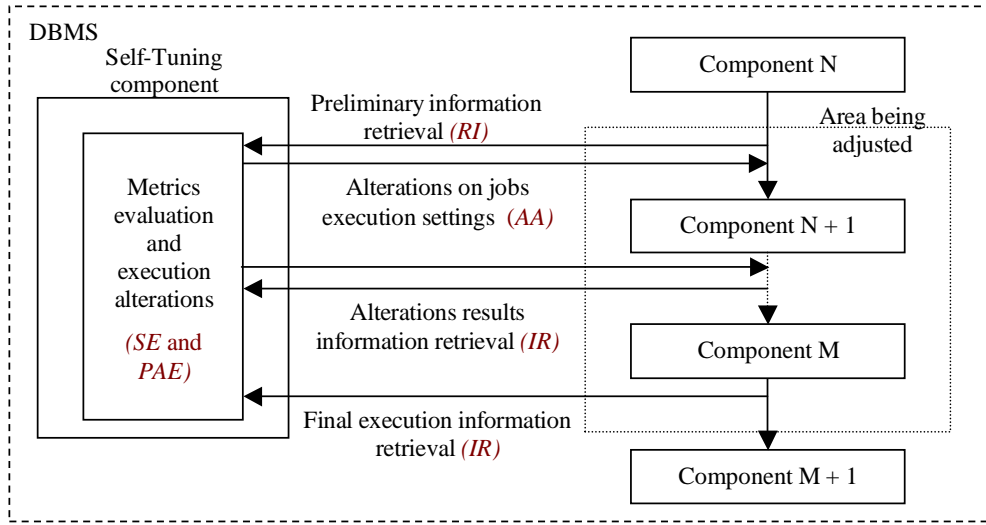
Figure 3: Component's Local Self-Tuning Architecture

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.

3.3 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 capturing 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 3.2) would occur.

3.4 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 component on decisions taken, alterations could be made preventively, before a problem occurred.
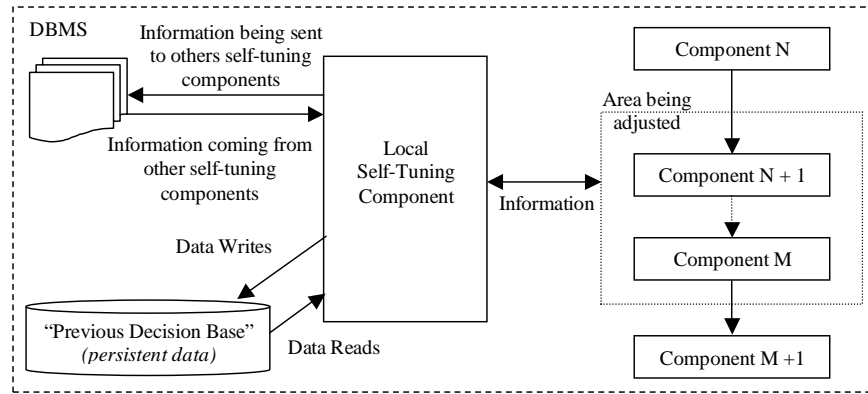
Figure 4: Self-Tuning global model

3.5 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 to the consequences measured in the first decision. These are exactly the features of intelligent software agents [32, 33].

Therefore we claim that an agent could act as a self-tuning component. We will give in Section 5 an agent architecture that will satisfy these features of a DBMS's self-tuning component.

3.6 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 an 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.

4 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. The 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 [7, 8] 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.

4.1 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. Some details on specific situations are presented in the next subsections.

4.2 Queries costs evaluation strategy

For a query operation, 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.

When the C factor is 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 higher 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 lower 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 5.

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

On the other hand, in case that C_H is equal or bigger than C_R, the query is executed according to real configuration. Note that if the optimizer always find the best plans, C_H would never be bigger than C_R, as C_H would represent the same plan as C_R. 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 the C_P factor of the used indexes the diference between the query execution cost in the real configuration and the query execution cost in a configuration where the indexes that are being used in the real configuration do not exist.

4.3 Updates/exclusions costs evaluation strategy

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 destrucion cost of an index. As presented before, CC is the creation cost of an index.

When C is negative or equal to zero, the actions to be taken are similar to those taken in the query operations evaluation. We must add to the C_P factor of the used indexes the difference between the execution

```
C = C_R – C_H ;
If C > 0, then
        If C > CC, then
                Create Indexes;
                Execute the query;
        Else
                Execute the query;
                C_P = C_P + C;
                If C_P >= CC then
                        Create indexes;
                        Reset C_P;
                End if;
        End if;
Else
        C_RI = Cost of execution if the indexes
                used in the real configuration
                were dropped;
        C_P = C_P + C_RI ;
End if;
```

Figure 5: Queries and previous operations evaluation

cost in the real configuration and the execution cost in a configuration where the indexes that are being used in the real configuration do not exist.

