# Utility Functions in Autonomic Workload Management for DBMSs

Mingyi Zhang[†], Baoning Niu[§], Patrick Martin[†], Wendy Powley[†], Paul Bird[‡]
[†]School of Computing, Queen's University, Kingston, ON, Canada
{myzhang, martin, wendy}@cs.queensu.ca
[§]Taiyuan University of Technology, Shanxi, China
niubaoning@tytu.edu.cn
[‡]Toronto Software Lab, IBM Canada Ltd., Markham, ON, Canada
pbird@ca.ibm.com

***Abstract*—Utility functions are a popular tool for achieving self-optimization in autonomic computing systems. Utility functions are used to guide a system in optimizing its own behavior in accordance with high-level objectives specified by the system administrators. It is, however, difficult to define a new utility function or evaluate whether an existing utility function is appropriate for a specific system management scenario. In this paper, we discuss the fundamental properties of an effective utility function for autonomic workload management in database management systems (DBMSs). We present two concrete examples of utility functions to illustrate the properties. The utility functions are used for dynamic resource allocation and for query scheduling in DBMSs. The utility functions help the systems translate high-level workload management policies into low-level tuning actions, and therefore ensure the workloads achieve their required performance objectives. A set of experiments are presented to illustrate the effectiveness of the two example utility functions.**

***Keywords-Self-Optimization; Utility Function; Autonomic Computing; Workload Management; Database Management Systems***

## 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 objectives (*e.g.*, response time or throughput objectives) [2]. Workload management in database management systems (DBMSs) is a performance management process. The primary objectives of workload management in DBMSs are to achieve the performance goals of all workloads (particularly, the critical ones, such as the workloads for directly generating revenue for business organizations, or those issued by a CEO or VP of the organizations), maintain DBMSs running in an optimal state (*i.e.*, neither under-utilized nor overloaded), and balance resource demands of all requests to maximize performance of the entire system.

For both strategic and financial reasons, many business organizations are consolidating individual data servers onto a single shared data server. As a result, multiple types of requests are present on the data server simultaneously. Request types can include on-line transaction processing (OLTP) and business intelligence (BI). OLTP transactions are typically short and efficient, consume minimal system resources, and complete in sub-seconds while BI queries tend to be more complex and resource-intensive and may require hours to complete. Requests generated by different applications or initiated from different business units may have unique performance objectives that are normally expressed in terms of service level agreements that must be satisfied for business success.

Multiple requests running on a data server inevitably compete for shared system resources, such as system 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 consume a large amount of system resources without control, the concurrently running requests may have to wait for the long queries to complete and release their used resources, thereby resulting in the waiting requests missing their performance objectives and the entire data server suffering degradation in performance. Moreover, the mix of arriving requests present on a data server can vary dynamically and rapidly, so it becomes virtually impossible for database administrators to manually adjust the system configurations to dynamically achieve performance objectives of all the requests during runtime. Therefore, a*utonomic workload management* becomes necessary and critical to control the flow of the requests and manage their demands on system resources to achieve their required performance objectives in a complex request mix environment.

Since *autonomic computing* was introduced [3], a great deal of effort has been put forth by researchers and engineers in both academia and industry to build autonomic computing systems. An autonomic computing system is a self-managing system that manages its own behavior in accordance with high-level objectives specified by human administrators [3] [4]. Such systems regulate and maintain themselves without human intervention to reduce the complexity of system management and dynamically achieve system objectives, such as performance, availability and security objectives. In particular, an autonomic workload management system for DBMSs is a self-managing system that dynamically manages workloads present on a data server in accordance with specified high-level business objectives such as workload business importance policies.

Achieving the goal of autonomic workload management may involve using utility functions to facilitate the mapping

of high-level business objectives to low-level DBMS tuning actions in order to guide a database system to optimize its own behavior and achieve required performance objectives. Utility functions are well known as a measure of user preference in economics and artificial intelligence [5]. In this paper, we illustrate the use of utility functions in different aspects of database workload management, namely dynamic resource allocation and query scheduling, to ensure mixed-type requests on a data server achieve their required performance objectives. The contribution of this study is a set of fundamental properties of a utility function used for building autonomic workload management systems, and the use of the properties to evaluate whether an existing utility function is appropriate for autonomic workload management in DBMSs. The methods and properties were first presented in our (ICAS'11) paper [1] and have been elaborated upon and extended with experimental validation here.

The paper is organized as follows. Section II reviews the background and related work, in which a short review of workload management for DBMSs, a brief description of autonomic computing, and utility functions used for building autonomic computing systems are presented. Section III discusses the fundamental properties of a utility function that can be used in realizing autonomic workload management for DBMSs. Section IV provides two examples to illustrate the properties of two different types of utility functions that are used in our studies. Section V presents experiments to evaluate and compare the two utility functions in accordance with some given high-level workload business importance policies. Finally, we conclude our work and propose future research in Section VI.

## II. BACKGROUND AND RELATED WORK

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], Teradata® Active System Management [7], Microsoft® SQL Server Resource and Query Governor [8] [9] and Oracle® Database Resource Manager [10]. The workload management facilities manage complex workloads (*e.g.*, a mix of business processing and analysis requests) present on a data server using predefined procedures. The procedures impose proper controls on the requests, based on the request's characteristics such as estimate costs, resource demands, or execution time, to achieve their required performance objectives.

Recent research [11] [12] shows that the process of workload management in DBMSs may involve three typical controls, namely admission, scheduling, and execution control. 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. Request scheduling determines the execution order of admitted requests based on some criteria, such as the request's level of business importance and/or performance objectives. Execution control dynamically manages some running requests to limit their impact on other concurrently running queries. In this paper, we demonstrate our techniques used for workload management in DBMSs.

In 2001, IBM presented the concept of autonomic computing [3]. The initiative aims to provide the foundation for computing systems to manage themselves according to high-level objectives, without direct human intervention in order to reduce the burden on the system administrators. An autonomic computing system (*i.e.*, a self-managing system) has four fundamental properties, namely *self-configuring*, *self-optimizing*, *self-protecting* and *self-healing*. Self-configuring means that a system is able to configure itself automatically to allow the addition and removal of system components or resources without system service disruptions. Self-optimizing means that a system automatically monitors and controls its resources to ensure optimal functioning with respect to the specified performance goals. Self-protecting means that a system is able to proactively identify and protect itself from arbitrary attacks. Self-healing means that a system is able to recognize and diagnose deviations from normal conditions and take action to normalize them [3] [4].

