Towards Model-based Management of Database Fragmentation

Asim Ali
Qatar University

Rui Jia
Mississippi State University

Abdelkarim Erradi
Qatar University

Sherif Abdelwahed
Mississippi State University

Rachid Hadjidj
Qatar University

Abstract

The performance of a database can significantly deteriorate due to the fragmentation of data/index files. Manual database defragmentation and performance optimization remain time consuming and even infeasible as it requires knowledge of the complicated behavior of fragmentation and its relationships with system parameters. We propose a model-based detection and management framework for the database fragmentation which can automatically optimize database performance, detect the fault existence, estimate its future impact on system performance and recover the system back to normal. A predictive controller is designed to take proper actions to guarantee the QoS and remedy faults. Experimental studies on a realistic test-bed show the applicability and effectiveness of our approach.

1 Introduction

Relational Database Management Systems (RDBMS) are critical components of enterprise applications. They provide multi-user access to persistent data efficiently while preserving data safety and consistency despite failures and concurrent updates. The access time of secondary storage is the major factor in RDBMS system performance as the disk I/O dominates all other database operations in term of cost. Therefore, the goal of improving the performance of RDBMS file systems is to minimize the time spent waiting for disk I/O. To achieve it, related data is stored on contiguous disk blocks to reduce seek time and rotational delay. However data files may still get fragmented over time after a large number of data modification operations.

Unmanaged fragmentations not only degrade the system performance by increasing the response time and decreasing the throughput, but also waste valuable resources such as CPU time, energy, and etc. Ultimately they will cause the system to violate its Service Level Agreements (SLA) [12]. Although periodic reorganization of database files is a way to alleviate the symptom [7], the management system needs to detect fragmentations and take corrective actions at the appropriate time to avoid SLA violations. For example, the possible actions for dealing with the fragmentation could be a defragmentation operation or to shift the system load to another standby server. Because defragmentation is very costly and time consuming, it is very critical to select the correct database tables/indexes and correct time to conduct defragmentation. A suboptimal decision in this regard may lead to performance degradation, SLA violations and revenue loss. In practice, defragmentation decisions are mostly manual and data driven. Administrators decide to defragment database objects when the fragmentation level reaches a certain threshold [6]. The problem with this approach is that it does not consider the impact of fragmentation on the Quality of Service (QoS) and consequently unnecessary defragmentation might be triggered that do not significantly affect the system performance.

In this paper, we propose a model-based disk fragmentation management framework where a system model can estimate the QoS parameters based on the fault level and the system load. The system model can be used to predict the future levels of fragmentation and system parameters based on the expected incoming load. A fragmentation detection method is developed to facilitate the fault recovery process. A predictive controller can decide the optimal and timely corrective actions for both performance management and fault recovery, given the detection results and the system model. The proposed framework can deal with disk fragmentations automatically which significantly reduces human intervention.

2 Related Work

Database systems are essential and important parts of enterprise multi-tier applications and hence form a pros-

perous domain for the application of autonomous management concepts [9]. In this area, research efforts have been focused towards autonomic workload management, query optimization, storage management, memory management, and etc. A survey of autonomic workload management techniques in databases is given in [8]. In [2] authors described a framework for diagnosing the query slow-down problem in databases. Authors in [11] presented a model-based approach for database tuning.

The fragmentation is a common database problem that grows over time and negatively impacts system performance [12]. This impact is observed in the form of increased response times, decreased throughput and increased load on system resources. Many commercial databases provide tools to estimate internal and external database fragmentations. These tools also provide commands to defragment (rebuild or reorganize) database tables and indexes. We argue that there are two distinct challenges related to defragmentation. The first is to decide when to launch a defragmentation operation and the second is to select the indexes to be defragmented. Existing research work advocate either a data driven or a workload driven approach for managing fragmentations [10, 3]. In the data driven approach described in [6], indexes are selected for defragmentation based on a set of heuristics based on the fragmentation level, index size and frequency of the index scan. Authors in [7] presented a workload driven approach where the workload information was used to select the appropriate indexes (or a part of indexes) to defragment so as to maximize query I/O performance. However, concerning the question when to defragment an index, most DBAs decide to use some rules of thumbs, such as defragmenting an index when its fragmentation level is above a threshold.

