

## Towards Autonomic Index Maintenance

Marcos Antonio Vaz Salles<sup>1</sup>, Eduardo Terra Morelli<sup>2</sup>, Sérgio Lifschitz<sup>2</sup>

<sup>1</sup> Institute of Information Systems  
ETH Zurich, Switzerland

<sup>2</sup>Departamento de Informática  
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)

marcos.vazsalles@inf.ethz.ch, {emorelli,sergio}@inf.puc-rio.br

**Abstract.** *Automatic index selection has received significant attention in the autonomic computing field. Previous works have focused on providing tools and algorithms to help the DBA in the choice of indices for a given static workload. We present an approach for indexing management that works for workloads that may dynamically change with no human intervention at all. We automatically monitor statements submitted to the database with a built-in component, that interacts directly with the DBMS, creating and dropping indices on-the-fly. This paper presents a mechanism to integrate automatic index management components in relational databases. We discuss the heuristics considered and the observed experimental results for a PostgreSQL implementation.*

**Resumo.** *A seleção automática de índices é um tópico dos mais relevantes na área de computação autônoma. Os trabalhos existentes têm foco em ferramentas e algoritmos que ajudem o DBA na escolha de índices para uma dada carga estática. Nesse trabalho sugerimos uma abordagem para gerência de índices que funciona para cargas que mudam dinamicamente, sem intervenção humana. Há um monitoramento automático dos comandos submetidos para o banco de dados com um componente embutido, que interage diretamente com o SGBD, permitindo a criação e destruição de índices com a base em operação. Este artigo apresenta um mecanismo que integra componentes de gerência automática de índices em bases de dados relacionais. Nós discutimos as heurísticas consideradas e também os resultados experimentais observados para uma implementação utilizando PostgreSQL.*

### 1. Introduction

The database tuning task consists of fine manipulations aiming at obtaining better performance of DBMS-based applications by means of an efficient use of the available computational resources. It is one of the main maintenance tasks performed by a database administrator (DBA). Tuning considers hardware configuration, physical design and query specifications, as well as commercial DBMSs parameters.

A good approach for the tuning process is to understand the functioning of the entire system, but only carry through improvements in specific points each time [Shasha and Bonnet 2003]. Some factors have made the tuning process more complex, as in the case of 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, the tuning activity becomes even more important and, at the same time, more expensive, as highly specialized professionals are needed [Chaudhuri and Weikum 2000].

In our work we present an approach to completely automate the index self-tuning process. This means no human intervention at all. We use a tuning component integrated with the DBMS optimizer to choose good indices and create them when needed. The component also detects and drops bad indices, if any are evaluated as so. Our main goals are, on one hand, to enable the experienced DBA to focus on more complex and unattended situations and, on the other hand, to offer a completely automatic solution when no DBA or database experts are present.

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, and static, workloads. Indeed, it is up to the DBA to decide upon choosing the right workload and parameters. Moreover, the DBA needs also to determine when to execute index creation (or destruction) commands. Our mechanism enables a complete automation of this whole process for dynamic workloads.

We have used PostgreSQL [Pos ] DBMS, an open source and full-fledged relational DBMS, in order to validate our ideas. We have coded a new version for PostgreSQL that includes hypothetical indices, that help the DBA with what-if simulations. It also enables our component to detect when new indices must be built or excluded. We observe also that there are no automatic tuning advisors (or wizards) present in PostgreSQL.

We have tried many different heuristics to support the DBMS decision upon the automatic creation of an index. We present here the *Benefits Heuristic* and give some practical results with a transactional (TPC-C) workload.

The rest of the paper is organized as follows: in the next Section, we present and discuss related works within a given classification on self-tuning database systems. A self-tuning engine that enables autonomic index management is presented in Section 3. Then, in Section 4, we present implementation and architectural issues, besides practical results obtained. Finally, Section 5 lists our contributions, future and ongoing work.

## **2. Autonomic and Self-tuning Databases**

Self-tuning techniques may fall into two broad groups: *local* self-tuning and *global* self-tuning. The latter refers to works that try to establish general principles to the construction of next generation data managers that should, by design, adapt automatically to their environments. The underlying idea is to keep system components *balanced* in such a way that the overall performance is better than the situation where each component would have to decide upon tuning issues separately (e.g. [Dias et al. 2005, Lifschitz and Milanés 2005]).

Relatively few works have been done to evaluate how different components that compete for resources affect each other's performance. Practical tuning experience indicates that understanding such correlations can be quite challenging [Shasha 1996]. As already mentioned in [Chaudhuri and Weikum 2000], these systems need to deal with the complexity of integrating self-tuning mechanisms with the system's code base.