In the past decade, autonomic computing has been intensively studied. Many autonomic computing components (with some self-managing capabilities) have been developed and proven to be useful in their own right, although a large-scale fully autonomic computing system has not yet been realized [13] [14]. In particular, Tesauro *et al.* [15] and Walsh *et al.* [16] studied autonomic resource allocation among multiple applications based on optimizing the sum of the utilities for each application. In their work, a data center consisting of multiple and logically separated application environments (AEs) was used. Each AE provided a distinct application service using a dedicated, but dynamically allocated, pool of servers, and each AE had its own service-level utility function specifying the utility to the data center from the environment as a function of some service metrics. The authors compared two methodologies, a queuing-theoretic performance model and model-free reinforcement learning, for estimating the utility of resources.

Bennani *et al.* [17] presented another approach for the same resource allocation problems in the autonomic data center. They observe that the table-driven approach proposed by Walsh *et al.* [16] has scalability limitations with respect to the number of transaction classes in an AE, the number of AEs, and the number of resources and resource types. Moreover, they claim that building a table from experimental data is time consuming and has to be repeated if resources are replaced within the data center. They instead proposed using predictive multi-class queuing network models to implement the service-level utility functions for each AE. In this paper, we show the principles of autonomic computing applied in workload management for DBMSs, and applications of utility functions in building autonomic workload management systems.

## III. UTILITY FUNCTIONS IN WORKLOAD MANAGEMENT

Achieving autonomic workload management for DBMSs can involve the use of utility functions. In this section, we consider the following questions:

- Why are utility functions appropriate for autonomic workload management?
- What utility functions are most suitable (*i.e.*, what properties does a utility function need to possess) for autonomic workload management?

The first question can be answered based on the research of Kephart *et al.* [5] and Walsh *et al.* [16], who proposed the use of utility functions to achieve self-managing systems. In their work, the authors presented utility functions as a general, principled and pragmatic way of representing and managing high-level objectives to guide the behavior of an autonomic computing system. Two types of policies were discussed in guiding behavior of a system, namely action policies and goal policies. An action policy is a low-level policy that is represented in the form of *IF (conditions) THEN (actions*). Namely, if some conditions are satisfied, then certain actions must be taken by the system. In contrast with an action policy, a goal policy only expresses high-level objectives of a system, and the system translates the high-level objectives into specific actions for every possible condition. Utility functions are proposed for the translation as they are capable of mapping system states to real numbers with the largest number representing a system's preferred state. In using utility functions, a computing system, via maximizing its utilities under each condition, recognizes what the goal states are, and then decides what actions it needs to take in order to reach those states. Thus by maximizing utilities, a computing system optimizes its own behavior and achieves the specified high-level objectives.

As introduced in Section I, in a mixed request data server environment, the concurrently running requests can have different types, levels of business importance, performance objectives and arrival rates. These properties may dynamically change during runtime rendering it impossible for human administrators to manually make an optimal resource allocation plan for all workloads in order to meet their resource requirements. A utility function, however, is suited for this situation, based on the properties discussed above. It dynamically identifies resource preferences for a workload during runtime, and the utility functions of the workloads can be further used to define an objective function. A solution to optimizing the objective function is an optimal resource allocation plan. Autonomic workload management systems use the resource allocation plan to allocate resources to the workloads and to achieve the required performance objectives. Thus, to manage workloads in DBMSs, using utility functions is naturally a good choice.

To answer the second question, we begin by discussing performance behavior of a workload. The performance of a running workload on a data server depends on the amount of desired system resources that the workload can access. Typically, the performance of a workload increases non-linearly with additional resources assigned to it. As an example, in executing a workload in an OLTP system, by increasing the multi-programming levels, the throughput of the workload initially increases, but at a certain point the throughput starts to level off. That is, at the beginning when the workload starts to run with a certain amount of resource allocated, performance of the workload increases rapidly. However, with additional resources allocated to the workload, the performance increment of the workload becomes very small. This can be caused either by a bottleneck resource among the system resources, such as too small buffer pools, which significantly limits the workload performance increase, or it may be the case that the database system has become saturated (*e.g.*, system CPU resource is fully utilized).

Utility functions in database workload management must capture the performance characteristics of a workload and represent the trend of the changes in performance based on the amount of assigned resources. A utility function defined for database workload management should be a monotonically non-decreasing function, and it should be capable of mapping the performance achieved by a workload with a certain amount of allocated resources into a real number, $u$.

There is no single way to define a utility function. However, we believe the following properties are necessary for an effective utility function in autonomic workload management for DBMSs:

- The value, $u$, should follow the performance of a workload. Namely, it should increase or decrease with the performance.
- The amount of change in the utility should be proportional to the change in the performance of a workload.
- The input to a utility function should be the amount of resources allocated to a workload, or a function of the resource allocation, and the output, $u$, should be a real number without unit.
- The value, $u$, should not increase (significantly) as more resources are allocated to a workload, once the workload has reached its performance objective.
- In allocating multiple resources to a workload, a utility function should capture the impact of the allocation of a critical resource on performance of the workload.
- For objective function optimization, a utility function should have good mathematical properties, such as an existing second derivative.

The first two properties describe the general performance behavior of a workload that a utility function needs to capture, and the third property presents the domain and codomain of a defined utility function. These three properties are fundamental for a utility function that can be used in building autonomic workload management in DBMSs. The fourth and fifth properties represent the relationships among workload performance, resource provisions, and performance objectives. Namely, if a workload has met its required performance objective, the value produced by the utility function would not increase (significantly) as additional resources are allocated to the workload. So, by checking the marginal utility (the value is very small), the database system can know it should stop allocating additional resources to the workload. If there is a critical resource for a workload, then the utility function

should reflect the impact of changes to the allocation of that resource. The database system then knows to provide the resource to the workload for meeting its performance objective. The last property provides a way of effectively optimizing objective functions.