Our work complements the previous research by providing a model-based fragmentation management framework which can detect and cure the fault while optimizing the performance to meet the QoS standard. This is an extension of our previous work in [5]. To the best of our knowledge, an autonomic model-based approach for managing the database fragmentation has not been addressed in published literature, and this motivated us to tackle this problem.

3 Fault Management Framework

Fig.1 shows the high level architecture of the proposed model-based fault management framework for computing systems. The main goal of the proposed structure is to enable the system to identify and manage undesired conditions and changes in the operating environment that may cause performance issues. The following parts of this section are brief descriptions of the different system components and their interactions.

3.1 Managed System

The *Managed System* is the target computing system to be managed. It hosts a set of software monitoring sensors that collect system events and metrics about the state of the system which are passed to the *Observer*. The collected data at time t is expressed as the vector $Y(y_1, y_2, .., y_k)$, where each y represents some software or hardware system parameters (e.g., average execution time or CPU usage, and etc.). For each parameter y there is a sensor deployed in the system that collects the value of the parameter at a specified frequency. The *environment* sends the workloads to the *Managed System*, as well as system related events such as the addition or the removal of system resources.

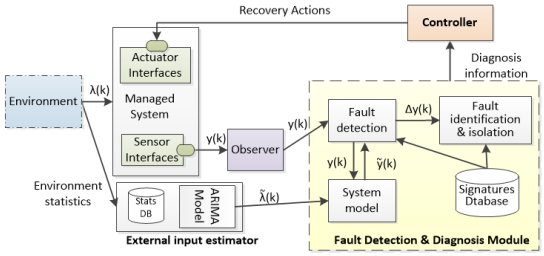

Figure 1: Key Components of Model-based Fault Management Framework

3.2 Fault Detection and Identification

This module uses a model-based diagnoser [5] where a *System Model* is used to detect and identify the possible faults in the system. A system model describes the system behaviors under different workloads and fault conditions, which can be developed offline by analyzing the historical performance data or through extensive simulations. Expected workload comes from the external input estimator.

During the diagnosis process the current state of the system represented by vector $Y = (y_1, y_2, ..., y_k)$ is passed to the diagnosis system. The *Fault Detection* module uses the system model to compute the estimated parameter value $\hat{y}$ for each $y \in Y$ corresponding to the current system load, which also generates a vector of residuals $R(r_1, r_2, .., r_k)$ (diagnostic signals) where each $r_i = y_i - \hat{y}_i, i = 1, 2, ..., k$ is the difference between the observed value and the estimated value for the parameter y_i. A threshold value test is applied on generated residuals to determine if the system is in a faulty or a normal state with reference to the residuals used in the test, which are also called as the fault signature. The fault signatures are stored in a database. The *Fault Identification and Isolation* process compares the residual vector with each fault in the signature database and the fault having the closest

signature to the observed residual vector is declared as the potential fault in the system.

3.3 Controller

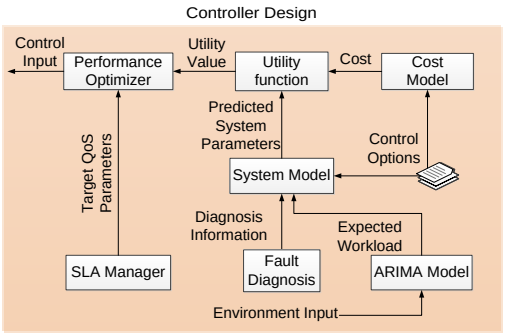

Figure 2: The Controller Module

Architecture of the proposed controller for managing the database fragmentation is shown in Fig. 2. The *Controller* module uses the *System Model* to predict system states (throughput, response time values). The *Fault Detection and Diagnosis* module provides the performance statistics as well as fault diagnosis information including the fault class and the fault intensity. There is a library consisting of the possible control actions to manage the performance and to cure faults of the target system. For each control action, the performance optimizer computes its cost with the help of a cost model and the expected system performance corresponding to the control action with the help of the system model. The cost and benefit analysis of the control actions are formulated in the form of a utility function. Finally, performance optimizer chooses the control action with the minimum utility cost that ensures the system meets the SLA requirements. This control input is executed on the managed system through the actuator interface.

4 Controller Design

In normal conditions, a database system runs under different types and levels of workloads. To ensure QoS under varying workload conditions, a number of available configuration options such as buffer size changes should be used. However when database performance degrades due to a specific fault such as fragmentation, the controller needs to recover the database from fragmentation in addition to performance management through configuration changes.

