Goal-Driven Autonomous Database Tuning Supported by a System Model

Karsten Schmidt
Databases and Information Systems Group
Department of Computer Science
University of Kaiserslautern
D-67653 Kaiserslautern, Germany
kschmidt@cs.uni-kl.de

ABSTRACT

Fast growing volumes of data, typically for today's database applications, require efficient and low-cost management to guarantee reliable processing and high throughputs. Although administrative costs for skilled DBAs are noticeable, it is often done manually due to the lack of available online tuning utilities. Typical *operating goals* for a database system comprise throughput, resource utilization, memory footprint, and due to recent efforts in saving energy also the green factor of a system. However, such high-level goals conceal the entire system complexity but help to address the essential cost factors.

With the help of an extendable system model, describing fix and variable input parameters, state, runtime behavior, and workload correlations of parameters and system components are disclosed. Based on that system model, goal-specific analyses should be possible to tune the system online or at least to recommend actions needed to fulfill the goals.

1. INTRODUCTION

Data management today goes beyond the scope of pure databases. Nevertheless, their good reputation for reliability, robustness, scalability, and functionality advocates their usage—often hidden to the front-end user—in numerous applications. Large volumes of data being processed call for the most efficient setup of the database management system (DBMS). This setup is hardly static and needs adaptation to changing runtime requirements, such as changes in data volumes, workload, system load, hardware and processing resources, and operational goals.

Since changing requirements occur at any time and for an unknown period of time, predicting them may fail or administrative support may not be available or not acting fast enough. That is why monitoring tools emerged to instantly report system state and potentially problematic behavior. Moreover, event-driven tuning tools, for instance to reconfigure memory pools, to trigger maintenance tasks, or to send reports are developed and established. However, there is a lack of comprehensive correlation analysis and understanding. For instance, increasing the buffer pool for a single container file may benefit the active reader/writer transaction for that container but penalize the efficiency of concurrency because the lock table had to be downsized. Another idea is to trade CPU costs for memory costs or for IO costs, simply by adjusting the chosen evaluation algorithm, however, this may result in an increased energy consumption. Thus, not only reacting to changing requirements is necessary but also taking care of the rest of the system (i.e., global optimization vs. local optimization).

An important feature for developers, users, or system architects of database-enabled applications is the visualization of costs in terms of throughput, bottlenecks, hardware underload, energy consumption, and their origin. Because they do not care about database internals, they should be confronted with a simple high-level tuning interface to set the operational goal and to monitor its behavior (i.e., system reaction to achieve the goal specifications).

Eventually, such a comprehensive system-model-based tool allows for so-called "what-if" analyses [6] and for, e.g., cost-effective, throughput-effective, or effective user-defined processing of the database as a "black box".

1.1 Research Issues

The main objective of this work is to develop a system model serving as platform for monitoring a database system, correlating measured metrics, and deducing measurements to improve performance in terms of the operational goal autonomously. The following issues need to be addressed for that work:

1. Specify DB monitoring points indicating state and trends for distinct components.

2. Define a system model that integrates metrics and their correlations to map physical resource allocation to the modeled components by using the monitoring points. Known tuning knobs (configuration parameters) should be integrated into the model.

3. Develop algorithms based on heuristics, learning, or static weights to recognize and to improve the system behavior during runtime, and moreover, to estimate the effects of model changes.

Proceedings of the Third SIGMOD PhD Workshop on Innovative Database Research (IDAR 2009), June 28, 2009, Providence, RI, USA
Copyright 2009 ACM X-XXXXX-XX-X/XX/XX ...$5.00.

Table 1: Self-tuning areas and goal contribution capabilities

Technique	Online	Multiuser-aware	Components	Operation goals**
Design advisor [24]	−	−	workload, storage	response time
Workload models [13]	/	−	workload, scheduler	throughput
Self-tuning memory [19]	+	/	memory pools	throughut, resource usage
Buffer tuning [20]	+	+	buffer pools	throughput, memory usage
Statistic management [3, 18, 1]	+	/	query optimizer	response time
Index selection [5, 21]	−*	−	index configuration	response time
What if analysis [6]	−	−	physical design	throughput, response time

− not available + available / unknown but possible

*) online recommedation, but no online management (decision, creation, deletion)

**) none of the techniques directly addresses costs or energy consumption, but most of them can be extended for that

4. Provide a convenient interface (GUI) for the end-user to specify certain operational goal(s), to observe the DB performance, and to present recommendations for non-automatable tuning tasks (e.g., RAM, CPU, Disk changes).