Our main focus here is, rather, with local self-tuning approaches. These seek for

solutions to specific tuning problems that state-of-the-art DBMSs encounter. We may cite some proposals, from academic works and also commercial products, that (semi) automate the tuning task for the following issues:

- Database design [Agrawal et al. 2004, Agrawal et al. 2000, Chaudhuri et al. 2004, Ganti et al. 2000, Lohman et al. 2000, Zilio et al. 2004]: determination of the physical design that maximizes throughput for a given database schema and workload. Much work has been invested in indexes and materialized views selection.
- Data placement and partitioning [Agrawal et al. 2003, Lee et al. 2000, Papadomanolakis and Ailamaki 2004, Scheuermann et al. 1998]: allocation and dynamic reallocation of relation fragments in order to obtain the best possible load balancing among elements even when workload characteristics change.
- Load control [Rahm 1997, Weikum et al. 1994]: definition, in a transaction processing system, about how to dynamically tune the multiprogramming level (MPL) so that data contention thrashing can be avoided.
- Page replacement [Chen and Roussopoulos 1993, Faloutsos et al. 1995, Johnson and Shasha 1994, O'Neil et al. 1993]: decision upon keeping in the system's shared buffer the most popular pages of the database, even when workload patterns change over time.
- Buffer and memory tuning [Brown et al. 1994, Weikum et al. 1999, Xu et al. 2002]: evaluation on how to partition the memory available to the DBMS among different transaction classes or objects.
- statistics management [Aboulnaga and Chaudhuri 1999, Chen and Roussopoulos 1994, Stillger et al. 2001]: the goals are to automatically decide which statistical information must be collected to the DBMS and to automate statistics gathering.

Most existing proposals focus on local self-tuning, as it is very difficult to characterize the performance behavior of a DBMS as a whole [Weikum et al. 2002]. The strategy followed by previous works give more attention to tuning issues related to DBMS individual components. This type of work allows researchers to gain insight into the factors that affect performance of specific DBMS modules and serves as a basis to the investigation (and validation) of the interrelationships among components.

### **Index Selection**

There are many approaches to automatic index selection described in the literature. Much of the early work has focused on constructing tools with detailed rules that encode the knowledge of good database designs.

In [Finkelstein et al. 1988] the proposed index selection tool makes use of the optimizer's cost estimates to compare the gains of alternative hypothetical designs. This is a key observation because it avoids asynchrony between the cost models used by the tool and the system. The work of [Frank et al. 1992] proposes that the optimizer's interface be extended to allow the experimentation of alternative hypothetical index sets. Hypothetical indices exist only in the database schema and are not actually materialized.

In [Agrawal et al. 2000, Chaudhuri and Narasayya 1998a, Chaudhuri and Narasayya 1998b] tools for indices suggestion implemented in Microsoft's SQL Server are discussed, as part of the AutoAdmin project. Particularly in

[Agrawal et al. 2000], materialized views suggestion is also considered. The objective of the index suggestion tools is to generate an index set for one determined input workload, obtained by the DBA. The workload is broken into single statement inputs and candidate indices for each statement are generated. As not all the possible indices evaluated exist in the database, a separate module enables the consideration of hypothetical indices. The candidate indices found for a statement are the hypothetical indices that would bring the largest benefit to its execution in case they were materialized. Such candidate indices are then arranged into configurations and costed by the query optimizer. A greedy algorithm extends the number of statements and index configurations considered until a best index configuration is determined to the workload as a whole.

The work of [Lohman et al. 2000, Zilio et al. 2004] suggests that the index selection heuristic be tightly integrated with the optimizer. In [Zilio et al. 2004] the index selection tool is augmented with materialized views suggestion. The optimizer itself is extended with an index suggestion mode. In this mode, before the optimization of a given query, hypothetical indices for all relevant column usages are generated. As the number of possible multi-column indices for a query can be large, an heuristic, called “Smart column Enumeration for Index Scans” (SAEFIS), is proposed to limit the enumeration. Then, optimization proceeds normally. At the end, the indices picked by the optimizer are recommended for the query. The single query recommendations serve as input to an index selection heuristic that tries to find the best index configuration for a given workload. In their strategy, the benefit assigned for each index is the entire benefit of the winning set of indices found for the query. Then, the problem is viewed as the 0-1 knapsack problem and the classic greedy heuristic is used to find a solution. In our work, we have used the SAEFIS heuristic for the hypothetical index enumeration step (see Section 3).

In [Chaudhuri et al. 2004] the authors study the complexity of the index selection problem and prove it to be both NP-hard and hard to approximate. Reviewing previous approaches, a new heuristic is proposed for assigning benefits to individual indices using a linear programming strategy. This strategy is more refined than the one presented at [Lohman et al. 2000]. The problem of selecting indices for the workload is also viewed as the 0-1 knapsack problem and the classic greedy heuristic is used to find a solution.