On the other hand, when C is positive, we should evaluate if C will be greater then 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 then $CC + CD$, then we should obtain C_{HC} that is the cost of execution of the operation if the sugested indexes are created but no index is dropped. $C_{HC} + CC$ should be compared to C_R . If C_{HC} + CC is lower then C_R the indexes are created and the update/delete is executed. If $C_{HC} + CC$ is not lower then C_R, then we should obtain C_{HD}. C_{HD} is the cost of execution of the operation if the sugested indexes are destructed but no index is created. If $C_{HD} + CD$ is lower then C_R 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_R - C_{HC}); (ii) for the indexes proposed which were proposed to be dropped, we update the value of C_P deducting (C_R - 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 6.

5   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. We will also present the UML design for the implementation and a discussion on an implementation on PostgreSQL.

5.1   Architecture

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 directly change DBMS's components functionalities, only extend them.

```
C = C_R − C_H ;
If C > 0, then
        If C > CC + CD, then
                Create proposed indexes;
                Drop indexes that were proposed
                        to be dropped;
                Execute the query;
        Else
                If C_HC + CC < C_R then
                        Create proposed indexes;
                        Execute the query;
                        Reset C_P;
                        Exit;
                End if;

                If C_HD + CD < C_R then
                        Drop proposed indexes;
                        Execute the query;
                        Reset C_P;
                        Exit;
                End if;

                Execute the query;

                C_P = C_P + (C_R - C_HC);

                C_P = C_P - (C_R - C_HD');

                If |C_P| >= |CC|  and C_P > 0 then
                        Create index;
                        Reset C_P;
                End if;

                If |C_P| >= |CD|  and C_P < 0 then
                        Drop index;
                        Reset C_P;
                End if;
        End if;
  Else
        C_RI = Cost of execution if the indexes
                used in the real configuration
                were dropped;

        C_P = C_P + C_RI ;
End if;
```

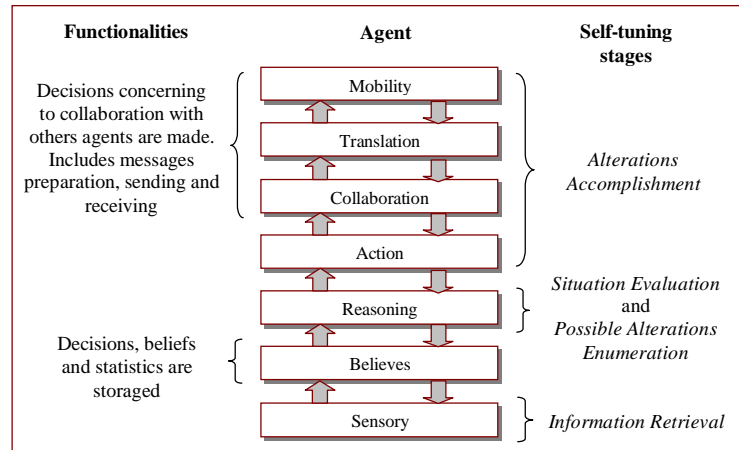Figure 6: Engine of costs evaluation for updates/exclusions

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

The layered agent presented in Section 2 can be used as we can attribute all the self-tuning process stages described in Section 3.2 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. 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 [4].

In [20] an implementation of an agent-based database is briefly described. The agents were placed in Minibase [24], 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 [14] implementation.

Agent Layers

The *Self-Tuning Agent Based on Differences* that implements the engine described in the Section 4 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;

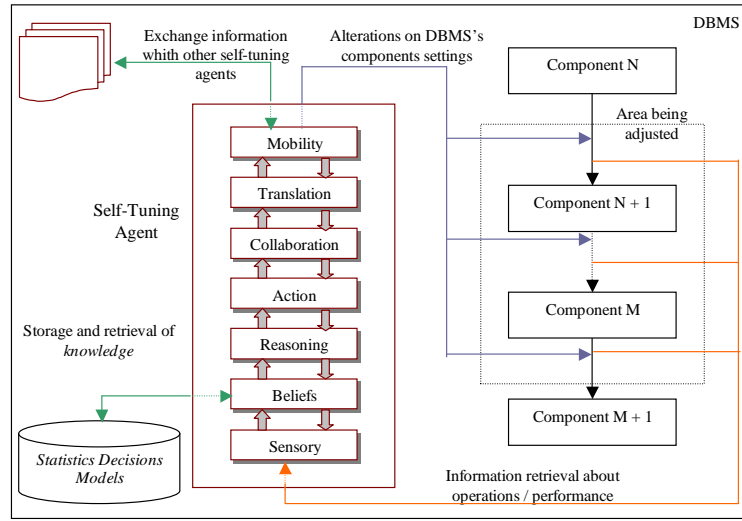- Believe: It controls the storage and backup of C_P;

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

- 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 least recently information could be discarded.