All concepts will be developed and tested based on our own native XML database prototype—XTC [9]. It allows to tightly integrate all of the identified monitoring points and tuning knobs into a real and well-understood XML DBMS. Of course, we want to keep the model, the algorithms, and the user front-end as general as possible.

1.2 Paper Roadmap

The rest of the paper is organized as follows: Section 2 discusses related work in the area of autonomous DBMS tuning. In Section 3, we will present our *Framework* containing the monitoring component, system model, algorithm types, and user front-end. Finally, we conclude our work in Section 4 presenting our results achieved so far and giving an outlook for future work.

2. RELATED WORK

Self-tuning, especially for DB systems, is not a recent topic but still a hot one. All of the major DB vendors integrated certain self-tuning features to facilitate administrative tasks [4, 19, 22, 23, 24]. However, very often these features only address a very specific part of the system, e.g., the *physical design*, *query optimizer*, *workload analysis*, *index/MQT recommendations*, or *memory allocation*. To the best of our knowledge, none of these approaches has tried to consider the entire DB system by defining a single system model. Moreover, most of these approaches fail to optimize multi-user situations and they focus on a single system component. Another disappointing aspect is that lots of the available solutions should only be used offline due to their impact on a production system, which, in turn, would lead to bad reaction times. In the following, we will present available concepts for the categories mentioned above.

2.1 Physical Design Tuning

Physical design tuning is one of the most difficult optimizations tasks due to the wide spectrum of possible configurations. However, most of the approaches only address recommendations for indexes or materialized query tables [2, 7, 23, 24] based on prior inspection of the given workload. More sophisticated approaches such as [23] include partitioning and clustering aspects into the design recommendation. Nevertheless, dynamic workload changes are still problematic to capture and, therefore, preventing outliers from deteriorating the entire system performance should be addressed as a minimal solution [7]. Of course, workload analysis is beneficial to physical layout tuning (e.g., indexes, MQTs), but isolating physical design decisions from other (and even physical) aspects (e.g., RAM, CPU, disks, load, users) may lead to suboptimal configurations.

2.2 Query Optimizer

Tuning query engines is one of the earliest topics in database research. Most attention is paid to correctly estimate selectivities and to apply a suitable cost model for alternative access paths. Besides plan restructuring, statistics gathering is *the* tuning option. Existing approaches such as [1, 3] observe the number of UDI[1] operations or take query frequencies into account to renew statistics. One of the major problems is to recognize valuable statistic candidates because having appropriate statistics is crucial and maintaining statistics for all (or too many) data values is expensive and counter-productive. To improve the candidate selection, newer approaches such as [18] piggyback runtime statistics and query feedback during query execution to offer them to the query optimizer.

Because statistical accuracy and topicality are crucial for the query optimizer, statistics should be gathered continuously considering system load, workload, and available resources.

2.3 Workload Analysis

Giving hints for the DB configuration by analyzing the workload is a common tuning approach (e.g., [10, 13, 14]). However, most research focuses on detecting OLTP and DSS-styled workloads[2] to adjust system parameters during runtime. One aspect is to identify or to predict the type of queries dominating the current load [10, 13]. Although OLTP and DSS workloads may change periodically or appear concurrently, we believe that blended workloads are more frequent and, therefore, should be addressed more efficiently. Another aspect in [14] addresses the business importance level of queries by formulating SLOs (Service Level Objectives) and instructing the scheduler accordingly. Such business goals are also important for other (autonomous) DB components (e.g., index tuning, statistics gatherer, memory allocator) and may be also supported by load and resource

[1]In contrast to Queries, *U*pdate, *D*elete, and *I*nsert operations may influence statistics.

[2]*O*n*L*ine *T*ransactional *P*rocessing and *D*ecision *S*upport *S*ystem workloads have different characteristics.

monitoring.