All of these previous proposals do not address the automation of the complete cycle of workload detection, index selection and actual data structure creation or destruction. It is important to stress that previous heuristics presented in the literature do not have an on-line operation. Therefore, the presence of a DBA is mandatory during the index tuning process, in order to characterize the system’s workload. Moreover, the final decision upon index management (creation or dropping) also requires human intervention.

Our work here studies a possible solution to these problems through the use of heuristics embedded in an autonomic system component. Our heuristic decides on-line which indices should be managed to speed workload execution. This approach was also adopted in [Sattler et al. 2004], though with a distinct cost model, specially with respect to the cumulative benefit considered for index creation.

### **3. Autonomic Index Management**

In our approach, all statements submitted to the database are considered equally important and no individual statement is bound to execution time limits. Our goal is to maximize

the throughput, making the best possible use of the available computational resources through the creation of an adequate index design. We do accommodate seasonal changes in workload patterns as long as there is enough time between changes for our component to recognize new workload characteristics and modify the index design.

Our architecture uses a tuning component based on software agents that monitors the system and makes decisions autonomically. Software engineering techniques needed to build such a component are further discussed in [Costa et al. 2005, Salles 2004]. The self-tuning component interacts with DBMS components through a generic self-tuning process with the following stages:

- **Information Retrieval:** the self-tuning component obtains measurements and information from DBMS components.
- **Situation Evaluation:** from the information obtained, the self-tuning component updates data structures that will guide his tuning decisions.
- **Possible Alterations Enumeration:** the self-tuning component uses heuristics to enumerate a set of alternative adjustments that can bring improved system performance. During this enumeration, the component can simulate hypothetical scenarios using mechanisms previously built in the DBMS components.
- **Alterations Accomplishment:** the self-tuning component applies the chosen modifications to system components.

The self-tuning process presented here is based on a *feedback control loop*. In the loop, tuning decisions are progressively refined by the evaluation of new measurements extracted from system components. The use of a feedback loop is one of the main characteristics of self-tuning database architectures proposed so far [Weikum et al. 2002].

### **Choice of Index Creations and Destructions**

Our goal is to eliminate human intervention from the index tuning process. To create good index designs, the autonomic index tuning component should have appropriate index choice procedures. In most previous works, the task of index selection is subdivided in two phases [Chaudhuri et al. 2004]:

- **Candidate Generation:** identifies candidate indices for the workload, typically chosen per query after an enumeration of hypothetical indices.
- **Configuration Selection:** chooses which indices among the candidates should be created or removed. The heuristic should obtain an index design that balances query execution speed and update index maintenance cost.

For Candidate Generation, we consider here an adaptation of the enumeration procedure described in [Lohman et al. 2000]. We use the SAEFIS heuristic for multi-column indices enumeration and we also list all one-column indices relevant to the query.

Configuration Selection heuristics in the literature are usually executed off-line, apart from the DBMS, and with an already supplied, and static, workload. In order to follow this paradigm, our autonomic component would have to accumulate the system's workload and evaluate index configurations at fixed time intervals. The derivation of what interval is appropriate for workload characterization is not easy. Workloads with different seasonality and patterns would have to be observed for different, and perhaps varying, intervals. Introducing such a parameter would create another tuning knob for the DBA.

One must also remember that our autonomic component runs concurrently with other DBMS tasks. This implies that the Configuration Selection heuristic must not be computationally intensive. We also explicitly compute the accumulated benefit of actual indices present in the database and drop indices when they become harmful to system performance. Furthermore, due to our built-in mechanism, we may capture all DBMS statements, including stored procedures, being able to make automatic fine tuning decisions.

### **Benefits Heuristics**

We need to first introduce some of the factors that are taken into account:

1.  $C_A$ : the cost, generated by the optimizer, corresponding to the best query execution plan over the actual indices configuration for a given query.
2.  $C_H$ : the cost, generated by the optimizer, for the best execution plan considering both actual and hypothetical indices for a given query.
3.  $C_N$ : the cost, also generated by the optimizer, considering a physical configuration with no indices (neither actual, nor hypothetical) for a given query.
4.  $C_U$ : the estimated index maintenance cost when an update operation happens. Usually this cost is not determined by the optimizer. Therefore, we have estimated it in our implementation according to the cost model of the DBMS being used.
5.  $B_I$ : the benefit that index  $I$  brings to the statement being evaluated. This benefit is determined by our heuristic, distinguishing actual and hypothetical indices.
6.  $AccB_I$ : the accumulated benefit that index  $I$  brings to all statements already evaluated. Again, the heuristic will update the accumulated benefit considering if the index is hypothetical or actual.
7.  $CC_I$ : the estimated creation cost for index  $I$ . Usually optimizers do not calculate this factor. Therefore, we have estimated it in our implementation.

