Discovering Indicators for Congestion in DBMSs

Mingyi Zhang[#1], Patrick Martin[#2], Wendy Powley[#3], Paul Bird[*4], Keith McDonald[*5]

[#]*School of Computing, Queen's University*
Kingston, Ontario, Canada
[1]myzhang@cs.queensu.ca
[2]martin@cs.queensu.ca
[3]wendy@cs.queensu.ca

[*]*IBM Toronto Lab*
Markham, Ontario, Canada
[4]pbird@ca.ibm.com
[5]kmcdonal@ca.ibm.com

***Abstract*— In today's data server environments, multiple types of workloads can be present in a system simultaneously. Workloads may have different levels of business importance and unique performance goals. An autonomic workload management system controls the flow of the workloads to help the database management system (DBMS) meet the performance goals. A task of the autonomic workload management system is to prevent congestion in the DBMS, which can result in severe degradation in overall system performance. Autonomic workload management should detect that a system is becoming congested and then act to restore normal system operation. In this paper, we describe an approach to identify a set of database monitor metrics that can serve as indicators for potential congestion in a specific scenario. We present experiments to illustrate two cases of congestion in a DB2® DBMS and use our approach to derive the indicators.**

I. Introduction

A database workload is a set of requests that have some common characteristics such as application, source of request, type of query, priority, and performance goals (*e.g.*, response time or throughput objectives) [1]. Workload management for database management systems (DBMSs) is a performance management process whose primary objective is to achieve the performance goals of all workloads, while balancing resource demands of the workloads to maximize overall performance of the entire system. In today's data server environment, multiple types of workloads, such as on-line transaction processing (OLTP) and business intelligence (BI) workloads, can be mixed and present simultaneously. Workloads generated by different applications or initiated from different business units may have unique performance goals, which are normally expressed in terms of service level agreements (SLAs) that must be satisfied for business success.

Multiple requests running concurrently on a data server inevitably compete for shared system resources, such as CPU cycles, buffer pools in main memory, disk I/O bandwidth, and various queues in the database system. If some requests, for example, long BI queries, are allowed to use a large amount of system resources without control, other concurrently running requests may have to wait for the long queries to complete and release the used resources, thereby resulting in the waiting requests missing their performance goals and the entire data server suffering degradation in performance. Moreover, the mix of workloads presenting on a data server can vary dynamically and rapidly, so it becomes virtually impossible for database administrators to manually adjust the system configuration to achieve the performance goals of the workloads during runtime. Therefore, *autonomic* [5] workload management becomes necessary and critical to control the flow of the requests and manage their demands on shared system resources to reach the workloads' performance goals.

In extreme cases, the absence of workload management can cause the DBMS to become congested, which results in severe reductions in overall system performance. In this study, we define a DBMS as *congested* when competition among executing requests for one or more shared system resources causes work in the system to stagnate and to not make any noticeable progress.

The workload management system must monitor the DBMS to detect approaching congestion and then take the necessary actions to avoid it. Simply monitoring performance metrics such as throughput and response time is not sufficient as workload control actions need to be taken before the metrics present the effects of congestion (*i.e.*, severe degradation in performance). We therefore propose using DBMS monitor metrics to detect approaching congestion.

DBMSs make a very large number of metrics available and collecting and analysing this information can impose significant overhead and cause high latency. Selecting a small number of specific metrics that best indicate congestion is, however, not obvious since the choice of metrics depends on properties of the system configuration and the workloads. The contribution of this work is an approach to discovering *system-congestion* indicators for a DBMS scenario, which can be used to indicate when the DBMS is entering a congested state. Since a vast number of monitor metrics are available in a DBMS, a data mining approach is used to identify the key metrics.

The paper is organized as follows. Section II reviews the previous work related to this study. Section III describes our approach to *system-congestion* indicator discovery in a DBMS. Section IV presents experiments to illustrate congestion in DBMSs leading to system performance deterioration and demonstrates the use of our approach to derive the indicators. Finally, we conclude our work and propose future research in Section V.