Life Cycle

As long as our 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:

- After having analyzed the queries and updates submitted to the system, the agent considers that either no index should be created for a creation request, or no index should be destroyed, for a destruction request;
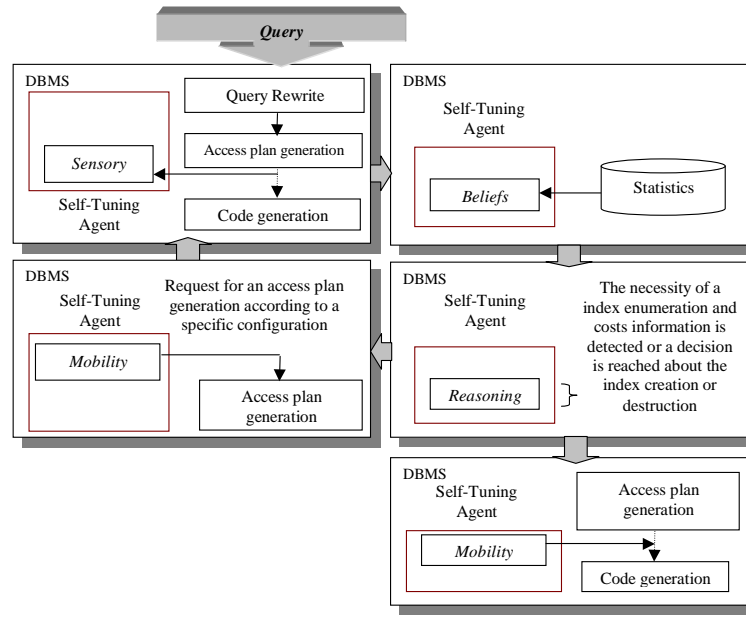
Figure 9: The cycle of a query/update operation

- 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.

But in the implementation considered in the next section, there is no need for cooperation between agents. So, Collaboration, Translation and Mobility are not present.

5.2 Agent Design

In this section, we present the UML design of the *Self-Tuning Agent Based on Differences* described previously. The design follows the ideas in [15]. As we have seen in Section 5, the Collaboration, Translation and Mobility layers in the agent's architecture deal primarily with the interaction with other agents possibly present in the database system. In our design, we focus on the implementation of the local self-tuning model shown in Section 3.3. Therefore, only the Sensory, Belief, Reasoning and Action layers are present. Below, we give a detailed description of the implementation rationale of each of these layers.

Sensory Layer

The Sensory layer is responsible for capturing the necessary information from the agent's environment. Figure 10 shows the classes present in this layer. The classes SQLInformation and BeliefFacade are not

actually part of the Sensory layer, but are shown in the diagram to ease understanding.
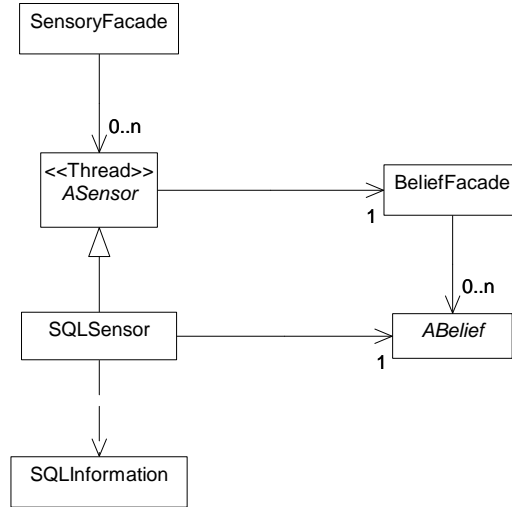

Figure 10: The Sensory layer classes

The class SQLInformation is responsible for the integration of the agent with the system's optimizer. We use this class to encapsulate the information retrieved from the DBMS. In this fashion, we can adapt the agent to distinct DBMS implementations and concentrate the changes in a few adapter classes. The code of the optimizer must be altered to inform to a SQLInformation instance the access plan used in the query along with its cost. As we will see in Section 5.2, it must also be altered to give the optimizer the ability to generate an access plan with a hypothetical configuration.

SQLSensor extends the abstract class ASensor. The existence of ASensor allows easy extension of the Sensory layer to include new sensors, if necessary. For example, we could include a sensor to monitor the amount of system resources used, such as CPU and memory. In the event of excessive resource consumption, the Reasoning layer could make a decision to stop the agent. Another useful sensoring ability that could be included is the measurement of the amount of free disk space in the system. This information can be used to avoid index creations that would compromise disk availability.

One important decision regarding the SQLSensor class is how to make it sensible to changes in the DBMS internal variables. There are two basic options to consider. In the first one, the sensors continually poll the DBMS's internal structures and check for changes. This can create considerable computational overhead if the polling interval is short. The second option is to instrument the DBMS to notify the sensors with fresh information. We belive this option brings advantages over the polling alternative because it is allows greater performance and more flexibility in the integration of the agent threads with the DBMS process model.

All the layers use the Facade design pattern [35]. This pattern creates a unified point of access to the main functionalities offered by each layer. In the Sensory layer, this is implemented through the SensoryFacade class. All the interaction with the Beliefs layer is done through the BeliefFacade. The use of the Facade design pattern thus decouples the layers, permitting each layer's implementation to occur with no dependency on the internal details of other layers.

In figure 11, we can see a typical interaction among the classes described. When the DBMS processes a new SQL statement, an instance of the SQLSensor class is notified. This implementation of this notification mechanism depends on the inner workings of the database system. As we will discuss later in this section, we have implemented this mechanism in the open-source DBMS PostgreSQL.