When a new SQL operation is submitted to the DBMS, the optimizer generates the best query plan considering the actual indices present in the database. On the occurrence of this event, the index tuning component is notified that a new statement is available and invokes the Candidate Generation heuristic. This heuristic interacts with the optimizer to obtain the best query plan considering both actual and hypothetical indices. The hypothetical indices selected for this plan are taken as the candidate indices for the statement. The component also performs an additional interaction with the optimizer to obtain the cost of the best query plan that uses no indices. After these steps, we will have the values of the factors  $C_A$ ,  $C_H$  and  $C_N$  for the current statement. The *Benefits Heuristic* can then be invoked. It treats queries and updates distinctly, as will be described in the following.

### **Query Evaluation**

If the statement submitted is a query, we apply the procedure detailed in Figure 1. For each candidate index, we calculate its benefit and update its accumulated benefit. The benefit is obtained as the difference between the cost of the best query plan over the actual indices configuration and the cost of the best query plan over the configuration that

```

For each candidate index  $I$  for the query do
     $B_I = C_A - C_H$ ;
     $AccB_I = AccB_I + B_I$ ;
    If  $AccB_I > CC_I$  then
        Create index  $I$ 
         $AccB_I = 0$ ;
    End If;
End For;

For each actual index  $I$  used by the query do
     $B_I = C_N - C_A$ ;
     $AccB_I = AccB_I + B_I$ ;
End For;

```

Figure 1. Query evaluation and index creation

includes actual and candidate indices. As the costs are obtained for the *best* query plans, the benefit calculated is always nonnegative.

If various candidate indices are used in the query plan, we will attribute the same benefit to all of them. As in this case the benefit brought by each index is not independent of the other indices used, we are incurring in double-counting. The benefit calculation procedure follows the one proposed in [Lohman et al. 2000]. More sophisticated schemes can be devised, such as the one in [Chaudhuri et al. 2004].

After calculating the benefit for an index, we will create it if and only if its accumulated benefit surpasses its estimated creation cost. Our objective is to create indices that are used enough times to compensate their creation cost.

For each actual index used to process the query, the heuristic calculates the benefit as the difference between the cost of the best query plan over a configuration with no indices and the cost of the best query plan over the actual indices configuration. Note that, once again, the benefit will be nonnegative. As actual indices are already materialized, we just update their benefits to reflect their usage.

One of the factors we use in our procedure, namely index creation cost, is not calculated by typical query optimizers. In our approach, we have devised a formula for index creation cost in accordance with the cost model of the DBMS we have used (PostgreSQL):

$$CC_I = 2P + cRl \quad R$$

In the formula,  $P$  represents the number of pages in the underlying table,  $R$  is the number of tuples and  $c$  is a factor that measures what percentage of the cost to obtain a page from disk should be applied to obtain the cost to process a tuple in main memory. The formula calculates the number of pages touched to scan the table's data, sort it and then construct the index in one pass.

Considering index creation costs explicitly differentiates our approach from others that make index suggestions and leave the actual materialization decision to the DBA.

## Update Evaluation

If the statement submitted to the DBMS is an update (insert/update/delete), we apply the procedure detailed in Figure 2. The procedure begins with an evaluation identical to the one made for queries. The first step of an update is to bring data to memory. Therefore, the optimizer must find a query plan that will retrieve from disk all the information relevant to the update. Once data is placed in memory, it can be modified and, eventually, written back to persistent storage. In most systems, modifications are logged sometime before transaction commit and the actual writing of data is delayed to checkpoints. Query optimizers typically estimate the cost for the query plan of the update, but they do not offer estimates for the cost involved in writing the data and updating index structures.

```

Execute "Query Evaluation Pseudocode"

For each index I affected by the update do
    AccBI = AccBI - CU;
    If ( I is actual ) and
        ( AccBI < 0 ) and
        ( |AccBI| > CCI ) then
        AccBI = 0 ;
        Drop index I
    End If ;
End For ;

```

**Figure 2. Update evaluation and index dropping**

After the procedure for query evaluation is applied, we compute index maintenance costs and debit these costs from the accumulated benefit of all candidate and actual indices affected by the update. To estimate index maintenance costs, we have introduced the factor  $C_U$ . In our implementation, we have supposed that the cost to update any of the indices affected by the statement is identical. This could easily be extended to make the factor  $C_U$  a function of the index affected. To calculate  $C_U$ , we have used a cost formula that estimates the number of pages written by the update:

$$C_U = 2 \left\lceil \frac{r}{R} \right\rceil P + cr$$

Again, this formula was created in accordance with the PostgreSQL's cost model. The  $r$  factor represents the number of tuples updated by the command and  $R$ ,  $P$  and  $c$  are analogous to those discussed before for index creation costs. The first term of the formula computes the cost necessary to update an amount of pages proportional to the fraction of table records that were updated, multiplied by the table size (in pages). We therefore make a simple, yet effective, assumption where updated records will be uniformly distributed among the pages of the table. The first term counts twice since it is necessary to read and write each page. The second term of the formula computes the cost to process each of the records updated in main memory.