2.4 Dynamic Memory Allocation

Finding an optimal memory distribution for key memory areas in a database is not only difficult but also quickly becomes outdated. The most promising area to tune is the buffer [20] where replacement algorithms and dynamic sizes are key concepts. In contrast, tuning all areas at the same time (e.g., buffer pools, hash join, sort heap, lock table size, log buffer, (query, package, result) caches) requires a sophisticated memory model such as *self tuning memory manager* (STMM) [19]. Although the STMM approach confines its tuning capabilities to a single resource—main memory—, it exemplifies how to address several aspects simultaneously (e.g., memory areas, thresholds, response time).

Table 1 summarizes a selection of distinct techniques and shows their characteristics which are important for our comprehensive system model approach and which operational goals they address. Note, mostly throughput gains are addressed whereas other important facts such as energy consumption or resource usage are omitted.

2.5 Self-tuning

The term self-tuning in the field of database management is used for loads of techniques, approaches, and design principles. We refer to the *monitoring-decision-action* principle where certain parts of the system configuration are changed automatically (i.e., on demand) by the system itself (see Figure 1). Today's systems are fairly complex and different configuration knobs often have dependencies with each other which makes tuning a difficult task at all. Besides, increased system performance (e.g., transactional throughput), cost reduction for administrative tasks and minimal reaction times are the main objectives.

In the following, we will give a typical example for DB tuning, namely index tuning, and define based on that example a formal description of *self-tuning.*

Index Tuning Example

For self-tuning index configurations the system accounts costs for index creation, maintenance, and space consumption to contrast them with the benefits during query processing using these indexes. For a given set of query statements $S = \{S_1, \ldots, S_m\}$, a set of index candidates $IC = \{IC_1, \ldots, IC_n\}$, query costs without indexes s_{noIdx}, query costs including indexes s_{Idx}, and index maintenance costs $mc(IC_i, Q_l)$, a cost-benefit formula is defined as follows:

Definition 1. *The benefit of a specific index configuration is the difference of query processing costs including indexes and without indexes less the index creation costs:*

$$benefit(S, IC) = \sum_{q \in S} (count_q \cdot (s_{noIdx} - s_{Idx}) - \sum_{i=1}^{n} mc(IC_i, s))$$

as long as space restrictions are observed:

$$\sum_{IC_i \in IC} size(IC_i) \leq MaxIndexSpace$$

A key issue is the separation of benefits to account each index candidate independently. For instance, a query may reduce the processing time by 20 % using two indexes and respectively 15 % using only one of them at a time. Because the cost-benefit calculation only sees combined benefit gains using both index candidates at a time, a benefit distribution becomes necessary.

Abstract Self-Tuning

Based on the index tuning example, a more generic equation for tuning assessment can be derived as follows.

Definition 2. *For a given workload $W = \{q_1, \ldots, q_n\}$, the operational goal G (e.g., throughput, runtime) has to be minimized*[3] *by adjusting all available resources and configuration parameters $P = \{p_1, \ldots, p_m\}$ and by observing all constraints $C = \{c_1, \ldots, c_i\}$. This leads to:*

$$G = \min_{0 \leq n, 0 \leq m, 0 \leq i} (W \times P \times C)$$

Now, geting back to the index benefit calculation, we have to take $cost(P)$ for reconfiguration as well as the system state for a point in time t into account too. This leads to the following (simplified) equation:

Definition 3. *The benefit for a potential new state can be estimated or calculated by reducing the gain with the costs for configuration changes and searching alternative configurations.*

$$\begin{aligned} benefit(t+1) = G(t+1) - G(t) - cost(P(t+1) - P(t)) \\ - cost(\min(W(t) \times P(t) \times C(t))) \end{aligned}$$

Given the state and $benefit$ for a specific point in time t the self-tuning system has to locally decide which measurements are profitable until $t+1$ is reached. However, tuning needs to integrate the benefit over time, at least for a suitably sized time-sliding window, to achieve overall gains w.r.t. the given operational goal.

2.6 Online vs Offline

Defining an optimal system configuration for a fixed workload and a nearly static database can be done by testing all possible configurations for that workload. Such an approach needs a lot of processing time and resources to properly select the most beneficial configuration and, therefore, can only be done offline on a non-production system. But for changing workloads (shifts or temporal hot spots) or changing data distribution, the offline approach is penalized by a weak reaction time to these changes. In contrast, an online approach is penalized by some overhead to monitor the workload, perform configuration changes, and process the cost-benefit calculation. Despite the overhead, we prefer an online approach to maximize self-tuning benefits while minimizing the error of weak or slow adjustments.