## IV. UTILITY FUNCTION EXAMPLES

Two examples from our work of the use of utility functions in autonomic workload management for DBMSs are presented in this section. The first example demonstrates Dynamic Resource Allocation, which is driven by workload business importance policies [18]. The second example shows a Query Scheduler managing the execution order of multiple classes of queries [19]. The two utility functions are discussed with respect to the properties listed in Section III.

### A. Dynamic Resource Allocation

In workload management for DBMSs, dynamic resource allocation can be triggered by workload reprioritization (a workload execution control approach) [6] [18]. That means a workload's priority may be dynamically adjusted as it runs, thereby resulting in immediate resource reallocation to the workload according to the new priority.

Two shared system resources are considered in the study, namely buffer pool memory pages and CPU shares, as they are key factors in DBMS performance management. The DBMS concurrently runs multiple workloads, which are classified in different business importance classes with unique performance objectives. A certain amount of the shared resources is allocated to a workload according to its business importance level. High importance workloads are assigned more resources, while low importance workloads are assigned fewer. The resource allocation is based on an *economic model* [18]. Namely, the DBMS conducts "auctions" to sell the shared system resources, and the workloads submit "bids" to buy the resources via an auctioning and biding based trade mechanism. All the workloads are assigned some virtual "wealth" to reflect their business importance levels. High importance workloads are assigned more wealth than low importance ones.

The dynamic resource allocation approach consists of three main components, namely the *resource model*, the *resource allocation method* and the *performance model*. The *resource model* is used to partition the resources and to determine an available total amount of the resources for allocation. We consider that each competing workload is assigned its own buffer pool, so buffer pool memory pages can be directly assigned to a workload. The CPU resources, on the other hand, cannot be directly assigned to a workload, so we partition CPU resources by controlling the number of database agents that are available to serve requests on a database server. In our study, we use a DB2 DBMS and configure it such that one database agent maintains one client connection request from the workloads. We conducted experiments and verified the relationship between the number of database agents and system CPU utilization of a workload, and observed that the more database agents that are allocated to serve requests for a particular workload, the more CPU resources the workload receives [18]. The available total amounts of resources are parameters in the resource allocation approach, so it can adapt to different system configurations.

The *resource allocation method* determines how to obtain an optimal resource pair of buffer pool memory pages and CPU shares for a workload in order to maximally benefit the workload performance. Namely, a workload needs to capture the resources in an appropriate amount such that none of the resources become a bottleneck resource. In our approach, a greedy algorithm is used for identifying resource preferences of a workload in a resource allocation process. The resource allocation is determined iteratively. In an iteration of the algorithm, by using its virtual *wealth*, a workload bids for a unit of the resource (either buffer pool memory or CPU) that it predicts will yield the greatest benefit to its performance. Figure 1 shows a representation of the search state space for the allocation of buffer pool memory and CPU to a workload in our experiments, as described in Section V. The starting node, $n_{1,1}$, represents the minimum resource allocation to a workload, namely one unit of buffer pool memory and one unit of CPU, at the beginning of a resource allocation process. The workload then traverses the directed weighted graph to search for the optimal <*cpu*, *mem*> pair in order to achieve its performance objective.

The *performance model* predicts the performance of a workload with certain amount of allocated resources in order to determine the benefit of the resources. In our approach, queuing network models (QNM) [20] are used to predict performance of a workload at each step of the algorithm, that is, to assign the weights to the edges of the graph in Figure 1. We consider OLTP workloads and use throughput as the performance metric to represent the performance required and achieved by the workloads. We model the DBMS used in our experiments for each workload with a single-class closed QNM, which consists of a CPU service center and an I/O service center. The CPU service center represents the

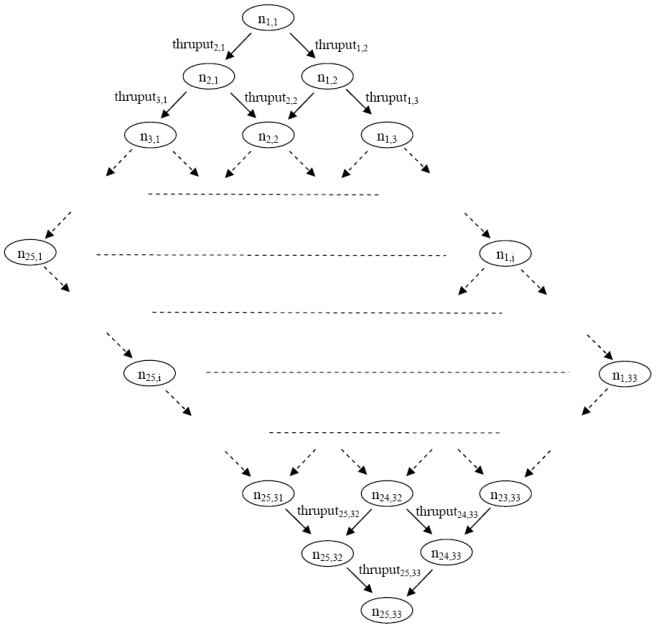


Figure 1. Resource Pair Search State Space

system CPU resources and the I/O service center represents buffer pool and disk I/O resources. The request concurrency level of a workload in the DBMS is the number of database agents (*i.e.*, CPU resources) assigned to the workload. The average CPU service demand of requests in the workload can be expressed as a function of the CPU shares allocated to the workload, using equation (1).

$$S_{CPU} = 1/(a * n + d) \quad (1)$$

We experimentally defined the relationship between the CPU service demand and the number of database agents used in a DBMS. In the equation (1), $n$ is database agents, $n \in \mathbf{N}$, and $a$ and $d$ are constants, $a \in \mathbf{R}^+$, $d \in \mathbf{R}^+$, that can be determined through experimentation.

For an OLTP workload, the average I/O service demand can be expressed as a function of buffer pool memory size, which can be derived from *Belady's* equation [21]. The I/O service demand is:

$$S_{IO} = c * m^b \quad (2)$$

where $c$ and $b$ are constants, $c \in \mathbf{R}^+$, $b \in \mathbf{R}^-$, and $m$ is buffer pool memory pages assigned to the workload, and $m \in \mathbf{N}$. In the equation the constants $c$ and $b$ can be determined through experimentation.