Once we have updated the accumulated benefit of an actual index, we verify if keeping it materialized in the database is still advantageous. When the benefit brought by the actual index is negative and, in absolute terms, greater than the cost to create the index, the index is dropped.

When compared to previous heuristics proposed in the literature for Configuration Selection, the *Benefits Heuristic* presents an on-line behavior that enables it to make index management decisions without forcing the DBA to define what is a relevant workload to the system. It is designed to be used by an autonomic component embedded in the DBMS that follows a self-tuning process based on a feedback control loop. The index design is continuously refined after each new statement submitted to the system.

#### **4. Implementation**

We have implemented our component within PostgreSQL 7.4, beta 3 [Pos ], running on Red Hat Linux 9 kernel 2.4.20-30.9. We have used a 512MB RAM Pentium 4 2Ghz server, with all codings in gcc and g++ 3.2.2. We have chosen PostgreSQL, not only because it is an open source DBMS, but also because it is highly modularized and well documented. Furthermore, it is a quite robust DBMS widely used that can actually reflect the expected behavior with practical results.

We have followed an index selection approach based on optimizer estimates, the same way as considered in other related works (e.g. [Finkelstein et al. 1988, Lohman et al. 2000]). In these proposals, the database server is extended to allow the simulation of hypothetical indices. We have coded similar server extensions for PostgreSQL, but detailing these is beyond the scope of this paper. Our prototype, and the server extensions, are freely available at [PUCDB ]. Detailed results of our autonomic index management research work are also presented in [Salles 2004, Morelli 2006].

We consider for our experiments here the Database Test 2 (DBT-2) toolkit provided by the Open Source Development Labs (OSDL) [OSDL-DBT2 ]. Its origin is the Transaction Processing Performance Council's TPC-C benchmark [TPC-C ]. This DBT-2 toolkit simulates a workload that represents a wholesale parts supplier operating out of a number of warehouses and their associated sales districts. The toolkit, as usual benchmarks, comes with a set of suggested (human created) indices that improve the evaluation of the given workload. Some indices are created due to primary key creation and others come up aiming at increasing the query processing performance. Due to space limitations, the reader may refer to [OSDL-DBT2 , Salles 2004] for further details.

In order to evaluate the contribution brought by our index tuning component, we have established three relevant configurations for running our experiments:

1. A database with no indices at all and our component turned off, which we call here IOC0 (both indices and component zero). If we do not consider small variations due to transaction submissions frequency, we expect that this would lead to the smallest throughput, once all queries would be executed through full scans on related tables. We remove all indices, including those that refer to primary keys.
2. No indices at the database and component turned on (IOC1 - indices zero, component one). We expect our component to find and materialize, automatically, those indices that may be beneficial to run the workload in consideration.

- Component turned off and the database with the indices suggested by the DBT-2 toolkit (I1C0 - indices one, component zero). Here we consider that the throughput would be the greatest one since we are using those indices that DBT-2 toolkit implementers have chosen and the system does not run the tuning component.

There are, still, two variables that may affect the experiments' throughput. First, the number of warehouses used to create the database may be viewed as a scale factor and, therefore, the greater the number of warehouses is, also greater will be tables' sizes. The system load will also be bigger since more terminals simulating users are created.

The second variable to be considered is how long the experiment takes. Specially when the component is active, time may directly influence the observed throughput. Indeed, there are 2 steps that the component goes through while analyzing and adapting indices to the database. The first step is the learning one, when the best indices for submitted commands are detected and materialized. The second step is the steady state, where the component only verifies whether or not the existing indices are still adequate, no indices are materialized anymore. The throughput in the steady state phase is considerably greater than during learning periods, leading to a greater average throughput.

### Throughput Results

With respect to our experiments, we have tried out a few configurations, basically changing warehouse quantities and the total execution time. In [Salles 2004] we give the complete test results not shown here due to space limitations.



Figure 3. Throughput in a 90 minute test

Figure 3 shows the results obtained for a 90 minute time test. It is worth noting that the throughput for the I0C0 configuration decreases when the number of warehouses increase. This is due to full scan query processing, leading to worse performances when the database size increase. When there are more terminals submitting queries, data contention becomes a problem straightforward.

As expected, an intermediate throughput appears for the I0C1 configuration. The tuning component eventually reaches the end of the learning step and the workload remains active for a while with adequate indices. Therefore, the throughput gets closer to the I1C0 configuration throughput as the total execution time increases.