Based on these design requirements, the set of available control options is divided into two categories: management actions and recovery actions denoted by the sets U_M and U_R, respectively. Usually $U_M \cap U_R \neq \varnothing$. While the system is under a recovery process, U_M can also be executed in parallel if the performance management is necessary. In this work, U_M consists of different sizes of the database buffer pool such as 128MB, 256MB and 512MB. As part of the recovery actions, we only consider two defragmentation methods: rebuilding and reorganization. Rebuilding process removes fragmentation by recreating the indexes. This process is normally executed in offline mode. In contrast reorganization is online defragmentation process during whcih table indexes are reorganised. Suppose u_{R1}={Rebuilding} and u_{R2}={Reorganization}, we have $U_R = \{u_{R1}, u_{R2}\}$. Similarly, $U_M = \{u_{M1}, u_{M2}, u_{M3}\}$ where u_{Mi} represent the events of setting the buffer size to 128MB, 256MB and 512MB, respectively.

4.1 Cost Model

Controller examines the system state at regular sampling intervals. When performance degradation is detected, the controller needs to choose and execute appropriate management and/or recovery actions. Each control action in U_M and U_R has some cost associated with it that could be expressed in terms of hardware or software resource requirements, computing and financial overhead, the impact on system performance, increased number of SLA violations during the control execution, and etc. To choose the best set of control inputs, controller computes the tradeoff between the performance degradation (cost) due to the control actions and the effectiveness (gain) of them on both performance and fault management in the current environment conditions. For this purpose, a set of utility functions are defined which outputs the cost and benefit analysis of a control input as a single utility value. Here, a negative utility value mean a performance gain, and vice versa. The system model and the cost model are used to compute the utility values. The input to the system model comes from three sources. The diagnosis module provides current and expected system states, which include the performance statistics (e.g., throughput and response time) as well as fault state (e.g., the fragmentation level). Expected workload comes from the external input estimator and the control input is provided by the control library. Using the three inputs, the system model computes the expected system performance parameters and passes this information to the utility function. The performance optimizer chooses a control input that has the minimum utility cost value.

The utility function for management actions J_M and for the recovery actions J_R are defined in terms of the percentage of response time increase and transaction throughput decrease.

The utility of a management action $u_M(i)$ at time i is

defined in Equation 1, where $r_x^M(i)$ is the response time during the execution of $u_M(i)$, r^* is the set point of response time, $\theta_x^M(i)$ is the transaction throughput during the execution of $u_M(i)$, θ^* is the set point of the throughput, $T_x^M(i)$ is the total execution time of $u_M(i)$, $\hat{r}^M(i)$ is the estimated response time after $u_M(i)$, $\hat{\theta}^M(i)$ is the estimated throughput after $u_M(i)$, and T is the sampling interval. All the estimated values are based on the current fault level and workload condition. This function reveals the tradeoff between cost of executing the action, and the performance gain in the next sampling interval.

$$J_M(i) = \left(\frac{r_x^M(i) - r^*}{r^*} + \frac{\theta^* - \theta_x^M(i)}{\theta^*}\right) \times T_x^M(i) + \left(\frac{\hat{r}^M(i) - r^M(i-1)}{r^M(i-1)} + \frac{\hat{\theta}^M(i) - \theta^M(i-1)}{\theta^M(i-1)}\right) \times T \quad (1)$$

The utility of a recovery action $u_R(i)$ at time i is defined in function 2, where $\hat{r}^R(i)$ is the estimated response time, $\hat{\theta}^R(i)$ is the estimated throughput, and the other symbols have the same meanings as those in function 1. All the estimated values are based on the current fault level, recovery actions and workload condition.

$$J_R(i) = \left(\frac{\hat{r}^R(i) - r^*}{r^*} + \frac{\theta^* - \hat{\theta}^R(i)}{\theta^*}\right) \times T \quad (2)$$

4.2 Control Algorithm

We propose a modified Limited Lookahead Control (LLC) algorithm [1] which has the lookahead horizon H_M for the management actions and H_R for the recovery actions. We assume that $H_M << H_R$ due to the long-term nature of disk defragmentation. The responsibility of the proposed algorithm is not to develop an optimal way of choosing the indexes for defragmentation. In fact, once the controller decides to launch a defragmentation operation, any strategy proposed in literature could be adopted to choose the appropriate indexes.