The SQLSensor instance records the query's text, access plan and cost information in an SQLInformation object. This object is given to the facade of the Beliefs layer, that then modifies the agent's beliefs to take the new information into consideration.
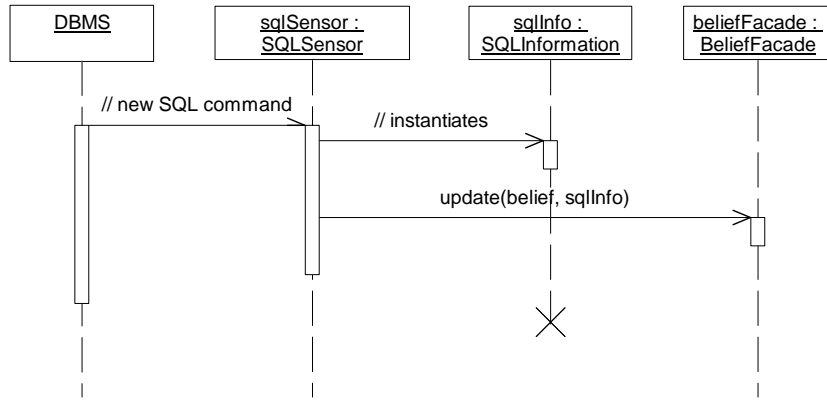
Figure 11: A typical interaction in the Sensory layer

Beliefs Layer

The Beliefs layer is responsible for representing the agent's knowledge about its environment and to store significant statistics derived from this knowledge. The Beliefs layer is also responsible for notifying the Reasoning layer about the availability of new information to be analyzed.

In Figure 12, we show the class model of the Beliefs layer. The class ReasoningFacade is not actually part of the Beliefs layer, but is shown on the diagram to illustrate the points of contact of this layer with its upper layer. The Observer and Subject classes are a generic implementation of the Observer design pattern [35]. In this pattern, a subject may have any number of dependent observers. Changes made to subjects are published to all of its observers, which are updated.
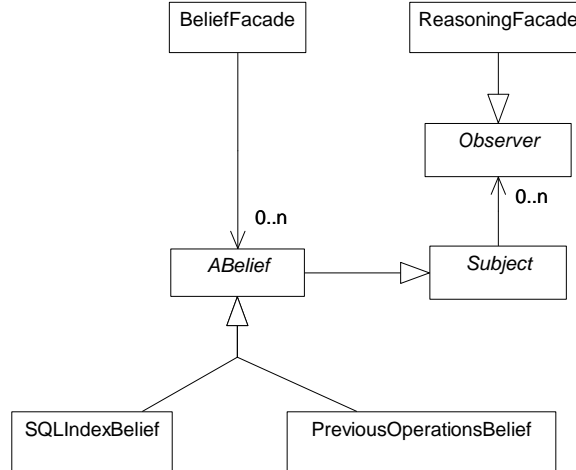

Figure 12: The Beliefs layer classes

The ABelief class hierarchy defines the types of believes that the Beliefs layer can represent. For the index selection problem, we must be able to represent a SQL query with associated cost and index set information. This is done by the SQLIndexBelief class. The PreviousOperationsBelief class offers a way to store and query the value of C_P for each index considered in the differences heuristic (see Section 4). It also stores other important information about the candidate indexes, such as their individual creation/destruction cost.

The BeliefFacade class offers a set of methods for believes management. Not only does it provide facilities for updating believes, but also it permits that any Observer registers itself to be notified of believes changes. The ReasoningFacade, in particular, observes all believes.

15

The instances of the classes derived from ABelief represent the knowledge accumulated by the agent so far. Every time an instance of ABelief changes (Subject), we notify the instance of ReasoningFacade of this happening (Observer). This flow of information can be better understood by looking at Figure 13.
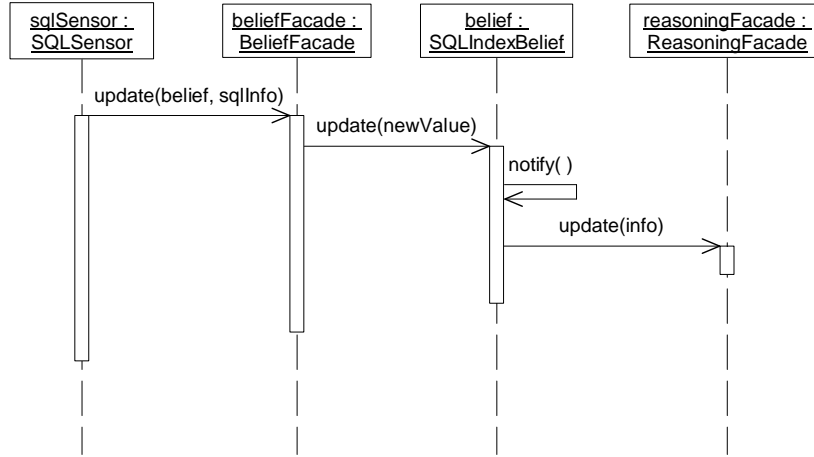

Figure 13: A typical interaction in the Beliefs layer

When the Sensor layer detects a new SQL statement submitted to the database, it forwards this information, represented as an instance of SQLInformation, to the BeliefFacade. The BeliefFacade object will search for the appropriate belief that represents the current SQL query submitted to the database. This is an instance of SQLIndexBelief. The belief will be updated with the value of the SQLInformation object and will, therefore, contain the most recent information sensored. This update will trigger a notification of the ReasoningFacade. On the Reasoning layer, the new SQL statement, with associated cost and index set information, will be analyzed and index creations or destructions will be decided.