## Index Design Quality

Besides the number of indices created by our component, it is also worth to evaluate the quality of indices. We could observe that the component has chosen about the same indices at each experiment, even when we vary the database size. There is a set of 10 indices repeatedly created, as given in Table 1.

Goal	Database Table	Toolkit Suggested Index	Component Created Index
Primary key enforcement	customer	c_w_id, c_d_id, c_id	c_id, c_d_id, c_w_id
	district	d_w_id, d_id	d_w_id
	item	i_id	i_id
	new_order	no_w_id, no_d_id, no_o_id	no_o_id, no_d_id, no_w_id
	order_line	ol_w_id, ol_d_id, ol_o_id, ol_number	ol_o_id, ol_d_id, ol_w_id
	orders	o_w_id, o_d_id, o_id	o_id, o_d_id, o_w_id
	stock	s_w_id, s_i_id, s_quantity	s_i_id, s_w_id
warehouse	w_id	NO corresponding index	
Performance	customer	c_w_id, c_d_id, c_last, c_first, c_id	c_d_id, c_w_id, c_last, c_first
	orders	o_w_id, o_d_id, o_c_id	o_d_id, o_w_id, o_c_id, o_id
	new_order	NO corresponding index	no_w_id

**Table 1. Indices automatically created and those suggested by the toolkit**

Table 1 shows database tables for which indices were created along with the columns indexed in each table. In the first line of the Table, we show an index for primary key enforcement on the *customer* database table. The index is on the warehouse id (*c\_w\_id*), the sales district id (*c\_d\_id*), and the customer id (*c\_id*). We show the index as it was created by the toolkit and as it was detected by the tuning component. Note that the component has no knowledge about semantic constraints on the table, such as primary and foreign keys. It bases its decisions on which indices are adequate to lower the workload's processing cost.

On the tenth line of the table, we present an index on the *orders* table that was created for performance enhancement. The index suggested by the toolkit is on the warehouse id (*o\_w\_id*), the sales district id (*o\_d\_id*), and the customer id (*o\_c\_id*). The component suggests an index with the same columns plus the order id (*o\_id*).

Indices presented in the seventh and eleventh lines of the Table were created by the component only when the database was populated with more than one warehouse; all other indices were created by the component in all database configurations.

It should be noted that most indices suggested by the component and by the toolkit are very similar, the only difference being the column order. The reason is that, for each column group detected by the component, the column order is established according to the corresponding positions at the base table. For those indices that are created from the DBT-2 toolkit suggestion, columns are placed with respect to the expected selectivity order. However, we must observe that this column ordering did not change the results obtained for the workload in consideration.

Besides those differences, we can also note from Table 1 that there is an index among those suggested by the toolkit that was not created by our component. Indeed, as

our tuning component cares only about indices that improve the workload performance, and the *warehouse* table is always small - in our experiments it contained at most 4 tuples -, from the optimizer point of view there is no need to create an index for it. The corresponding toolkit index is, as shown, a primary key constraint enforcement index.

A more interesting situation corresponds to component's decision upon creating an index for table *new\_order* on column *no\_w\_id*. This index allows us to scan the table ordered by warehouse id. There exists a given workload query for which the optimizer estimates the creation of such an index would be interesting when the table size increases. Thus, even with a low query frequency with respect to the complete workload, the benefits expected justify the index creation.

In order to compare the quality of indices, either suggested by the DBT-2 toolkit, or suggested (and created) by our component, we have decided to use the optimizer to estimate the execution cost of each workload command in both configurations. Next, we have studied related costs considering expected frequencies, obtaining an weighted cost for each command. The results are given in Figure 4, with the sum of all weighted costs obtained for the whole workload and a 4-warehouse database.

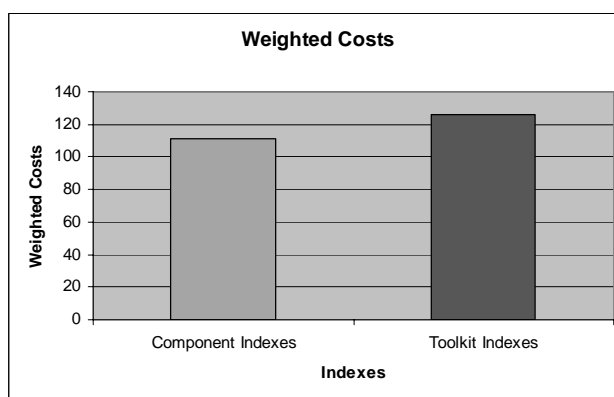


Figure 4. Weighted workload costs, as estimated by the query optimizer

We have obtained a weighted cost for indices used by the component 11,64% lower than the cost obtained for those indices suggested by the toolkit. The extra index created by the component for table *new\_order* on column *no\_w\_id* has a fundamental role: without this index, we would have obtained a weighted cost 4,79% bigger instead.