II. BACKGROUND

In the past several years, considerable progress has been made in workload management for DBMSs. New techniques have been proposed by researchers, and new features of workload management facilities have been implemented in commercial DBMSs. These workload management facilities include IBM® DB2® Workload Manager [6], Microsoft® SQL Server Resource Governor [10], Oracle® Database Resource Manager [14], and Teradata® Active System Management [17]. These workload management facilities control complex workloads (*e.g.*, a mix of business processing and analysis requests) present on a data server using predefined procedures, based on the characteristics of the requests, such as estimated costs, resource demands, and execution time.

Recent research [7], [11] shows that a process of workload management for DBMSs can involve three controls, namely, admission, scheduling, and execution controls. Admission control determines whether or not an arriving request can be admitted into a database system, thus it can avoid increasing the load while the system is busy. Scheduling determines the execution order of admitted requests based on some criteria, such as the request's business importance and/or performance goals. Execution control dynamically manages some running requests to limit their impact on other concurrent queries.

Applying autonomic computing theory in DBMSs has been extensively studied in the past decade [2], [9]. In particular, in the studies of autonomic workload management for DBMSs, many autonomic components have been developed and shown to be useful in their own right, such as the autonomic request throttling approaches [15], [16], which dynamically control large requests in a database system based on specified high-level business objectives, and the autonomic query scheduler [13], which manages the execution order of multiple classes of queries present on a single data server to achieve their service level objectives. Despite all the efforts to provide facilities to dynamically manage highly varied and frequently changing workloads, a fully autonomic database workload management system has not yet been realized.

Performance degradation of a database system may happen for different reasons, such as *thrashing* [3] and request congestion. Thrashing may occur in a database system for (at least) two distinct reasons, namely, memory contention or data contention [11], [12]. Thrashing behavior of DBMSs has been intensively studied, and a set of performance metrics were proposed for indicating performance deterioration in OLTP systems [12] and in a BI batch data warehousing system [11]. In this paper, we propose a generic approach to discovering indicators of potential problem states in a specific DBMS like congestion and thrashing.

III. INDICATOR DISCOVERY

Vast amounts of monitoring data can be collected while a DBMS is servicing requests. Processing and analysing this data can be time consuming and introduce significant system overhead and latency. Often, however, a small subset of the data can indicate a dangerous state. We apply data mining techniques to a set of collected monitoring data to identify the most important monitor metrics and use them to indicate the impending system congestion. In this approach, we collect monitoring data from the DBMS in both its *normal* and *transition* regions (discussed in Section III-A), and use this as training data for the system. Then, we mine the metrics from the training data, extracting those which are key to identify that DBMS congestion is about to occur.

The approach consists of three main components, namely, *data collection*, *data pre-processing*, and *attribute selection*. Section III-A explains how a set of training data can be experimentally collected, Section III-B presents a way of pre-processing the collected raw data, and Section III-C discusses the analysis of the pre-processed data to discover indicators.

A. Data Collection

For a specific DBMS scenario (configuration and set of workloads) our approach divides the DBMS state space into three mutually exclusive regions: *normal*, where workloads are meeting their SLAs and/or general system performance is at desired levels; *transition*, where the system starts to suffer some degradation in performance; *congested*, where requests running in the system are not progressing, and the system is experiencing severe degradation in performance.