Reasoning Layer

The Reasoning layer is responsible for deciding how the agent will interact with its environment in order to achieve its goals. In our design, this layer has weak agency characteristics [33]. We do not create an inferencing mechanism for the agent to derive its next steps. Instead, we systematically apply a set of heuristics specific to the index selection problem.

To better understand this idea, let us focus our attention in Figure 14, which illustrates the layer's class model. The APlan class hierarchy represents the heuristics that can be applied to the believes of the agent.

Its first subclass, CandidateIndexesGeneratorPlan, represents an appropriate hypothetical index generation heuristic. This kind of heuristic has been sufficiently discussed in the literature (we refer the reader to Section 3.1 for an indication of some related work). Basically, when a new SQL statement belief is received, this heuristic will conduct an enumeration of possible hypothetical index configurations. Each possible configuration must be costed by the optimizer for the given query.

In order to interact with the optimizer, the plan uses services available in the Action layer. The plan instantiates an ActionPlan instance and forwards the action plan to the ActionFacade instance for execution. The action plan contains a sequence of action requests. In this interaction, action requests are commands to create/drop hypothetical indexes and optimize the statement using the hypothetical configuration. The appropriate cost information is directly returned from the call to ActionFacade. After a number of hypothetical index configurations have been tested, the candidate indexes generator heuristic will select a given best configuration. This best configuration will be stored in the Belief layer through the BeliefFacade.

This is an example of an information flow from the upper layers of the agent down to lower layers of the agent. The cost information returned from the Action layer to the Reasoning layer will be used to prune hypothetical index configurations that do not represent performance gain. The Reasoning layer will then select an index configuration to be stored in the Belief layer. In the literature, the most usually implemented
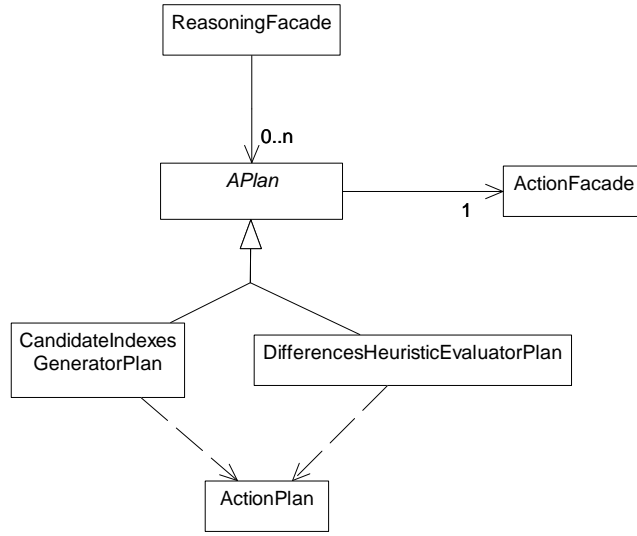
16

Figure 14: The Reasoning layer classes

information flow is from the Sensory layer to the upper layers of the agent. In our work, we not only implement this kind of information flow, but also an information flow in the opposite direction.

The storage of this hypothetical configuration in the Belief layer will trigger the plan represented by the class DifferencesHeuristicEvaluatorPlan. This plan is responsible for the evaluation of the differences model shown in Section 4. In case there is a need to create or drop indexes, the appropriate service will be requested by instantiating an action plan. So, once again, the information flows from the lower layers of the agent to its upper layers.

So, as we can notice, the APlan class hierarchy permits that we easily code new heuristics for the evaluation of candidate indexes once the agent is integrated with a real DBMS. This extensibility is very important to permit making investigations on the performance and quality trade-offs involved on using different heuristics for index enumeration and selection.

In Figure 15, we can see an interaction that causes index enumeration. First, a belief with information from the last executed SQL statement in the DBMS is updated in the Beliefs layer. As we discussed in the previous subsection, this will trigger the Subject/Observer mechanism, notifying the ReasoningFacade.

The ReasoningFacade will try to select all the plans that have some interest in this belief. We achieve this through the use of the Chain of Responsibility design pattern [35]. This design pattern decouples the belief update from the plans that will actually process the update.

In the case of the SQL statement belief, only the CandidateIndexesGeneratorPlan instance will evaluate itself, causing hypothetical index enumeration and further optimizer interaction through the use of an Action-Plan instance. When a best hypothetical configuration is found, we will update the PreviousOperationsBelief instance to reflect this new information.