## 5. Conclusions

In this paper we have presented a self-tuning approach that allows autonomic index management for dynamic workloads. One main contribution is to have a built-in component that captures statements issued by the DBMS, including stored procedures. As a matter of fact, everything that goes through the optimizer is considered and, consequently, good tuning decisions can be made. The heuristics embedded in the automatic tuning component also distinguishes itself from previous approaches by evaluating on-line which indices should be created and destroyed. The heuristic explicitly considers the balance of query speedup and index maintenance costs.

We have conducted experiments with a transactional workload to check the viability of our approach. Our results indicate that the component is capable of altering

the database's index design dynamically, though some time is needed for it to recognize the workload's characteristics and effectively materialize indices. The component consistently achieved an index design that favors transactional throughput. It should be noted another distinction here from previous works is that we do not consider static workloads for index tuning as most tuning wizards usually do.

We are currently studying the extension of this work to deal with more complex workloads, such as those involving *ad hoc* queries and decision support applications, and index updates [Morelli 2006]. We also plan to bring enhancements to the *Benefits Heuristic* by looking at alternative ways to attribute benefits to individual candidate indices and by including storage space limitation in the on-line decision procedure.

## References

- Aboulnaga, A. and Chaudhuri, S. (1999). Self-tuning histograms: building histograms without looking at data. In *Procs ACM SIGMOD Intl. Conf. on Management of Data*, pages 181–192.
- Agrawal, R., Chaudhuri, S., Das, A., and Narasayya, V. (2003). Automating layout of relational databases. In *Procs IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 607–618.
- Agrawal, S., Chaudhuri, S., Kollar, L., Marathe, A., Narasayya, V., and Syamala, M. (2004). Database tuning advisor for microsoft sql server 2005. In *Procs Intl. Conf. Very Large Databases (VLDB)*, pages 1110–1121.
- Agrawal, S., Chaudhuri, S., and Narasayya, V. (2000). Automated selection of materialized views and indexes for sql databases. In *Procs Intl. Conf. Very Large Databases (VLDB)*, pages 496–505.
- Brown, K. P., Mehta, M., Carey, M. J., and Livny, M. (1994). Towards automated performance tuning for complex workloads. In *Procs Intl. Conf. Very Large Databases (VLDB)*, pages 72–84.
- Chaudhuri, S., Datar, M., and Narasayya, V. (2004). Index selection for databases: A hardness study and a principled heuristic solution. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1313–1323.
- Chaudhuri, S. and Narasayya, V. (1998a). Autoadmin “what-if” index analysis utility. In *Procs ACM SIGMOD Intl. Conf. on Management of Data*, pages 367–377.
- Chaudhuri, S. and Narasayya, V. (1998b). Microsoft index tuning wizard for sql server 7.0. In *Procs ACM SIGMOD Intl. Conf. on Management of Data*, pages 553–554.
- Chaudhuri, S. and Weikum, G. (2000). Rethinking database system architecture: Towards a self-tuning risc-style database system. In *Procs Intl. Conf. Very Large Databases (VLDB)*, pages 1–10.
- Chen, C. M. and Roussopoulos, N. (1993). Adaptive database buffer allocation using query feedback. In *Procs 19th Intl. Conf. Very Large Databases (VLDB)*, pages 342–353.
- Chen, C. M. and Roussopoulos, N. (1994). Adaptive selectivity estimation using query feedback. In *Procs ACM SIGMOD Intl. Conf. on Management of Data*, pages 161–172.