3. FRAMEWORK

We propose a framework following the *monitoring-decision-action* approach to autonomously manage a DBMS configuration. Typically most research activities in the field of autonomous DB management rely on such feedback-control-loop cycles to either react on certain events or threshold violations, or to refine the decision base. In Figure 1, the framework layout is sketched, including the DB side, monitoring part, and self-tuning controller.

[3]Some goals may need to be maximized, however since minimizing is the counterpart its obviously possible to simply minimize their drawbacks.

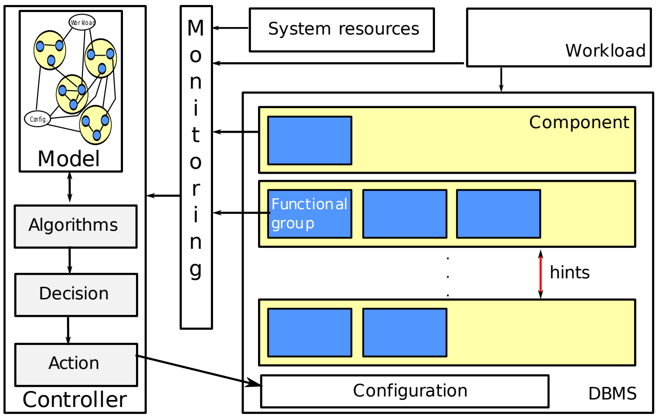


Figure 1: DB monitoring and model framework

3.1 Monitoring

Just like other approaches, we want to monitor important performance figures, state, and progress indicators from all tunable and non-tunable system parts. Even non-tunable parts are important because tuning has to observe their static characteristics; whereas tunable parts are important because tuning may change their behavior in an unpredictable way.

Important DB Monitoring Figures

Beginning at the (external) storage layer, pure IO figures (e.g., frequency, sequentiality, or randomness) and delay times are indicative sources for monitoring. Combining these figures with device specifications may help improve storage-related conclusions (e.g., emerging Flash disks (SSDs) have totally different performance characteristics compared to magnetic disks).

Following the layered architecture of nowadays DBMSs, monitoring proceeds for buffer pools. Not only hit and miss ratios, but also dynamic pool size, usage (load), switchable replacement strategies, and trend detection are important. As presented in Section 2 dynamic resizing features are cornerstones for each buffer tuning technique today.

Next, we address secondary access path structures such as indexes. Here, simple access statistics (i.e., UDI, read) and size figures may be sufficient to monitor.

More complex algorithms such as access, join, and sort operators should use the aforementioned piggybacking technique for feedback (no. of processed elements, timings) support. However, because tuning has to assign buffer sizes (e.g., input buffer for access operator, map size for hash joins, heap space for sorts) their usage has to be monitored.

To cover the entire memory distribution, monitoring has to observe all memory pool usages, i.e., query caches, result caches, lock table, log buffer, transaction contexts, and even all the other (often small) DB data structures.

Eventually, more sophisticated monitoring is necessary for workloads (i.e., queries), statistics quality, and transaction statistics (i.e., runtime, commit, abort ratios, etc.).

Important System Monitoring Figures

System resources are critical for database performance figures and, therefore, have to be monitored, too. Especially scalability is dependent on the available hardware (think of KIWI—Kill It With Iron). Thus, CPU load and process (or thread) monitoring for each CPU allows to validate theoretical runtime costs for database algorithms. However, exceptional runtime behavior such as swapping or hardware failure has to be recognized as well.

Recent attention is paid to energy consumption and, therefore, monitoring should also include power figures for system devices, i.e., CPU(s), disks, memory, fans.

Making Monitoring Lightweight

Unfortunately, monitoring consumes valuable time and system resources (CPU and memory) to collect data. Moreover, meaningful correlations may require snapshot semantics of the gathered information for a highly dynamic system. Because *suspend-collect-resume* cycles are not feasible, a lightweight monitoring has to meet all requirements. There are already approaches to prune excessive data monitoring.

A very common approach is to use epoch-based monitoring, which basically coarsens data gathering to reduce overhead. Finding suitable periods and aggregation can be achieved during runtime.

Similar to customary learning approaches, is the idea to reduce the sampling frequency for rarely changing values in order to slow down an initial eager monitoring ("learning phase"). This kind of monitoring overhead reduction is similar to aging approaches, where data sampling is triggered by renewing stale values (aging).