This will, in turn, trigger a new cycle of plan evaluation on the Reasoning layer. This time, only the DifferencesHeuristicEvaluatorPlan instance will evaluate itself. If any index should be actually created or destroyed, an action plan will be instantiated and given to the Action layer.

Action Layer

The Action layer is responsible for the execution of the decisions made in the Reasoning layer. These decisions are the results of plan interpretation and are represented in the Action layer through intentions. There are two main types of intentions: collaboration intentions and reaction intentions. In our design, only reaction intentions are employed, because we do not yet interact with other self-tuning agents. Figure 16 shows the classes involved.

The AIntention class hierarchy is used to implement intentions. A ReactionIntention will be created
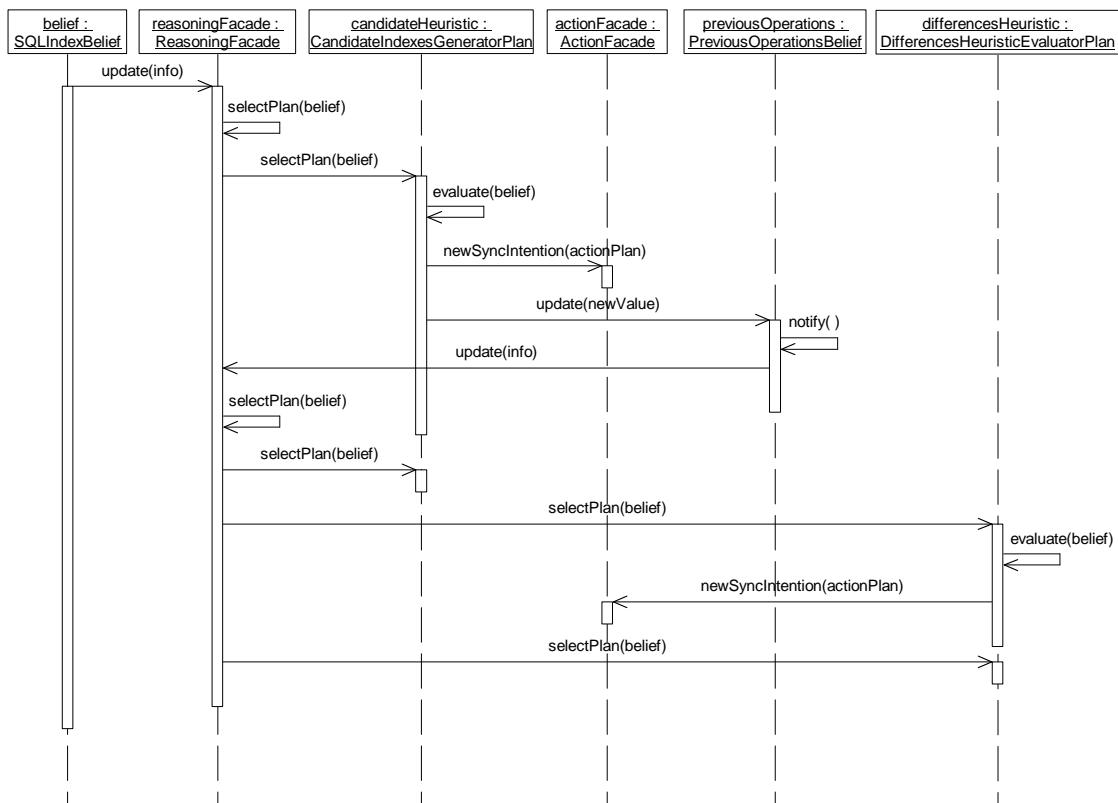
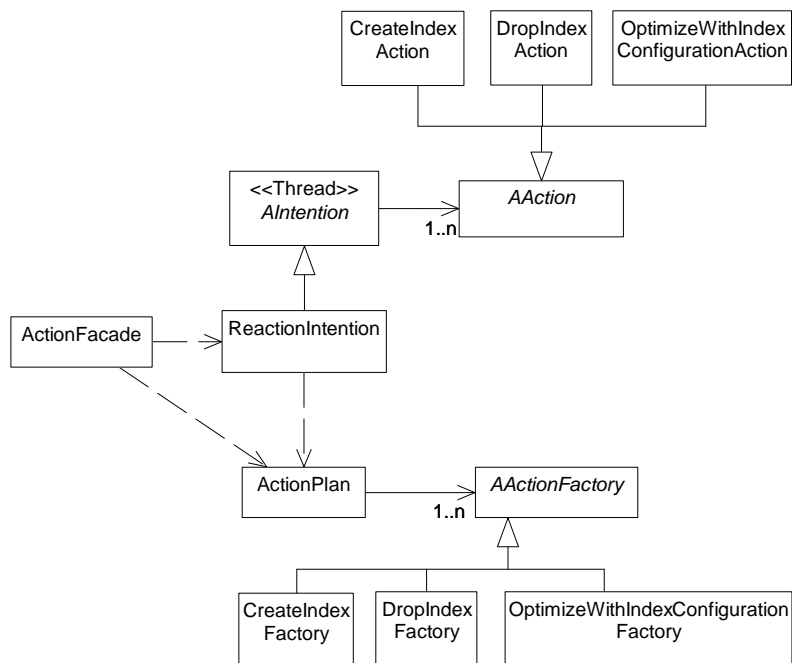Figure 15: A typical interaction in the Reasoning layer

Figure 16: The Action layer classes

by the ActionFacade every time an ActionPlan instance is received for processing. Action plans contain a sequence of AActionFactory objects. In the AActionFactory hierarchy we apply the Abstract Factory design pattern [35]. This allows us to create actions dynamically when the intention attempts to execute the action plan.