Performance of a workload with some allocated resources, <*cpu, mem*>, can be predicted by solving this analytical performance model (*i.e.*, the QNM) with Mean Value Analysis (MVA) [20]. The predicted throughput of a workload can be expressed as a function of its allocated resources, using equation (3).

$$X = MVA(n, S_{CPU}(cpu), S_{IO}(mem), Z) \quad (3)$$

where, $X$ is the predicted throughput of a workload by using MVA on the QNM for a workload with its allocated resource pair, <*cpu, mem*>; $n$ is the number of requests from the workload concurrently running in the system (*i.e.*, the number of database agents assigned to the workload); $S_{CPU}(cpu)$ is the average CPU service demand determined in equation (1); $S_{IO}(mem)$ is the average I/O service demand determined in equation (2); and $Z$ is think time.

To guide workloads to capture appropriate resource pairs, utility functions are employed in the approach. We define a utility function that normalizes the predicted throughput from the performance model relative to the maximum throughput that the workload could achieve when all the resources are allocated to it. The utility function is given by:

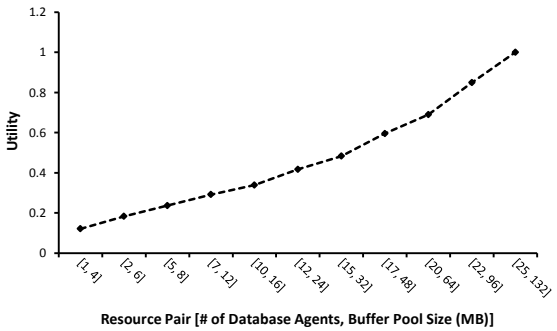


Figure 2. Sample Curve of Utility Function in Resource Allocation

$$u = MVA_{throughput}(n, S_{CPU}(cpu), S_{IO}(mem), Z)/X_{max} \quad (4)$$

where, $MVA_{throughput}(n, S_{CPU}(cpu), S_{IO}(mem), Z)$ is the predicted throughput determined in equation (3), and $X_{max}$ is the maximum throughput achieved by a workload with all the resources allocated, which can be determined through experimentation.

This utility function, as shown in Figure 2, maps performance achieved by a workload given a certain amount of resources into a real number $u$, $u \in [0, \ldots, 1]$. If the utility of resources allocated to a workload is close to 1, it means the performance of the workload is high, while if the utility of resources allocated to a workload is close to 0, it means the performance of the workload is low. Workloads employ the utility function to calculate marginal utilities, that is, the difference in utilities between two possible consecutive resource allocations in a resource allocation process. As the utility function is non-decreasing, the value of a marginal utility is also in the range [0, …, 1].

The marginal utility reflects potential performance improvement of a workload. For some resources, if the calculated marginal utility of a workload is close to 1, then it means these additional resources can significantly benefit the workload's performance, while if the calculated marginal utility is close to 0, then the additional resources will not greatly improve the workload's performance. By examining the marginal utility value, a workload can determine the preferred resources for bid. The bid of a workload is the marginal utility multiplied by current available wealth of the workload, and indicates that a workload is willing to spend the marginal-utility percentage of its current wealth as a bid to purchase the resources. Wealthy workloads, therefore, can acquire more resources in the resource allocation processes. A workload ceases bidding for additional resources when it has reached its performance objective.

### *B. Query Scheduling*

Our *query scheduler* [19] is built on a DB2 DBMS and employs DB2 Query Patroller (DB2 QP) [6] (a query management facility) to intercept newly arriving queries. Information about the queries is then acquired, and the scheduler determines an execution order for the queries. The *query scheduler* works in two main processes, namely the *workload detection* and the *workload control*. The workload detection process classifies arriving queries based on their service level objectives (SLOs), and the workload control process periodically generates new plans to respond to the changes in the SLOs of arriving requests.

In the query scheduler's architecture shown in Figure 3, DB2 QP is set to inform the *query scheduler*'s *monitor* when an arriving query has been intercepted. The *monitor* collects information about the query from the DB2 QP control tables, which includes query identification, query costs and query execution information, and passes the query's information to the *classifier* and the *scheduling planner*. The *classifier* assigns the query to a service class based on its performance goals and puts the query in a queue, which is associated with the service class and managed by the *dispatcher*. The *dispatcher* receives a

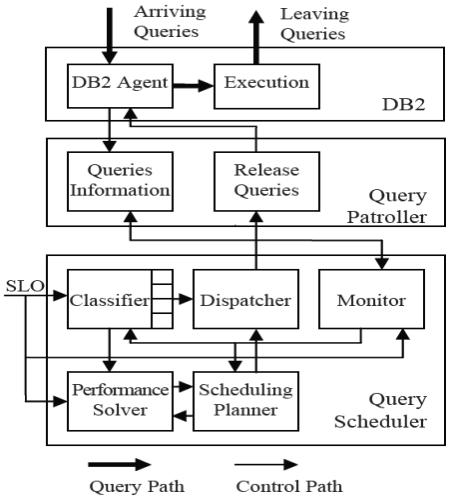


Figure 3. Architecture of Query Scheduler

scheduling plan from the *scheduling planner* and releases the queries in the queues according to the plan's specifications. The *scheduling planner*, given SLOs, receives query information from the *monitor*, and consults the *performance solver* to make a scheduling plan for all the queued queries.

We consider a system with *n* service classes for arriving requests, each with a performance goal and a level of business importance, denoted as $\langle \overline{g}_i, m_i \rangle$, where $\overline{g}_i$ is the performance goal of the *i-th* service class, and $m_i$ is the class business importance level. The pair $\langle \overline{g}_i, m_i \rangle$ is a service level objective. We denote $g_1, g_2, \cdots, g_n$ as the predicted performance of the *n* service classes given a resource allocation plan $r_1, r_2, \cdots, r_n$ (*i.e.*, multi-programming levels in our case). The performance of the *i-th* service class, $g_i$, can be predicted by using a performance model (queuing network models [20] are used in our study) given $r_i$, the amount of resources allocated to the service class. The utility of the *i-th* service class, $u_i$, can be expressed as a function of $\overline{g}_i$, $m_i$ and $g_i$, namely $u_i = f_i(\overline{g}_i, m_i, g_i)$, and the *n* SLOs can be encapsulated into an objective function $f(u_1, u_2, \cdots, u_n)$. Thus, the scheduling problem can be solved by optimizing the objective function *f*.