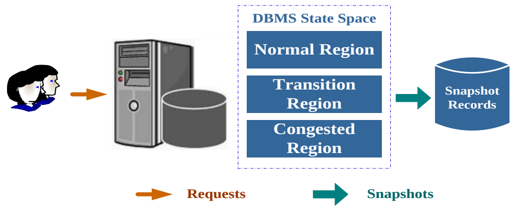

Fig. 1 Experimental Framework for Training Data Collection

A sample scenario is shown in Figure 1. A set of training data is collected by running the DBMS under load. Initially, the data server runs a light load, and its state is in the *normal* region. By incrementally adding additional load onto it, the state of the server changes, conceptually moving from its *normal* region to its *transition* region and eventually to its *congested* region. The data server is sampled periodically by taking snapshots of the system running in all three regions. The samples (*i.e.*, the system monitoring data) are saved in a database (or a set of text files) along with a timestamp as they are taken. After the experiment, the samples are traced back with the timestamps, and a set of samples taken in the *normal* region as well as the same number of samples taken in the *transition* region are selected. A class label, *normal*, is attached to each sample taken from the *normal* region, and a class label, *pre-congested*, is attached to each sample taken from the *transition* region. The samples are saved in a text file as a set of training data. The raw data is then fed into the *data pre-processing* component (discussed in Section III-B) for its pre-processing and then sent to the *attribute selection* component (discussed in Section III-C) to produce a set of *system-congestion* indicators. The indicators can be used to recognize the system is in the *transition* region and to initiate workload control actions to prevent system congestion.

As introduced in Section I, in a mixed request data server environment, the concurrently running workloads can have different types, business importance, performance objectives, as well as arrival rates, and these properties can dynamically change during runtime. Therefore, it becomes very difficult to discover *system-congestion* indicators by monitoring and analysing the entire system with its complex and dynamic workloads. Thus, in emulating a workload mix environment in this study, we assume there is a workload, such as an OLTP workload, that runs at a constant load (*e.g.*, in a batch execution manner, or more generally, it can be represented with a closed queueing network model [8]). By tracking the workload (as an OLTP workload is often characterized as having high business importance and predictable execution behavior, it becomes sensitive to congestion with any change during its execution), and collecting and analysing its monitoring data, namely analysing the records of the workload execution behavior collected in each sample interval, a set of key monitor metrics can be identified, where the values of the monitor metrics present critical changes when the database system moves into its *transition* region from its *normal* region.

The (chosen) workload can then be monitored by using the discovered metrics to indicate whether or not the database system is on the verge of congestion whenever additional requests are added on it. Thus, based on the assumption, instead of sampling the entire DBMS and all its workloads, only the workload with its relevant database objects, such as buffer pools, need to be monitored and sampled.

B. Data Pre-processing

The DBMS monitoring data consists of a set of system samples. A monitoring data record is comprised of values for a set of monitor metrics, such as the number of data logical reads, the number of data physical reads, or the total amount of CPU time used by the system. In this work, we consider a monitor metric to be an *attribute*, while each row is a record in the collected monitoring data set.

Typically, the values of monitor metrics are the system's accumulated counter values, which are determined by the DBMS's specific environment such as its configurations. As discussed in Section III-A, a workload that constantly runs in a complex workload mix environment is monitored. The main idea of our approach of discovering DBMS congestion indicators is to identify the critical differences in the workload execution behaviours in the *normal* and *transition* region, and then to use a set of key monitor metrics to recognize the differences. Since the accumulated counter values cannot be directly used for discovering the key monitor metrics, we employ a data pre-processing to eliminate the accumulative effects on the collected monitoring data and then normalize the data.

The raw data pre-processing includes four steps, namely, *calculating differences*, *calculating moving averages*, *calculating z-scores* and *discretization*. We consider the data set has $n+1$ attributes, namely ($a_1, ..., a_k, ..., a_n, c_h$), where, $n \in N$, a_k is the *k-th* attribute, ($1 \leq k \leq n$), and c_h is the class label attribute, (for $h = 1,2$, c_1=*normal* and c_2=*pre-congested*), and t samples (rows), where, $t/2$ samples belong to the *normal* class, while the other $t/2$ samples are in the *pre-congested* class, and $t \in 2N$. a_k^i represents the value of the *k-th* attribute in the *i-th* sample, where $1 \leq i \leq t$.

The first step, *calculating differences*, involves finding the difference between corresponding attributes (for each attribute, a_k) in two consecutive records. This step is applied to both the *normal* and *pre-congested* class. By calculating the differences, the accumulative historic effects on the data set are eliminated. Therefore, the number of data records (rows) in the calculated new data record set, *D*, is *t-2*, where *(t-2)/2* records belong to the *normal* class, and the other *(t-2)/2* records belong to the *pre-congested* class.