The monitoring component has not necessarily to reside on the same server as the DB system and threshold-based filters may reduce data volume directly at the source. Furthermore, selective monitoring may hide temporarily unused components to avoid their monitoring overhead at all as long as not being used.

3.2 System Model

Having all DB components monitored is not enough to describe or visualize internal states. We need a *system model* which allows for:

- integrating all preferable monitoring sources
- mapping physical resources to monitoring metrics
- linking correlated inputs and modeling dependencies
- making tuning knobs visible and accessible
- estimating state changes due to parameter changes

In [22], the authors recommend an economic model to allocate resources in database management systems. We want to adapt this suggestion and aim for a comprehensive *system model* crossing system layers. Building such a model is fairly complex, that's why we want to form *functional groups* and find common metrics applicable for those groups. For instance, each IO container, each buffer, and each lock table or query cache may expose timings, access frequencies, and memory sizes, which serve as common metrics very well. In parallel, system monitoring extends these basic figures by assigning resource consumption to the *functional groups*. Figure 2 exemplarily depicts a cut-out of the system model showing, for the *pyhsical layout* component, its hierarchical model elements, constraints, and relationships.

Functional groups help to create a hierarchical *system model*, where well-known tuning techniques can be locally

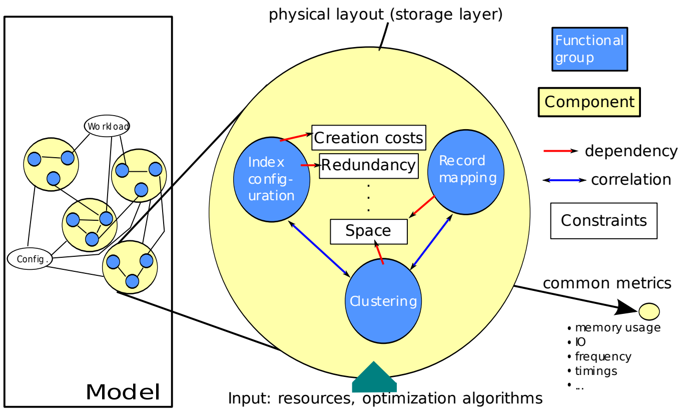
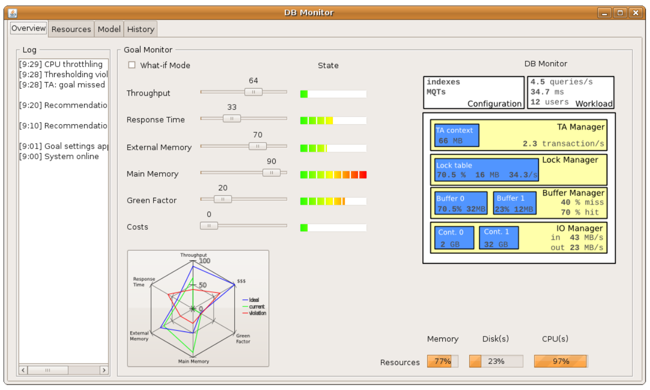


Figure 2: Physical layout example including functional groups, constrains, and relatinsships



```
[9:29] CPU throtthling
[9:28] Thresholding viol
[9:28] TA: goal missed

[9:20] Recommendatio

[9:10] Recommendatio

[9:01] Goal settings ap
[9:00] System online
```

☐ What-if Mode

Throughput 64
Response Time 33
External Memory 70
Main Memory 90
Green Factor 20
Costs 0

Configuration: indexes, MQTs
Workload: 4.5 queries/s, 34.7 ms, 12 users

TA Manager: TA context 66 MB; 2.3 transaction/s
Lock Manager: Lock table 70.5 % 16 MB 34.3/s
Buffer Manager: Buffer 0 70.5% 32MB; Buffer 1 23% 12MB; 40 % miss; 70 % hit
IO Manager: Cont. 0 2 GB; Cont. 1 32 GB; in 43 MB/s; out 23 MB/s

Resources: Memory 77%; Disk(s) 23%; CPU(s) 97%

Figure 3: GUI screenshot for a possible user front-end (including goal control and monitoring).

applied within a functional group, before we propagate their individual state (using the common metrics) to the system-wide and coarser component model. The resulting component model should serve as visualization model, indicating data hot spots, data flow, bottlenecks, and resource usage.