In normal conditions, the controller generates the management control inputs to meet QoS in the system, which uses the traditional LLC given H_M. Suppose $H_M = m$ and the current sampling interval is the kth, the total cost of a given path is

$$J_{normal}(k) = \sum_{i=k}^{k+m-1} [W_M(i) J_M(i))]$$

where $W_M(i)$ is an user defined weight of the ith management action. The control objective is to find the path with the minimum total cost (minimum cost refers to the maximum utility) $J_{normal}(k)$. The first management action of this path is selected as the control input. This mechanism works in both normal and faulty situations. When there is a fragmentation, the output of the controller is a recovery plan in the set U_R. Figure 3 shows the idea of finding the appropriate recovery action. For the simplicity of illustration, it only shows three lookahead horizons (H_R), although it can be farther more practically. The interval between each state is T. The forecasting paths are established by enumerating all the effective recovery actions, estimating the future system states and evaluating the healthiness of them until the end of the horizon. The system state under a given fault level and after a recovery action can be predicted based on the historical experiment data. In this case, the recovery horizon H_R is the number of sampling intervals which is selected based on how further we are interested in the future system states. Suppose the number of horizons of a given path is n, and the current sampling interval is the kth, the total cost of a path with recovery action $u_R(j)$ is

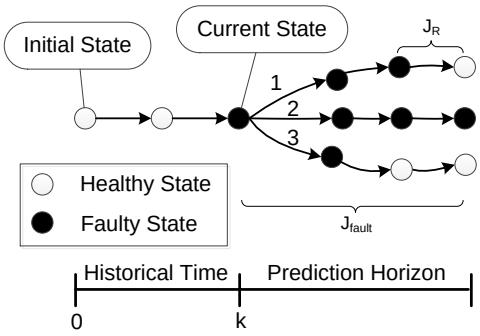

Figure 3: The Fault Adaptive Prediction Control Method

$$J_{fault}(j) = W_R(j) \sum_{i=k}^{k+n-1} J_R(i)/n$$

where $W_R(j)$ is an user defined weight of the jth recovery action. This cost function denotes the average cost per interval given H_R. Note that both $H_R > T_x^R(j)/T$ (Path 1, 3) and $H_R \leq T_x^R(j)/T$ (Path 2) are possible, where $T_x^R(j)$ is the execution time of the jth recovery action. When $H_R > T_x^R(j)/T$, the estimation of response time and throughput should be adjusted accordingly.

5 Case Study

Based on the fault management framework and the control algorithm described in the previous sections, we built a testbed to implement our ideas. The testbed consists of three servers connected via 1GB Ethernet. Each machine

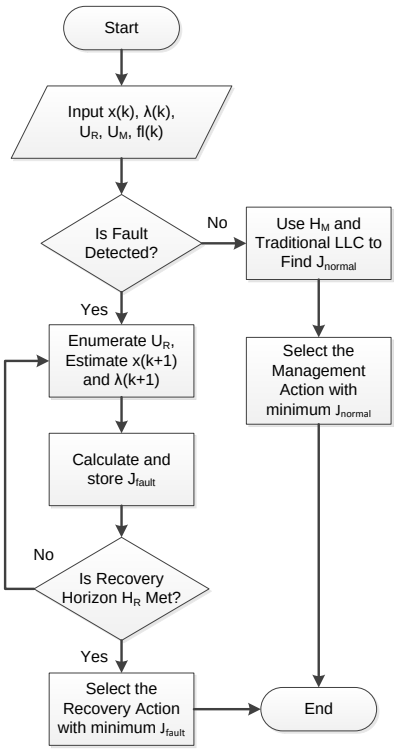

Figure 4: Adaptive LLC algorithm

runs CentOS 5.7 operating system with a 2GB RAM and a 40 GB hard disk. The Daytrader [4] is used as the Web application and the database is MySQL. The system model for fragmentation detection & diagnosis is developed through simulations on the testbed. The fragmentation is injected by a script that inserts and deletes the records in the StockQuote table following a particular strategy to produce the required level of the fault. An example showing how the controller selects different control actions is provided at the end of this section.

5.1 System Model

As discussed above, the system behavior is modeled under different workload and fault conditions. For managing database fragmentation, the proposed system model consists of two parts: a performance model that predicts the QoS parameters (throughput, response time) for various arrival rates and a fault signature that models the impact of the fragmentation on QoS as well as on other system parameters such as CPU usage, RAM usage, and etc. To develop the system model, we simulated the system under different workload and fragmentation levels and analyzed the data using tools such as regression analysis. Table 1 describes the performance model for fragmentation level 0.