In order to reduce the amount of variation present in the data set, *D*, *a moving average of order m* is applied, where, $m \in N$ and $1 < m \leq (t-2)/2$. The moving average is also applied to both the *normal* and *pre-congested* record class. Thus, the number of data records in a new set, *V*, is *(t-2)-2m+2*, where *(t-2)/2–m+1* records belong to the *normal* class, and the other *(t-2)/2–m+1* records belong to the *pre-congested* class.

The third step, *calculating z-score normalization* [4], is to normalize the data records in the set, *V*, to generalize the data from its specific experimental environment. In the data record set, *V*, if the *i-th* data record is $(a_1^i, a_2^i, \cdots, a_k^i, \cdots, a_n^i, c_h)$, then its *z-score normalization* can be taken using equation (1):

$$\left(a_k^i\right)' = abs\left(\frac{a_k^i - \overline{A}}{\sigma}\right) \tag{1}$$

for each attribute, a_k. In the equation, $\overline{A}$ and σ are the mean and standard deviation, respectively, of the attribute, a_k, in the normal region, which can be experimentally estimated [4], and *abs* is the absolute function. Thus, a new set, *O*, is calculated.

The fourth step is to *discretize* the data records in the set, *O*. This minimizes possible values of an attribute to simplify the *attribute selection* process (discussed in Section III-C). One approach to discretizing data records is the use of a ceiling function, so a new data set, *W*, can be calculated. In our study, we consider the mapping: [0, 0.5] -> 0.5, (0.5, 1] -> 1, (1, 1.5] -> 1.5, (1.5, 2] -> 2, (2, 2.5] -> 2.5, (2.5, 3] -> 3, (3, 3.5] -> 3.5, and (3.5, +∞) -> 4. Thus, the calculated values are in the set {0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4}, which are used in our experimental result analysis discussed in Section IV.

C. Attribute Selection

The goal of the *attribute selection* process is to identify a set of attributes (*i.e.*, monitor metrics) to act as classifiers (*i.e.*, indicators) so that given a sample with an unknown class label, by testing the monitor metric's values, the class label of the data sample can be determined.

The *information gain* measure [4] is applied to select the attributes. Information gain is widely used in data mining applications in choosing test attributes for building a decision tree. The same idea is used here. Given the pre-processed set, *W*, with *n+1* attributes and *(t-2)-2m+2* records, the significant attributes are chosen iteratively. We calculate the information gain of all the attributes, and the attribute with the highest information gain is chosen as a significant attribute. Then, all the data records in the set, *W*, are partitioned with respect to this chosen attribute. The information gain measure is then repeated in each of the partitions. The algorithm for inducing a decision tree from the set of training data is shown in Figure 2 [4] as follows:

```
Input: training set, W, attributes, ($a_1$, ..., $a_k$, ..., $a_n$)
Output: a decision tree (i.e., a set of key attributes)
Algorithm: Generate_Decision_Tree (W, ($a_1$, ..., $a_k$, ..., $a_n$))
1.  attribute-list ← ($a_1$, ..., $a_k$, ..., $a_n$);
2.  create a node, d;
3.  if samples in W are all of the same class then
4.      return d as a leaf node labelled with the class;
5.  if attribute-list is empty then
6.      return d as a leaf node labelled with the most common
        class in samples; //majority voting
7.  select the attribute, $a_k$, among attribute-list with the highest
    information gain;
8.  label d with $a_k$;
9.  for each known value, $v_j$, of $a_k$ //partition all the samples in W
10.     grow a branch from node d for the condition $a_k = v_j$;
11.     let $S_j$ be the set of samples in W for which $a_k = v_j$; //a
        partition
12.     if $S_j$ is empty then
13.         attach a leaf labelled with the most common class in
            the samples in W;
14.     else attach the node returned by Generate_Decision_Tree
        ($S_j$, (attribute-list - $a_k$));
```

Fig. 2 Algorithm for Generating a Decision Tree [4]