The AAction class hierarchy represents the actions that can be created by an action plan. These actions are concrete executions of interactions with the agent's environment. In our case, actions are responsible for the integration of the agent with DBMS components. Each given DBMS component, in this context, is said to be an effector of the action. The possible actions in our agent are:

- hypothetical and real index creations (CreateIndexAction class);

- hypothetical and real index destructions (DropIndexAction class);

- optimization of a SQL statement under the current hypothetical configuration defined in the DBMS (OptimizeWithIndexConfigurationAction class);

To carry off an intention and its associated actions, we use the Command design pattern [35], that encapsulates each action as an object. The intention is dynamically configured with action objects and the intention's execution is done by the execution of its component actions. To better grasp this concept, analyze the interaction shown in Figure 17.
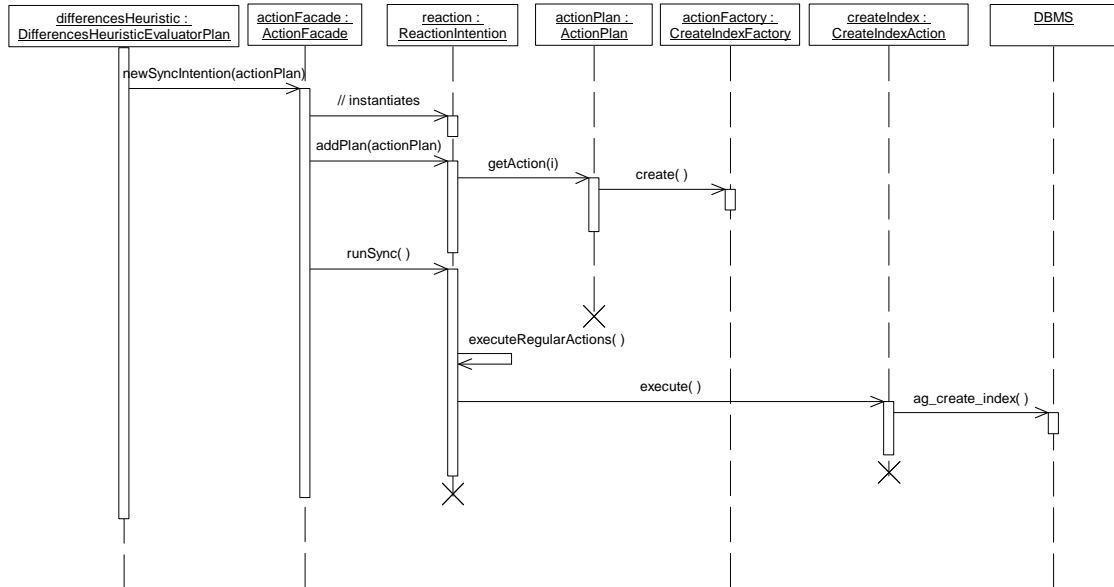

Figure 17: A typical interaction in the Action layer

When the differences heuristic plan decides that a new index configuration should actually be materialized, it passes an action plan to the ActionFacade. The ActionFacade then creates a ReactionIntention instance and configures it with the given ActionPlan instance. We will dynamically create each action in the actin plan and store these actions in the reaction intention.

Following, the reaction intention will be executed in a synchronous fashion. The execution of the intention is actually the execution of each of its actions. The actions interface with DBMS components to carry out the creation or destruction of indexes.

5.3 PostgreSQL Implementation Considerations

We have shown the design of all the layers that comprise the *Self-Tuning Agent Based on Differences*. However, we have not shown how to assemble those layers when the DBMS is started. We can see the necessary classes in Figure 18.

We introduce a new Agent class that keeps references to each layer's Facade. When the DBMS is started, if the agent is to be activated, an instance of the IndexAgent derived class is created and it takes care of the instantiation of the Facade classes. Each Facade class will create the necessary internal classes of each layer, in a recursive fashion. The IndexAgent instance will pass to the Facade instances configuration information in order to create the correct believes and plans for the agent.
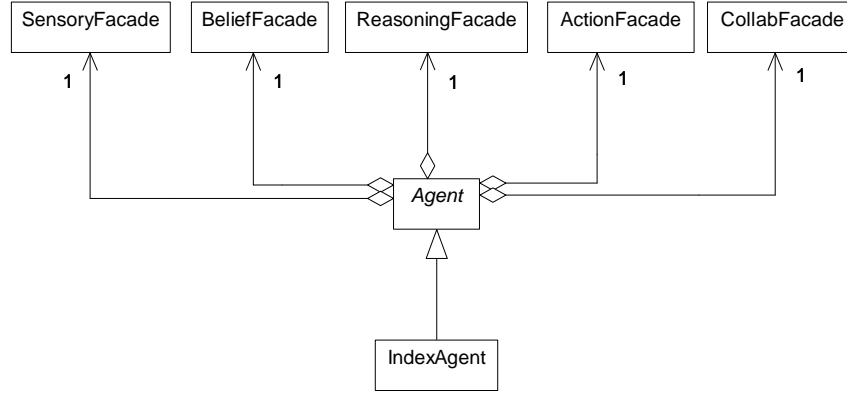