We specifically consider business analysis requests, such as those found in decision support systems. In emulating the environment, we use the TPC-H benchmark [22] as the database and workloads in our experiments. Since queries in decision support systems can widely vary in their response times, we employ the performance metric *query execution velocity*, which is the ratio of expected execution time of a query to the actual time the query spent in the system (*i.e.*, the total time of execution and delay), to represent the performance required and achieved by the queries. Query execution velocity captures both the performance goals and the business importance levels of queries.

Through our experiments we found the following general form of utility functions satisfies our requirements:

$$u = 1 - \mathrm{e}^{\mathrm{am}\frac{\overline{g}-g}{g-\hat{g}}} \quad (5)$$

where, $\overline{g}$ is the performance goal of a service class to be achieved, $m$ is the importance level of the service class, $m \in \mathbf{N}$, $\hat{g}$ is the lowest performance allowed for the service class, $g$ is the actual performance, and $a$ is an importance factor that is a constant, $a \in \mathbf{N}$, and can be experimentally determined or adjusted to reflect the distance between two adjacent importance levels. In using $a$, we control the size and shape of the utility function, as shown in Figure 4.

The objective function, *f*, is then defined as a sum of the service class utility functions, using equation (6):

$$f = \sum_{i=1}^{n} u_i \quad (6)$$

In *query scheduler*, the *performance solver* employs a performance model to predict query execution velocity for a service class. That is, given a new value of service class cost limit, the performance of the service class can be predicted for the next control interval, which is based on its performance and service class cost limit at the current control interval. The performance at the next control interval is predicted by:

$$V_i^k = \begin{cases} V_i^{k-1} C_i^k / C_i^{k-1} & if\ V_i^{k-1} C_i^k / C_i^{k-1} \le 1 \\ 1 & if\ V_i^{k-1} C_i^k / C_i^{k-1} > 1 \end{cases} \quad (7)$$

where, $V_i^{k-1}$ and $V_i^k$ are query execution velocity of service class *i* at *(k-1)-th* and *k-th* control intervals, respectively; $C_i^{k-1}$ and $C_i^k$ are cost limits of service class *i* at the *(k-1)-th* and the *k-th* control intervals, respectively.

Therefore, a scheduling plan can be determined. From equations (5), (6) and (7), we have:

$$f = \sum_i^n u_i^k \quad (8)$$

$$u_i^k = 1 - e^{a_i m_i \frac{\overline{V}_i - V_i^k}{V_i^k - \hat{V}_i}} \quad (9)$$

$$V_i^k = V_i^{k-1} C_i^k / C_i^{k-1} \quad (10)$$

replacing $V_i^k$ in equation (9) with equation (10) and $u_i^k$ in equation (8) with equation (9), the solution for maximizing the objective function, $f(C_1^k, C_2^k, \cdots, C_n^k)$, is the query scheduling plan for *k-th* control interval, where the object function must maintain the constraint, $C_1^k + C_2^k + \cdots + C_n^k \le C$, and $C$ is the system cost limits.

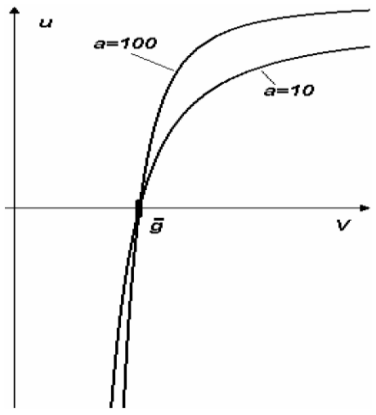


Figure 4. Sample Curves of Utility Function in Query Scheduling

## V. EXPERIMENTS

The experimental objective was to validate the utility functions defined in our studies of autonomic workload management for DBMSs. We developed a dynamic resource allocation simulator and implemented a prototype of the Query Scheduler to examine whether the utility functions can effectively guide the dynamic resource allocation and query scheduling actions in accordance with a given high-level workload business importance policy. We present the results of experiments run using the simulator and the prototype in Subsection *A* and *B*, and discuss the two utility functions in Subsection *C*.

### *A. Experiments for Dynamic Resource Allocation*

To allocate the buffer pool memory and CPU resources, we first experimentally determined the appropriate amount of total resources for a given data server as well as set of workloads. Our experiments were conducted with DB2 database software [6] running on an IBM xSeries® 240 PC server with the Windows® XP operating system. The data server was equipped with two Pentium® processors, 2 GB of RAM and an array of 11 disks. The databases and workloads were taken from the TPC-C benchmark [22]. The size of the database was 10GB. The three workloads were similar to TPC-C OLTP batch workloads.

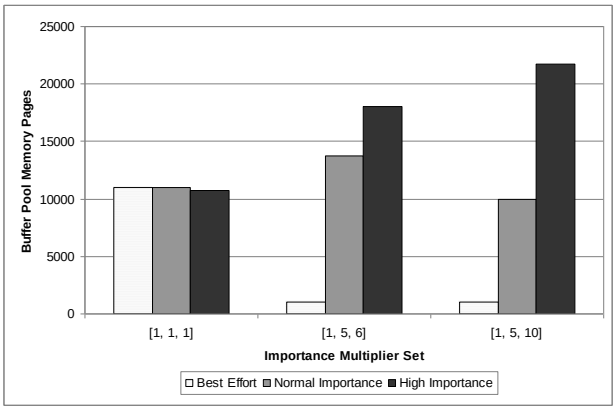


Figure 5. Buffer Pool Memory Allocation for Different Business Importance policies