The information gain [4] of the attribute a_k, where, $1 \leq k \leq n$, can be calculated using:

$$Gain(a_k) = I(e_1, e_2) - E(a_k) \quad (2)$$

where, e_1 is the number of records with the class label c_1 in W, and e_2 is the number of records with the class label c_2 in W. Thus, $I(e_1, e_2)$ can be determined by:

$$I(e_1, e_2) = -\sum_{i=1}^{2} p_i \log_2(p_i) \quad (3)$$

where, $p_1 = (e_1/b)$, $p_2 = (e_2/b)$ and b is the number of records in W. $E(a_k)$ can be determined by:

$$E(a_k) = \sum_{j=1}^{u} \frac{x_{j,1}+x_{j,2}}{b} I(x_{j,1}, x_{j,2}) \quad (4)$$

where, the attribute a_k has u distinct values, and $x_{j,1}$ is the number of samples, whose class label is c_1, and value of the attribute, a_k, is the attribute's *j-th* value. $x_{j,2}$ is the number of samples, whose class label is c_2, and value of the attribute, a_k, is the attribute's *j-th* value. $I(x_{j,1}, x_{j,2})$ can be similarly determined using equation (3).

IV. EXPERIMENTS

In this section we present a set of experiments that illustrate two different cases of congestion in a database system and examine the effectiveness of our approach of discovering DBMS congestion indicators. Section IV-A describes the experimental environment. Section IV-B and IV-C present the two different cases of congestion, and Section IV-D discusses the experimental results.

A. Experimental Environment

All experiments were conducted with DB2 (Version 9.7) database software [6] running on an Acer® server with the Windows® Server 2008 operating system. The data server was equipped with four Intel® Core™ 2 Quad processors, 8GB of RAM, and two disks configured with one disk controller. Three DB2 clients running on three PCs sent multiple types of requests to the data server to build a server-client environment. TPC benchmark workloads and databases [18] were used in the experiments. One client issued TPC-C transactions and the other two clients generated TPC-H queries and other requests.

Congestion in database systems was presented on the DB2 data server, by adding additional work to the data server, until it moved from its *normal* region to its *transition* region and then to its *congested* region, while we sampled the system in all the three regions to collect a set of monitoring data.

In the experiments, we considered a TPC-C workload as the *constant load* workload (as discussed in Section III-A) in the DB2 data server. Thus, the workload and its related objects (*bufferpool*, *tablespace*, *container*, and *service class*) in the database system were sampled using the DB2 monitoring (table) functions [6] to access in memory counter values of monitor metrics, where around seven-hundred metrics were accessed. After collecting the monitoring data, each sample was attached a class label as discussed in Section III-A, and the data mining technique was then used to identify the most important monitor metrics to indicate the system congestion.

B. Disk I/O Contention

The data server was run in its *normal* region using the TPC-C workload issued by 100 connections. The workload was run for 900 seconds, at which point 200 TPC-H queries arrived at the system, namely introducing bursts of *ad hoc* queries to the system, and, thus, moving the system into its *transition* region then into its *congested* region. These two workloads were run concurrently for another 900 seconds. To isolate the data for the two workloads, two buffer pools were created; one 5GB buffer pool for the TPC-C workload and one 1GB buffer pool for the TPC-H workload.

To reduce the amount of disk I/O accesses generated by the TPC-C workload, the *read-only* TPC-C transactions were used in the experiment, namely *order-status* and *stock-level* [18], and the mix rate of the transactions in the workload was 50% for each. The TPC-C and TPC-H data and index tables resided on a single disk. The sizes of the TPC-C and TPC-H databases were 2GB and 6GB, respectively.