3.3 Algorithms

Our approach of goal-driven and autonomous DB tuning, requires several algorithms to be applied and developed for different tasks. First, we want to emphasize on model-building algorithms, before we sketch possible algorithms to efficiently search the space of alternative configurations. Note, general algorithms such as compression, aging, or sampling are taken as available.

Model-building Algorithms

For our system model, we use three compilation types:

1. **Static weights**: On the one hand, initial values such as IO parameters, CPU costs, number of internal agents (pool sizes), or memory areas have to be declared (even though these values are obtained by monitoring, too, they had to be set before). On the other hand, known correlations should be modeled and weighted in advance, even if these intitial weights turn out to be inappropriate.

2. **Heuristics**: Runtime-critical but known component behavior and correlations, e.g., buffer access, query frequency, or lock table size, should be modeled using well-known heuristics. Such heuristics enhance the quality of the initial model.

3. **Learning**: Because we want to model dynamic systems, not only the initial model has to be refined, but also non-obvious correlations determined and weak or bad correlations removed. Weights, patterns, and trends can be learned by observing the model's input/output correlations, thereby improving the model quality, reducing calculation complexity, and exposing indicative behavior.

Search Algorithms

The sheer endless search space of possible configurations (see Definition 2) prevents explorative brute-force search. Therefore, we make use of our hierarchical model to alleviate the search complexity. *Functional groups* have to find their local optimum using Definition 3 before propagating to the system-wide components in order to find a global optimum. Two more features may help to reduce search effort, first, a feature skyline to prune the search space and second, knowledge about the search quality of previous searches. Note, we do not rely on specific pruning or quality measures in favor to keep them exchangeable.

For instance, the introductive (index) self-tuning example shows that benefit calculation easily leads to the problem of Cartesian products (solution is NP-complete). Thus, searching has to be flexible and smart enough to integrate new measures, because the combinatorial space of all existing input parameters is doubled for each new parameter.

Our system model supports extensibility, because we want to integrate new algorithms into the chain of existing algorithms or to substitute them. Thereby, the quality of modeling as well as the computation and adaptation speed may be improved, too.

Since dynamic changes within the system (i.e., state, configuration, scheduling, load, etc.) impose more complexity to the model, we advocate a hint mechanism enabling components (or functional groups) to convey their current state among each other directly.

Eventually, the model allows to quantitatively explore the impact of user-proposed changes using the model's algorithmic estimations.

3.4 User Front-end

Although self-tuning features should work unattended, meaningful visualization is required to verify their effectiveness and to allow for self-explanatory user interaction knobs. Thus, one aspect is the observability of the entire system performance including facilities to descend into certain (more interesting) components to selectively monitor them in more detail.

For goal-driven processing, the user needs appropriate tools to control the goal(s). This should be able by weighting the available goals depending on current user needs. In Figure 3, a GUI screenshot shows the sliding-based goal control. For instance, savings may require to throttle throughput to reduce resource load which, in turn, leads to lower power consumption. Specifying operating templates may help to configure such a system running autonomously with different types of goal configurations depending on, e.g., a daily schedule, current load situation, or electricity tariff.

Another important feature of the user front-end is the recommendation of non-autonomous tuning tasks. For instance, backed by the system model, an estimation algorithm tries to increase goal compliance by evaluating changes in available resources. This implies removing or adding hardware components such as disks, CPUs, or memory. Moreover, having costs (for purchase and for operating) and power consumption specified the recommender component may estimate favorable assemblies.

4. CONCLUSIONS

This paper presents the main issues and concepts of self-tuning database systems which will be important for my PhD work. The main objectives of that work are: First, extend the existing monitoring framework [15] for our prototype XDBMS; second, refine the functional groups, components, and the system model; and third, develop a user-front allowing to specify operational goals. Some of the functional groups are already addressed such as workload-dependent storage structures in [17], adaptive storage structures in [16], autonomous indexing in [11, 12], and energy savings in [8]. The functional group approach allows to develop each part using existing techniques. However, the preliminary ideas and achievements so far are promising for an integrated but extendable and generic system model.