Figure 18: The Agent class

It is important to emphasize that, even though the agent is created in the DBMS's execution context, is not a module of the DBMS. The DBMS does not need the agent to carry out its tasks and the agent can be activated or deactivated causing no harm to DBMS functionality. The agent's goal is to improve DBMS performance through the tuning of the indexes used by the queries submitted.

The design we have shown can be used to integrate the agent with any standard relational DBMS. To achieve a full implementation, we must define the appropriate integration structures in the Sensory and Action layers. That demands some knowledge of the DBMS's internal structure. Presently, we are working with the PostgreSQL open-source database system [36]. Our goal is to build a fully operational *Self-Tuning Agent Based on Differences* for this DBMS and conduct detailed performance impact and index generation quality studies.

PostgreSQL follows a client-server model for communication. Each client is handled by a dedicated server process. A master process is created by the system to control all dedicated server processes and also to handle special tasks such as initialization, shutdown and checkpoints. It is important to integrate the agent with the system respecting this process model. The agent should be run as an independent process and comunicate, for example via message passing, with all the other processes in the system to obtain information about the SQL statements processed by the database system.

Another important point is that the database does not have at present the capability to optimize a query with a given hypothetical index configuration. Therefore, we plan to extended PostgreSQL to include hypothetical index creation and destruction commands. We also plan to extend the system's optimizer to permit the planning of a SQL statement using hypothetical indexes [26].

6 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. We have also presented an object-oriented design of the index tuning agent that indicates the feasibility of our implementation using a real DBMS. 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 [20] 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 Intl Conference on Very Large Databases (VLDB), 2000, pp 496–505.

[2] Akker, J. and Siebes, A. Enriching Active Databases with Agent Technology. Procs Intl Workshop on Cooperative Information Agents (CIA), LNCS 1202, 1997, pp 116–125.

[3] 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.

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

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

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

[7] 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).

[8] Costa, R.L.C. and Lifschitz, S., Index Self-Tuning and Agent-Based Databases, XXVIII Latin-American Conference on Informatics (CLEI), 2002, 12pp. CD-ROM Procs, pp 76 Abstracts Procs.

[9] Chaudhuri, S. and Narasayya, V., An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server, Procs Intl Conference on Very Large Databases (VLDB), 1997, pp 146–155.

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

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

[12] Chaudhuri, S. and Weikum, G., Rethinking Database System Architecture: Towards a Self-tuning RISC-style Database System, Procs Intl Conference on Very Large Databases (VLDB), 2000, pp 1–10.

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

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

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

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

[17] Kendall, E. Krishna, P. Murali, P. Chirag and Suresh, C.B. Implementing Application Frameworks. John Wiley & Sons, 1999.

[18] Li, L. and Chakravarthy, S. An Agent-Based Approach to Extending the Native Active Capability of Relational Database Systems; Procs IEEE Intl Conference on Data Engineering (ICDE), 1999. pp 384–291.

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

[20] 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.

[21] Macêdo, J., A Study of Agent-based DBMSs, MSc thesis (in portuguese), PUC-Rio - Departamento de Informática. September 2000.

[22] Nagi, K. and Lockemann, P.: An Implementation Model for Agents with Layered Architecture in a Transactional Database Environment. Procs Intl Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS) 1999.

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

[24] Ramakrishnan, R. and Gehrke, J.; Database Management Systems, McGraw-Hill, 2003.

[25] Rozen, S. and Shasha, D.; A Framework for Automating Physical Database Design, Procs Intl Conference on Very Large Databases (VLDB), 1991, pp 401–411.

[26] Salles, M., Autonomic Index Creation in Databases, MSc thesis in preparation (in portuguese), PUC-Rio - Departamento de Informática. To be published in 2004.

[27] Shasha, D. and Bonnet, P.; Database Tuning: Principles, Experiments and Troubleshooting Techniques, Morgan Kaufmann, 2003.

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

[29] Trker, C. Saake, G. and Conrad, S.: Modeling Database Federations in Terms of Evolving Agents. Procs Intl Symposium on Methodologies for Intelligent Systems (ISMIS),1997. pp 197–208.

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

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

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

[33] Wooldridge, M., Intelligent Agents, chapter in [34]

[34] Weiss, G. (editor); Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence, The MIT Press, 1999.

[35] Gamma, E.; Helm, R.; Johnson, R. and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.

[36] http://www.postgresql.org .