- Costa, R., Lifschitz, S., de Noronha, M., and Salles, M. (2005). Implementation of an agent architecture for automated index tuning. In *Procs IEEE Intl. Workshop on Self-Managing Database Systems (SMDB)*.
- Dias, K., Ramacher, M., Shaft, U., Venkataramani, V., and Wood, G. (2005). Automatic performance diagnosis and tuning in oracle. In *Procs Biennial Conf. on Innovative Data Systems Research (CIDR)*, pages 84–94.
- Faloutsos, C., Ng, R. T., and Sellis, T. K. (1995). Flexible and adaptable buffer management techniques for database management systems. *IEEE Transactions on Computers*, 44(4):546–560.
- Finkelstein, S., Schkolnick, M., and Tiberio, P. (1988). Physical database design for relational databases. *ACM Transactions on Database Systems*, 13(1):91–128.
- Frank, M., Omiecinski, E., and Navathe, S. (1992). Adaptive and automated index selection in rdbms. In *Procs Intl. Conf. on Extending Database Technology (EDBT)*, pages 277–292.
- Ganti, V., Lee, M., and Ramakrishnan, R. (2000). Icicles: Self-tuning samples for approximate query answering. In *Procs Intl. Conf. Very Large Databases (VLDB)*, pages 176–187.
- Johnson, T. and Shasha, D. (1994). 2q: A low overhead high performance buffer management replacement algorithm. In *Procs Intl. Conf. Very Large Databases (VLDB)*, pages 439–450.
- Lee, M. L., Kitsuregawa, M., Ooi, B. C., Tan, K.-L., and Mondal, A. (2000). Towards self-tuning data placement in parallel database systems. In *Procs ACM SIGMOD Intl. Conf. on Management of Data*, pages 225–236.
- Lifschitz, S. and Milanés, A. (2005). Design and implementation of a global self-tuning architecture. In *Procs Brazilian Symp. on Databases (SBBD)*, pages 70–84.
- Lohman, G., Valentin, G., Zilio, D., Zuliani, M., and Skelley, A. (2000). Db2 advisor: An optimizer smart enough to recommend its own indexes. In *Procs IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 101–110.
- Morelli, E. T. (2006). Updating database indices automatically (in portuguese). Master’s thesis, Departamento de Informatica PUC-Rio.
- O’Neil, E., O’Neil, P., and Weikum, G. (1993). The lru-k page replacement algorithm for database disk buffering. In *Procs ACM SIGMOD Intl. Conf. on Management of Data*, pages 297–306.
- OSDL-DBT2. Open source development labs database test 2. [http://www.osdl.org/lab\\_activities/kernel\\_testing/osdl\\_database\\_test\\_suite/osdl\\_dbt-2/](http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_suite/osdl_dbt-2/).
- Papadomanolakis, S. and Ailamaki, A. (2004). Autopart: Automating schema design for large scientific databases using data partitioning. In *Procs IEEE Intl. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 383–392.
- Pos. Postgresql. <http://www.postgresql.org>.
- PUCDB. Puc-rio database self-tuning group. <http://www.inf.puc-rio.br/postgresql>.

- Rahm, E. (1997). Goal-oriented performance control for transaction processing. In *Procs ITG/GI MMB Conference (Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen)*, pages 285–300.
- Salles, M. A. V. (2004). Autonomic index creation in databases (in portuguese). Master's thesis, Departamento de Informatica PUC-Rio.
- Sattler, K.-U., Geist, I., and Schallehn, E. (2004). Autonomous query-driven index tuning. In *Procs Intl. Database Engineering and Applications Symp. (IDEAS)*, pages 439–448.
- Scheuermann, P., Weikum, G., and Zaback, P. (1998). Data partitioning and load balancing in parallel disk systems. *The VLDB Journal*, 7:48–66.
- Shasha, D. (1996). Tuning databases for high performance. *ACM Computing Surveys*, 28(1):113–115.
- Shasha, D. and Bonnet, P. (2003). *Database Tuning: Principles, Experiments and Troubleshooting Techniques*. Morgan Kaufmann.
- Stillger, M., Lohman, G., Markl, V., and Kandil, M. (2001). Leo - db2's learning optimizer. In *Procs Intl. Conf. Very Large Databases (VLDB)*, pages 19–28.
- TPC-C. Tpc benchmark c: Standard specification - revision 5.2. [http://www.tpc.org/tpcc/spec/tpcc\\_current.pdf](http://www.tpc.org/tpcc/spec/tpcc_current.pdf).
- Weikum, G., Hasse, C., Mönkeberg, A., and Zaback, P. (1994). The comfort automatic tuning project. *Information Systems*, 19(5):381–423.
- Weikum, G., König, A. C., Kraiss, A., and Sinnwell, M. (1999). Towards self-tuning memory management for data servers. *IEEE Data Engineering Bulletin, Special Issue on Self-Tuning Databases and Application Tuning*, 22(2):3–11.
- Weikum, G., Monkeberg, A., Hasse, C., and Zaback, P. (2002). Self-tuning database technology and information services: from wishful thinking to viable engineering. In *Procs Intl. Conf. Very Large Databases (VLDB)*, pages 20–31.
- Xu, X., Martin, P., and Powley, W. (2002). Configuring buffer pools in db2 udb. In *Procs IBM Centers for Advanced Studies Conference (CASCON)*. CDROM/On-line Proceedings, 12 pp.
- Zilio, D., Zuzarte, C., Lightstone, S., Ma, W., Lohman, G., Cochrane, R., Pirahesh, H., Colby, L., Gryz, J., Alton, E., Liang, D., and Valentin, G. (2004). Recommending materialized views and indexes with ibm db2 design advisor. In *Procs IEEE Intl. Conf. Autonomic Computing (ICAC)*, pages 180–188.