In the future, more self-* features such as self-healing may be interesting. For instance, detecting data structure failures (e.g., in the buffer) to restart the affected component, or to handle disk-overflow errors by triggering a "vacuum" run and switching to read-only mode may help to solve such problems. Therefore, the system model and the adaptive algorithms are helpful when deciding where, how, and if "healing" is possible. Moreover, having multi-tenancy systems, how to isolate different customer goals within one system or how to push a common goal is challenging. Another interesting area is the application of our approach within virtual environments where resource modeling is fairly different.

5. REFERENCES

[1] A. Aboulnaga, P. Haas, M. Kandil, S. Lightstone, G. Lohman, V. Markl, I. Popivanov, and V. Raman. Automated statistics collection in DB2 UDB. In *VLDB Proc.*, pages 1158–1169, 2004.

[2] S. Agrawal, E. Chu, and V. Narasayya. Automatic physical design tuning: workload as a sequence. In *SIGMOD Proc.*, pages 683–694, 2006.

[3] S. Chaudhuri and V. Narasayya. Automating statistics management for query optimizers. *IEEE Trans. on Knowl. and Data Eng.*, 13(1):7–20, 2001.

[4] S. Chaudhuri and V. Narasayya. Self-tuning database systems: a decade of progress. In *VLDB Proc.*, pages 3–14, 2007.

[5] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *VLDB Proc.*, pages 146–155, 1997.

[6] S. Chaudhuri and V. R. Narasayya. AutoAdmin 'What-if' index analysis utility. In *SIGMOD Proc.*, pages 367–378, 1998.

[7] K. E. Gebaly and A. Aboulnaga. Robustness in automatic physical database design. In *EDBT Proc.*, pages 145–156, 2008.

[8] T. Härder, K. Schmidt, Y. Ou, and S. Bächle. Towards flash disk use in databases - keeping performance while saving energy. In *BTW Proc.*, pages 167–186, 2009.

[9] M. P. Haustein and T. Härder. An efficient infrastructure for native transactional XML processing. *Data Knowl. Eng.*, 61(3):500–523, 2007.

[10] M. Holze and N. Ritter. Towards workload shift detection and prediction for autonomic databases. In *PIKM Proc.*, pages 109–116, 2007.

[11] M. M. Hossain. Autonomous indexing and management in a native XML database management system. Master thesis (supervisor: K. Schmidt), 2008.

[12] M. Lühring, K.-U. Sattler, K. Schmidt, and E. Schallehn. Autonomous management of soft indexes. In *ICDE Workshops*, pages 450–458, 2007.

[13] P. Martin, S. Elnaffar, and T. Wasserman. Workload models for autonomic database management systems. In *ICAS Proc.*, page 10, 2006.

[14] B. Niu, P. Martin, W. Powley, R. Horman, and P. Bird. Workload adaptation in autonomic DBMSs. In *CASCON Proc.*, page 13, 2006.

[15] Y. Ou. Performance analysis and optimization of the XML database system: XTC. Diploma thesis (supervisor: K. Schmidt), 2008.

[16] K. Schmidt and T. Härder. An adaptive storage manager for XML documents. In *BTW Workshops*, pages 317–328, 2007.

[17] K. Schmidt and T. Härder. Usage-driven storage structures for native XML databases. In *IDEAS Proc.*, pages 169–178, 2008.

[18] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's learning optimizer. In *VLDB Proc.*, pages 19–28, 2001.

[19] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in DB2. In *VLDB Proc.*, pages 1081–1092, 2006.

[20] D. N. Tran, P. C. Huynh, Y. C. Tay, and A. K. H. Tung. A new approach to dynamic self-tuning of database buffers. *Trans. Storage*, 4(1):1–25, 2008.

[21] G. Valentin, M. Zuliani, D. C. Zilio, G. M. Lohman, and A. Skelley. DB2 Advisor: An optimizer smart enough to recommend its own indexes. In *ICDE Proc.*, pages 101–110, 2000.

[22] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *VLDB Proc.*, pages 20–31, 2002.

[23] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: integrated automatic physical database design. In *VLDB Proc.*, pages 1087–1097, 2004.

[24] D. C. Zilio, C. Zuzarte, G. M. Lohman, H. Pirahesh, J. Gryz, E. Alton, D. Liang, and G. Valentin. Recommending materialized views and indexes with IBM DB2 design advisor. In *ICAC Proc.*, pages 180–188, 2004.