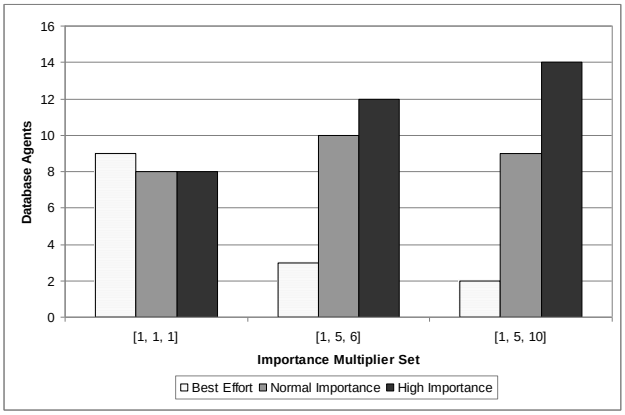


Figure 6. Database Agent Allocation for Different Business Importance policies

We consider the case of a single DB2 instance with three identical databases for three competing workloads from different importance classes. Each database has one workload running on it, thus each workload has its own buffer pool and CPU shares while still having accesses to all the same database objects. Our dynamic resource allocation technique allocates buffer pool memory space and CPU (*i.e.*, database agents) resources across the three identical databases based on a given workload business importance policy.

We selected a minimum amount of each resource (*i.e.*, buffer pool memory and CPU) where maximum system performance was achieved. We experimentally determined 32,768 buffer pool memory pages as the total buffer pool memory and 25 database agents as the total CPU resources [18]. We use 1,000 buffer memory pages as one unit of buffer pool memory and 1 database agent as one unit of CPU resources in our resource allocation experiments (as discussed in the following paragraphs) as these granularities give a reasonable workload performance increment and make the resource allocation process efficient.

We developed a simulator of our dynamic resource allocation approach to generate the resource allocations for competing workloads on a DBMS based on a given workload business importance policy. The simulator was written in Java™ and the three workloads (*i.e.*, the TPCC-like OLTP batch workloads) were used as the simulator input. The output of the simulator was resource allocations, that is, a list of the number of buffer pool memory pages and database agents for each of the workloads.

A set of experiments was conducted to determine whether our approach generates the resource allocations which match a given workload business importance policy. The workloads were assigned one of three different importance classes, namely the *high importance* class, the *normal importance* class, and the *best effort* class. The relative importance of the classes was captured by a set of importance multipliers for the base wealth assigned to the classes. We experimented with three different sets of importance multipliers that were of the form [best effort, normal importance, high importance]: [1, 1, 1], [1, 5, 6], and [1, 5, 10]. The multiplier sets were chosen to demonstrate the effect of business importance policies on the resource allocations.

Figures 5 and 6 respectively show buffer pool memory page and database agent (representing system CPU resources) allocations produced by the simulator using the three workload business importance multiplier sets. The workload importance multiplier set [1, 1, 1] represents the case where the three competing workloads are from three different business importance classes of equal importance. In this case, the three workloads are allocated approximately the same amount of buffer pool memory and CPU resources as shown in Figures 5 and 6. Using the importance multiplier set [1, 5, 6], the high importance and the normal importance classes are much more important than best effort class, and the high importance class is also slightly more important than the normal important class. When the simulator is used to allocate resources in this case, the high importance and normal importance workloads are allocated

significantly more resources than the best effort workload, while the high importance workload is allocated slightly more resources than the normal importance workload. The set [1, 5, 10] represents the case where the high importance class is much more important than the normal importance class, and the normal importance class is much more important than the best effort class. In this case, the high importance workload is allocated more resources than the normal important workload, and the normal importance workload wins significantly more resources than the best effort workload.

By observing the experimental results shown in Figure 5 and Figure 6, we have that the defined utility functions (the key components of the dynamic resource allocation) can effectively guide the resource allocation processes and generates resource allocations for the competing workloads which match the given workload business importance policies (that is, more important workloads assigning more resources than less important ones).

### B. Experiments for Query Scheduling

The same data server, as described in Subsection *A*, was used in the experiments. Our experiments were conducted with DB2 database software as well as DB2 Query Patroller as a supporting component [6]. The database and workloads were taken from the TPC-H benchmark [22]. The size of the database was 500MB, and two workloads that consisted of TPC-H queries were submitted by interactive clients with zero *think time* [20]. Each workload was assigned to a service class described in Section *IV-B*, namely either *class 0* or *class 1*, with a different business importance level and a unique performance goal, where we considered *class 0* is more important than *class 1*. The intensity of a workload in the data server was controlled by the number of clients used by the workload. Each experiment was run for 12 hours that consisted of 6 *2-hour periods* (as shown in Figure 7, 8 and 9).

To evaluate whether our Query Scheduler can manage multiple classes of workloads towards their performance goals based on given workload business importance policies, we first need to determine the *total cost limits*, as mentioned in Section *IV-B*, for the DBMS and workloads. Thus, we experimentally determined 300,000 *timerons*, a measure unit for the resources required by the DB2 database manager to execute the plan for a query [6], as the *total cost limits* in our query scheduling experiments [19].

The first experiment was conducted to show performance of the workloads without control and served as the baseline measure to observe how the performance of the workloads changes as they run. The performance goals of *query execution velocity*, as described in Section *IV-B*, for the workload (belonging to *class 0*) and the workload (belonging to *class 1*) were set as 0.65 and 0.45, respectively. The results are shown in Figure 7. It shows that the "class 0" workload missed its performance goal in *periods* 2 and 3, and the "class 1" workload over performed almost all the time in the experiment.