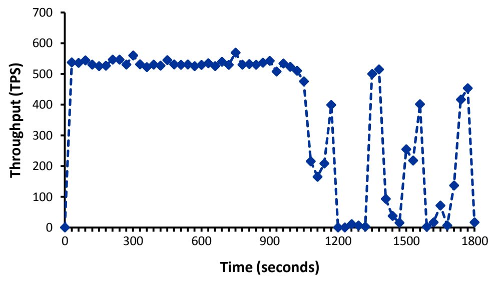

Fig. 3 Throughput of the System in the Case of Disk I/O Contention

Figure 3 shows the system throughput as the two workloads were executed. When the transactional workload was running alone, the system throughput was high and stable, however, once the TPC-H queries began to execute, the throughput of the system became low and unstable, fluctuating rapidly between low and high levels. This severe degradation in the performance did not mean *thrashing* occurred in the system as there was neither memory contention (as two separate buffer pools were used) nor data contention (as two separate data sets were applied) caused by the execution of the TPC-H workload. The bottleneck resource in the system was disk I/O since all data resided on a single disk and only one disk controller was used. The highly I/O intensive TPC-H queries competed with the TPC-C transactions for the I/O resources, occupying most of the disk I/O bandwidth in its execution and thus, impacting the TPC-C transactions significantly and resulting in severe degradation in system performance. (Although the buffer pool was large enough to hold all the TPC-C data, the TPC-C workload generated requests to access randomly selected data records in the TPC-C data set instead of scanning the entire tables).

By observing the experimental results in Figure 3, we see that they illustrate the database system's state change, namely, moving from the *normal* region to the *transition* region and then to the *congested* region as the workloads ran on the system. The system was in its *normal* region when the TPC-C workload was running alone, and it was in its *transition* region when the bursts of TPC-H queries began until the first sudden and sharp drop of its throughput, and then it was in its *congested* region when the throughput was critically low and unstable.

By selecting the monitoring data collected in the *normal* and *transition* region, respectively, and applying the approach introduced in Section III a set of *system-congestion* indicators is discovered for the case of disk I/O contention, as shown in Figure 4.

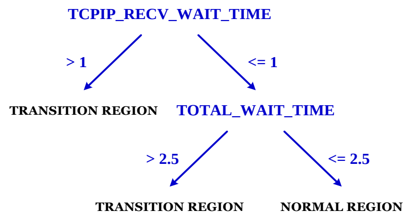

Fig. 4 Indicators for the Case of Disk I/O Contention

Figure 4 shows a *pruned* decision tree [4]. It means that if the "TCPIP_RECV_WAIT_TIME", a monitor metric (further discussed in Section IV-D), pre-processed value of a data sample with an unknown class label is greater than 1, then its class label is *pre-congested*, namely, the DBMS is currently running in its *transition* region. Otherwise, we examine the "TOTAL_WAIT_TIME" attribute, a monitor metric (further discussed in Section IV-D). If its pre-processed value is less than or equal to 2.5, then the data sample belongs to the *normal* class, namely, the DBMS is currently running in its *normal* region, otherwise, it is *pre-congested*.

C. Shared Data Contention

In this experiment we used an 8GB TPC-C database running the TPC-C workload (with all 5 types of transactions [18]) along with a database application designed to introduce lock contention. The application, called "DB-Lock", attempts to lock all the TPC-C data tables for updated (exclusive locks). The data server was run in its *normal* region using the TPC-C workload issued by 100 connections. The workload was run for 900 seconds, at which point DB-Lock began, moving the system into the *transition* region and eventually to its *congested* region. The two types of requests then ran concurrently for another 900 seconds.

Figure 5 shows the system throughput as the two types of requests were executed. When the transactional workload was running alone, the system throughput was high and stable, however, once the DB-Lock began to execute, the system throughput decreased significantly. DB-Lock competed with the TPC-C transactions for locks of the TPC-C data set since the application tried to lock all the tables to update their contents. DB-Lock held the locks for a (long) period of time thus making requests from TPC-C wait for the locks available, therefore resulting in severe degradation in overall system performance. This experiment illustrates the state change of the database system, namely, moving from its *normal* region to *transition* region and then to its *congested* region The system was in its *normal* region when the TPC-C workload was running alone, and it was in its *transition* region when DB-Lock began to execute until it grabbed all the locks of the TPC-C data tables, and then the system was in its *congested* region when the throughput totally declined.