Arrival Rate	Throughput	Avg. Resp. T
$\lambda_i < 350$	$0.9 \times \lambda_i + 8.9$	$224ms$
$\lambda_i \geq 350$	$0.01 \times \lambda_i + 318.1$	$189ms$

Table 1: System Performance Model

In Table 1, λ_i denotes the arrival rate at the time i. The second and third columns give the throughput and the average response time values corresponding to the arrival rate λ_i. The sampling interval T is one minute. Arrival rate is the number of read operations observed during one minute and the throughput is the number of read operations successfully completed per minute. During simulations we observed that system reached its maximum capacity when the arrival rate was 350 transactions per minute (TPM) which is considered as the threshold arrival rate. With respect to arrival rate, we define that the system is in state A when the arrival rate is less than 350 TPM, and it is in state B when the arrival rate is greater than 350 TPM. The average response time observed during each sampling interval in state A was approximately 224 ms, which in state B was 189 ms.

To develop the fragmentation signature we ran the same workload under different fragmentation levels in the system and studied the behavior of the following parameters: the throughput, the average response time, the disk read operations, the CPU I/O waiting time percentage and the memory usage. To reduce complexity, we assume there is no correlation among selected system parameters and their values change only because of fragmentation levels. We simulated the system in both state A and B with respect to arrival rate. Fig.5 is a part of the fault signature showing changes in the throughput and the average response time due to different fragmentation levels. In this experiment, the arrival rate was changing but was always greater than 350 TPM (the threshold value). The data in Fig.5 shows the average values achieved and the standard deviation for the 35 minutes. It also shows the percentage variation in the parameter value regarding to the fragmentation level 0. Using the curve fitting techniques on the data given in Fig.5, we draw a formula relating the response time R and the fragmentation level fl in the system as follows

$$fl = 0.212 * R - 225$$

Following the same procedure the relationships between the fragmentation level and the other monitored parameters can be developed. All these equations form the fault signature, which can be used for diagnosing the intensity of the fault as well as for predicting the future system behavior.

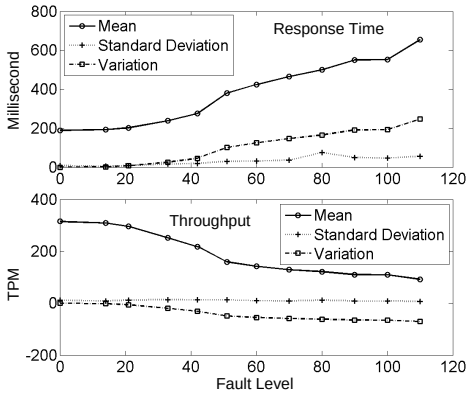

Figure 5: The Performance Statistics

5.2 System Model Application

An example is given to describe the use of the system model (Table 1 and Fig.5) to detect and diagnose a fragmentation fault in the system. Let us suppose that the current arrival rate is 400 requests per minute and the observed throughput value in the last sampling interval was 200 TPM. The detection module uses the system model to compute the estimated throughput for this arrival rate (i.e., based on the system model in Table 1, the expected throughput is 322 TPM). Comparing the estimated throughput of 322 TPM and the observed value of 200 TPM the detection module computes the percentage decrease which is approximately 38%. If we have fixed the tolerated deviation of observed and expected throughput values at 30%, then a fragmentation will be declared in the system. Using the observed deviation of value of 38% between throughput values and the fault signatures in Fig.5, the diagnosis module estimates that the fragmentation level in the system is between 42% and 51%.

5.3 Control Algorithm Application

We did a series of experiments to quantify the utility cost J_M and J_R in Section 4. Table 2 shows the impact on the response time (RT) during and after the defragmentation operation, where the first column is the buffer size and the experiment mode, the second column is the average RT, and the last is the percentage variation (a part of J_M) which computes the RT increase as the percentage of the RT achieved in the normal system state. Data in these tables was generated at the fragmentation level 69. Due to space constraint, the similar results for other throughput levels are not included.