The experiments were then conducted using our Query Scheduler to control the workloads. The performance goals for *class 0* and *class 1* were still 0.65 and 0.45, respectively. The results are shown in Figure 8. The dynamic adjustment of *service class cost limits* to achieve the performance goals is shown in Figure 9. The experimental results show that our Query Scheduler can provide differentiated services for competing workloads. As shown in Figure 8, for the Query Scheduler, the "class 0" workload could better meet its performance goal than the "class 1" workload, which was in accordance with the given importance policy. Although the

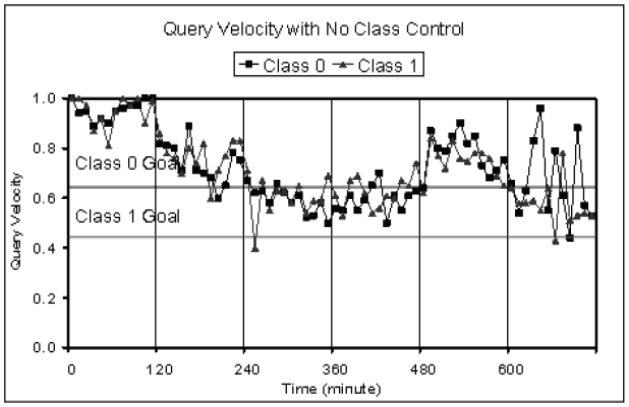


Figure 7. No Service Class Control for Competing Workloads

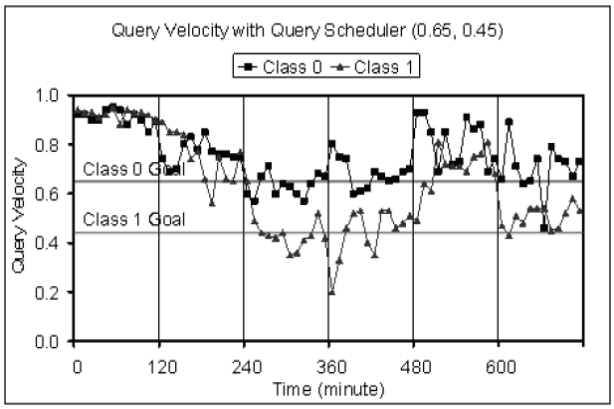


Figure 8. Query Execution Velocity for Multiple Competing Workloads

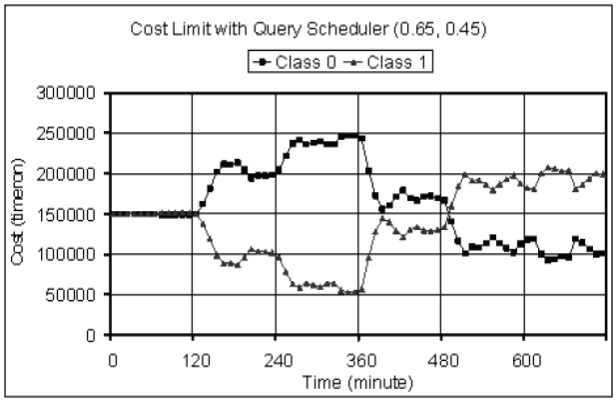


Figure 9. Dynamically Assigned Service Class Cost Limits for Multiple Competing Workloads

Query Scheduler gave preference to the important class, *class 0*, it never allocated too many resources (*i.e.*, multi-programming levels, discussed in Section *IV-B*) *class 0* to prevent *class 1* from meeting its performance goal. When the workloads were too heavy to meet both performance goals as shown at *periods* 3 and 4 in Figure 8, Query Scheduler was still able to help both classes approach their goals. From Figure 9, we can observe that our Query Scheduler dynamically adjusts the *service class cost limits* according to the workload changes. The amount of resources allocated to a class is based on its need in order to meet its performance goal, as shown in Figure 9.

By observing the experimental results shown in Figures 7, 8 and 9, we have that our Query Scheduler is able to respond to query changes and give preference to the queries assigned to an important service class, and to the service class whose performance goals are violated. These results also validate the utility functions as they are key components defined in the Query Scheduler. The results show that the utility functions effectively guide the Query Scheduler to dynamically generate query scheduling plans for competing workloads bases on a given workload business importance policy with more important workloads receiving more shared system resources than less important ones.

### C. Discussion

In dynamic resource allocation, the utility function was defined based on a single-class multi-center closed QNM, while in query scheduling, the utility function was chosen based on an exponential function. These two types of utility functions are different in their forms and research requirements, but both strictly maintain the same fundamental properties listed in Section III.

The input to the dynamic resource allocation utility function is an amount of allocated resources (*i.e.*, the resource pair, <*cpu*, *mem*>), the output is a real number in the range [0, …, 1], and the applied QNM properly predicts performance behavior of the workload. A workload ceases bidding for additional resources using assigned virtual *wealth* when it has reached its performance objective.

In query scheduling, the input to the utility functions is the query execution velocity of the service classes predicted by the performance model given a level of allocated resources and the output is a real number in ($-\infty$, $+\infty$). Based on the exponential function properties, as the input of the utility function increases, the output (*i.e.*, the utility) increases and at a certain value, it begins to level off. That means, when the service class approaches its performance goal, the utility increase is less, and it indicates that the database system should not assign more resources to the service class.

If an objective function is continuous, the *Lagrange* method can be applied to solve it [19], otherwise searching techniques may be used. In query scheduling, the second derivative of the utility function exists and this allows mathematical methods to be applied to optimize the objective function. In dynamic resource allocation, instead of defining an objective function based on the utility functions, *economic models* (the use of virtual wealth and auctions and bids) [18] are applied to coordinate the utility functions to allocate the shared system resources to competing workloads.

TABLE I. COMPARISION OF THE TWO UTILITY FUNCTIONS

| | **Utility functions in Dynamic Resource Allocation** | **Utility functions in Query Scheduling** |
|---|---|---|
| **Utility Increasing Normally** | yes | yes |
| **Marginal Utility Increasing Normally** | yes | yes |
| **Utility Function Input** | allocated resources | a function of the allocated resources |
| **Utility Function Output** | a number in [0…1] | a number in **(**$-\infty$**,** $+\infty$**)** |
| **Critical Resource Identifying** | yes | no |
| **Having Mathematical Property** | no | yes |
| **Utility Increase Stops as Goals Achieved** | yes | yes |

In evaluating the two types of utility functions (using the set of properties listed in Section III), both utility functions preserve the fundamental properties, that is, *a)* the utility increases as a workload performance increases, and decreases otherwise; *b)* the marginal utility is large as a workload performance increases quickly, and is small otherwise; *c)* the input and output are in the required types and values. In comparing the two utility functions presented in Table 1, we observe that the utility function used in dynamic resource allocation has the property of identifying critical resources for a workload, but it does not have mathematical properties for optimizing objective functions (as there is not an objective function defined in the approach). The utility function used in query scheduling possesses a good mathematical property for optimizing its objective function, but it does not have the property of identifying system critical resources (as it is not necessary to identify critical resources in the problem). In Table 1, *Utility Increasing Normally* means whether the utility increases as a workload performance increases, and decreases otherwise, and *Marginal Utility Increasing Normally* means whether the marginal utility is large as a workload performance increases quickly, and is small otherwise.