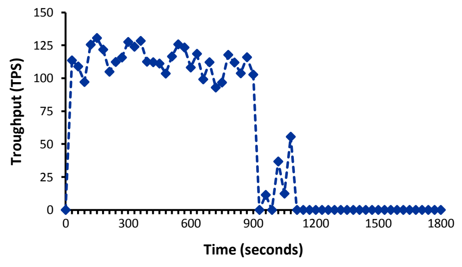

Fig. 5 Throughput of the System in the Case of Shared Data Contention

By selecting the monitoring data collected in the *normal* and *transition* region, respectively, and using the approach introduced in Section III a set of *system-congestion* indicators is discovered for the case of shared data contention, as shown in Figure 6.

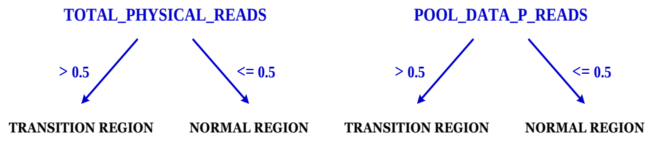

Fig. 6 Indicators for the Case of Shared Data Contention

Figure 6 shows two *pruned* decision trees [4]. It presents two separate sets of *system-congestion* indicators (when using the collected training data to build a decision tree, we found that there were two attributes that could perfectly divide all samples into their own classes). Either can potentially act as a classifier to classify the data server's state. The set presented on the left indicates that if the "TOTAL_PHYSICAL_READS", a monitor metric (further discussed in Section IV-D), pre-processed value of a data sample is greater than 0.5, the system is currently running in its *transition* region, otherwise, it is in its *normal* region. Similarly, the right one shows that if the "POOL_DATA_P_READS", a monitor metric (further discussed in Section IV-D), pre-processed value of a data sample is greater than 0.5 then the data server is currently running in its *transition* region, otherwise, it is *normal*.

D. Discussion

In these two cases, cross validation is applied to validate the classification accuracy of the discovered indicators (Weka [19], a suit of machine learning software, was used for generating and pruning the decision trees, as well as doing the cross validation). Our results of cross validation show that the correctly classified instances are greater than 96% for both cases. The indicators can also be validated with test data sets, and if the accuracy is lower than a required level, they need to be rediscovered with new training data.

Through observing the discovered indicators in these two cases, we note that the indicators can vary depending upon the source of congestion, and there may not be a set of generic indicators for all potential sources. For a given DBMS-workload scenario, to discover a set of indicators for the DBMS, the first phase is to identify a workload with high business importance and (stable) execution behavior. We let the workload run on the system, and adding additional loads on the DBMS till it is congested. By collecting monitoring data in the *normal* and *transition* regions, and applying our approaches, a set of indicators can then be discovered and used to protect the performance of the DBMS.

The effectiveness of our approach is examined through the experiments. In the two different cases, the TPC-C workload execution behaviour was captured by the indicators. In the first case, the monitor metrics, "TCPIP_RECV_WAIT_TIME" and "TOTAL_WAIT_TIME", became significant as in the case the TPC-C workload generated less disk access requirements when the database system moved into its *transition* region from its *normal* region. "TCPIP_RECV_WAIT_TIME" says that the system spent more time waiting for an incoming client request over a TCP/IP connection than what the system did in its *normal* region. This was because current requests were stuck in the system, and new requests could not enter the system before the others completed (the workload was the *constant load*, so it was executed in a batch execution manner, as discussed in Section III-A). "TOTAL_WAIT_TIME" indicates that a transactional request spent longer waiting within the data server. This monitor metric further described the congestion situation. In the second case, the monitor metrics show that in a *transition* region, the TPC-C workload's execution rate in the data server became slow, and therefore the workload could not generate as many physical reads as it did in its *normal* region.

V. Conclusions

In this paper we present an approach to discovering DBMS congestion indicators. The indicators are a set of database monitor metrics to indicate impending congestion of a database system. Through a set of experiments, we validated the effectiveness of our approach and also showed that the indicators can vary depending upon the source of system congestion.

In the future, we plan to use system modelling approaches, such as queuing theory models or predictive analytics, to model a DBMS with its workloads to predict system performance problems.

Acknowledgment

This research is supported by IBM Centre for Advanced Studies (CAS) Canada Research and Natural Science and Engineering Research Council (NSERC) of Canada.