As an example, let us assume the controller needs to calculate the cost of the management actions J_{normal}. Using Table 2 the set point response time can be defined as 216ms while the current RT is $r(k)$. If the prediction horizon is 1 and we only consider RT in Equation 1, the utility cost of the management action $\{$128MB buffer size$\}$ is $J_{normal}(k) = W \times [r(k) - 216]/216 + 224$ where W is the user selected weight of the actions. Similarly, the cost of other management actions can be calculated. When the controller prepares a recovery plan, it should also estimate the cost of recovery actions J_R. Thus, $J_R(online) = (1197 - 216)/216 \times T$, while $J_R(offline) = (\pi - 216)/216 \times T$ where π is a user defined penalty value since there are no observable RT in the blackout period. According to our experiment, the online defragmentation takes 6627 seconds whereas the offline counterpart takes only 2359 seconds, given the 128MB buffer size. If the recovery horizon is 100 (6000 seconds), $J_{fault}(online) = J_R(online) \times 600$ and $J_{fault}(offline) = J_R(offline) \times 39 + J_R(normal) \times 61$, where $J_R(normal)$ is the utility cost due to the parameters after the offline recovery.

Buffer Size, Mode	Resp. T.	%Var.
128MB, before failure	216	0
128MB, before recovery	701	224
128MB, during recovery	1197	219
128MB, after recovery	279	29
256MB, before recovery	709	228
512MB, before recovery	716	231
1024MB, before recovery	803	271

Table 2: Response Time under Various Conditions

6 Conclusion

We have proposed a general model based framework for fault management in distributed systems and its application to database fragmentation. We developed the system model to detect fragmentation faults, a control algorithm and the necessary control models. This work is a part of an ongoing project and as a next step we are implementing and evaluating a fully automatic online controller for dealing with database fragmentation. In the future we will test the proposed framework using variety of the faults in the domain of large distributed systems.

7 Acknowledgments

This paper was made possible by NPRP grant # NPRP 09-778-2299 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

References

[1] ABDELWAHED, S., KANDASAMY, N., AND NEEMA, S. Online control for self-management in computing systems. In *Proc. 10th IEEE Real-Time and Embedded Technology and Applications Symp RTAS 2004* (2004), pp. 368–375.

[2] BORISOV, N., UTTAMCHANDANI, S., ROUTRAY, R., AND SINGH, A. Why did my query slow down? In *CIDR 2009 - 4th Biennal Conference on Innovative Data Systems Research* (2009).

[3] BRUNO, N., AND CHAUDHURI, S. An online approach to physical design tuning. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on* (2007), pp. 826–835.

[4] IBM. The apache daytrader benchmark, 2012. Available at https:// cwiki.apache.org/GMOxDOC20/daytrader.html.

[5] JIA, R., ABDELWAHED, S., ERRADI, A., HADJIDJ, R., AND ALI, A. A model-based framework for automatic recovery from incipient faults in computing systems. In *The 7th International Workshop on Feedback Computing* (2012).

[6] MORELLI, E., ALMEIDA, A., LIFSCHITZ, S., MONTEIRO, J. M., AND MACHADO, J. Autonomous re-indexing. In *Proceedings of the ACM Symposium on Applied Computing* (2012), pp. 893–897.

[7] NARASAYYA, V., AND SYAMALA, M. Workload driven index defragmentation. In *Proceedings - International Conference on Data Engineering* (2010), pp. 497–508. Cited By (since 1996): 2.

[8] RAZA, B., MATEEN, A., AWAIS, M. M., AND SHER, M. Survey on autonomic workload management:algorithms, techniques and models. *JOURNAL OF COMPUTING 3* (2011).

[9] RAZA, B., MATEEN, A., HUSSAIN, T., AND AWAIS, M. M. Autonomic success in database management systems. In *Proceedings of the 2009 8th IEEE/ACIS International Conference on Computer and Information Science, ICIS 2009* (2009), pp. 439–444.

[10] SATTLER, K.-U., SCHALLEHN, E., AND GEIST, I. Autonomous query-driven index mining. In *Database Engineering and Applications Symposium, 2004. IDEAS '04. Proceedings. International* (2004), pp. 439–448.

[11] SCHMIDT, K. Goal-driven autonomous database tuning supported by a system model. In *Third SIGMOD PhD Workshop on Innovative Database Research (IDAR)* (2009).

[12] TECHNOLOGY, W. S. Fragmentation and database performance. Tech. rep., White Sands Technology, Inc., Canoga Park, CA, USA, 2004.