Since the utility functions were strictly defined based on their research requirements, the specific research problems shaped the utility function's properties. So, we conclude (based on the properties listed in Section III) that the two types of utility functions are good in terms of their specific research requirements and considered acceptable based on the set of properties listed in Section III.

## VI. CONCLUSION AND FUTURE WORK

In this paper, we have presented two concrete examples to illustrate how utility functions can be applied to database workload management, namely dynamic resource allocation and query scheduling. Based on the examples, we generalized a set of function properties that are fundamental for defining utility functions in building autonomic

workload management for DBMSs in future practice and research. Through experiments, we validated the utility functions defined in our studies of autonomic workload management for DBMSs.

As more workload management techniques are proposed and developed, we plan to investigate the use of utility functions to choose during runtime an appropriate workload management technique for a large-scale autonomic workload management system, which can contain multiple techniques. Thus, the system can decide what technique is most effective for a particular workload executing on the DBMS under certain particular circumstance.

## ACKNOWLEDGMENT

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

## TRADEMARKS



## DISCLAIMER

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

## REFERENCES

[1] M. Zhang, B. Niu, P. Martin, W. Powley, P. Bird, and K. McDonald. "Utility Function-based Workload Management for DBMSs". In Proc. of the 7th Intl. Conf. on Autonomic and Autonomous Systems (ICAS'11), Mestre, Italy, May 22-27, 2011, pp. 116-121.

[2] D. P. Brown, A. Richards, R. Zeehandelaar and D. Galeazzi, "Teradata Active System Management: High-Level Architecture Overview", A White Paper of Teradata, 2007.

[3] IBM Corp., "Autonomic Computing: IBM's Perspective on the State of Information Technology". On-line, retreieved in June 2012. http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.

[4] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing", Computer, Volume 36, Issue 1, January 2003, pp. 41-50.

[5] J. O. Kephart and R. Das, "Achieving Self-Management via Utility Functions," IEEE Internet Computing, Vol. 11, Issue 1, January/February, 2007, pp. 40-48.

[6] IBM Corp., "IBM DB2 Database for Linux, UNIX, and Windows Information Center". On-line, retreived in June 2012. https://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp.

[7] Teradata Corp., "Teradata Dynamic Workload Manager", On-line, retrieved in June 2012. http://www.info.teradata.com/templates/eSrchResults.cfm?prodline=&txtpid=&txtrelno=&txtttlkywrd=tdwm&rdsort=Title&srtord=Asc&nm=Teradata+Dynamic+Workload+Manager.

[8] Microsoft Corp., "Managing SQL Server Workloads with Resource Governor". On-line, retrieved in June 2012. http://msdn.microsoft.com/en-us/library/bb933866.aspx.

[9] Microsoft Corp., "Query Governor Cost Limit Option", On-line, retrieved in June 2012. http://msdn.microsoft.com/en-us/library/ms190419.aspx.

[10] Oracle Corp., "Oracle Database Resource Manager", On-line, retrieved in June 2012. http://download.oracle.com/docs/cd/B28359_01/server.111/b28310/dbrm.htm#i1010776.

[11] S. Krompass, H. Kuno, J. L. Wiener, K. Wilkison, U. Dayal and A. Kemper, "Managing Long-Running Queries", In Proc. of the 12th Intl. Conf. on Extending Database Technology: Advances in Database Technology (EDBT'09), Saint Petersburg, Russia, 2009, pp. 132-143.

[12] A. Mehta, C. Gupta and U. Dayal, "BI Batch Manager: A System for Managing Batch Workloads on Enterprise Data-Warehouses", In Proc. of the 11th Intl. Conf. on Extending Database Technology: Advances in Database Technology (EDBT'08). Nantes, France, March 25-30, 2008, pp. 640-651.

[13] J. O. Kephart, "Research Challenges of Autonomic Computing". In Proc. of the 27th Intl. Conf. on Software Engineering (ICSE'05). St. Louis, MO, USA, 15-21 May, 2005, pp. 15-22.

[14] D. A. Menasce and J. O. Kephart, "Guest Editors' Introduction: Autonomic Computing", In IEEE Internet Computing, Volume 11, Issue 1, January 2007, pp. 18-21.

[15] G. Tesauro, R. Das, W. E. Walsh and J. O. Kephart, "Utility-Function-Driven Resource Allocation in Autonomic Systems", In Proc. of the 2nd Intl. Conf. on Autonomic Computing (ICAC'05), Seattle, Washington, USA, June 13-16, 2005, pp.342-343.

[16] W. E. Walsh, G. Tesauro, J. O. Kephart and R. Das. "Utility Functions in Autonomic Systems", In Proc. of the 1st Intl. Conf. on Autonomic Computing (ICAC'04), New York, USA, 17-18 May, 2004, pp.70-77.

[17] M. N. Bennani and D. A. Menasce, "Resource Allocation for Autonomic Data Centers using Analytic Performance Models", In Proc. of the Intl. Conf. on Autonomic Computing, (ICAC'05), Seattle, Washington, USA, 13-16 June, 2005, pp. 229-240.

[18] M. Zhang, P. Martin, W. Powley and P. Bird. "Using Economic Models to Allocate Resources in Database Management Systems", In Proc. of the 2008 Conf. of the Center for Advanced Studies on Collaborative Research (CASCON'08), Toronto, Canada, Oct. 2008, pp. 248-259.

[19] B. Niu, P. Martin and W. Powley, "Towards Autonomic Workload Management in DBMSs", In Journal of Database Management, Volume 20, Issue 3, 2009, pp. 1-17.

[20] 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, 1984.

[21] L. A. Belady. "A Study of Replacement Algorithms for a Virtual-Storage Computer". IBM Systems Journal, Volume 5, Issue 2, June 1966, pp. 78-101.

[22] Transaction Processing Performance Council. On-line, retrieved in Feb. 2012. http://www.tpc.org.