Trademarks

IBM, DB2 and DB2 Universal Database are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Disclaimer

The views expressed in this paper are those of the authors and not necessarily of IBM Canada Ltd. or IBM Corporation.

References

[1] D. P. Brown, A. Richards, R. Zeehandelaar and D. Galeazzi, "Teradata Active System Management: High-Level Architecture Overview". [Online]. Available: http://www.teradata.com/white-papers/Teradata-Active-System-Management-High-Level-Architectural-Overview-eb4685/?type=WP.

[2] S. Chaudhuri and V. Narasayya. "Self-Tuning Database Systems: A Decade of Progress". In *Proc. of VLDB'07*, Austria, pp. 3-14.

[3] P. J. Denning. "Thrashing: Its Causes and Prevention". In *Proc. of the AFIPS Joint Computer Conf.*, San Francisco, CA, USA, December 9-11, 1968, pp. 915-922.

[4] J. Han and M. Kamber. "Data Mining: Concepts and Techniques" *Morgan Kaufmann Publishers*, San Francisco, CA, USA, 2001.

[5] IBM Corp., "Autonomic Computing: IBM's Perspective on the State of Information Technology". [Online]. Available: http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.

[6] IBM Corp., "IBM DB2 Database for Linux, UNIX, and Windows Information Center". [Online]. Available: https://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp.

[7] S. Krompass, H. Kuno, J. L. Wiener, K. Wilkison, U. Dayal and A. Kemper, "Managing Long-Running Queries", In *Proc. of EDBT'09*, Saint Petersburg, Russia, pp. 132-143.

[8] E. Lazowska, J. Zahorjan, G. S. Graham and K. C. Sevcik "Quantitative System Performance: Computer System Analysis Using Queueing Network Models", *Prentice-Hall Inc.*, Englewood Cliffs, New Jersey, USA, 1984.

[9] S. Lightstone, G. Lohman and D. C. Zilio, "Toward Autonomic Computing with DB2 Universal Databases". *ACM SIGMOD Record, Volume 31 Issue 3*, September 2002.

[10] Microsoft Corp., "Managing SQL Server Workloads with Resource Governor". [Online]. Available: http://msdn.microsoft.com/en-us/library/bb933866.aspx.

[11] A. Mehta, C. Gupta and U. Dayal, "BI Batch Manager: A System for Managing Batch Workloads on Enterprise Data-Warehouses", In *Proc. of EDBT'08*. Nantes, France, pp. 640-651.

[12] A. Moenkeberg and G. Weikum, "Performance Evaluation of an Adaptive and Robust Load Control Method for the Avoidance of Data Contention Thrashing". In *Proc of VLDB'92*, Vancouver, BC, Canada, pp. 432- 443.

[13] B. Niu, P. Martin and W. Powley, "Towards Autonomic Workload Management in DBMSs". In *Journal of Database Management*, 20(3), 1-17, July-September 2009.

[14] Oracle Corp., "Oracle Database Resource Manager", [Online]. Available:download.oracle.com/docs/cd/B28359_01/server.111/b28310/dbrm.htm#i1010776.

[15] S. Parekh, K. Rose, J. Hellerstein, S. Lightstone, M. Huras and V. Chang. "Managing the Performance Impact of Administrative Utilities". In *Proc. of Self-Managing Distributed Systems*, Springer Berlin, Heidelberg, February 19, 2004. pp. 130-142.

[16] W. Powley, P. Martin, M. Zhang, P. Bird and K. McDonald. "Autonomic Workload Execution Control Using Throttling". In *Proc. of ICDE'10 Workshops (5th SMDB)*, Long Beach, CA, USA.

[17] Teradata Corp., "Teradata Dynamic Workload Manager", [Online]. Available: www.info.teradata.com/edownload.cfm?itemid=082330034.

[18] Transaction Processing Performance Council. [Online]. Available: http://www.tpc.org.

[19] Weka. [Online]. Available: http://www.cs.waikato.ac.nz/ml/weka/