Interaction-Aware Data Management in the Cloud

Manoel Siqueira[1], José Maria Monteiro[1], Angelo Brayner[2],
Flávio R. C. Sousa[1], Javam C. Machado[1]

[1] Universidade Federal do Ceará - Brasil
{manoeljr, monteiro, flavio, javam}@lia.ufc.br
[2] Universidade de Fortaleza - Brasil
brayner@unifor.br

Abstract. Cloud computing is a recent trend of technology aimed at providing on-demand Information Technology (IT) services usually priced on a pay-per-use model. One of the main services provided by a cloud computing platform consists of the data management service, or data service for short. This service accepts responsibility for the installation, configuration and maintenance of database systems, as well as for efficient access to stored data. This paper presents a framework, denoted QIDMaC, for management of cloud databases. The proposed framework aims to provide software infrastructure required for the provision of data services in cloud computing environments in an efficiently manner. In this sense, the proposed solution seeks to solve some outstanding problems in the context of cloud databases, such as: query dispatching and scheduling. The proposed approach extends previous work by adding important features such as: support for unpredictable workloads and the use of information about query interactions. Support for the seasonal workloads is related to one of the main properties of cloud computing: fast elasticity. Query interactions can provide significant impacts on database systems's performance. For this reason, QIDMaC uses information about these interactions in order to reduce the execution time of the workloads submitted to the data service and thereby increase the service provider profit. In order to demonstrate the QIDMaC efficiency an experimental evaluation using TPC-H benchmark was performed on PostgreSQL. The results show that the designed solution has the potential to increase the profit of cloud data service providers.

Categories and Subject Descriptors: H.2 [**Database Management**]: Miscellaneous

Keywords: cloud database, data management, interaction-aware

1. INTRODUCTION

Cloud computing is an extremely successful paradigm of service-oriented computing and has revolutionized the way in which computing infrastructure is abstracted and used. Scalability, elasticity, pay-per-use pricing, and economies of scale are the major reasons for the successful and widespread adoption of cloud infrastructures. Since the majority of cloud applications are data-driven, database management systems (DBMSs) powering these applications are critical components in the cloud software stack [Elmore et al. 2011].

Nowadays, there is a high demand for data management service to be provided by cloud systems [Abadi 2009]. Thus, we advocate that data management service should be responsible for installing, configuring and maintaining database systems. Additionally, it has to ensure efficient access to stored data. To provide data service in a cloud system, the provider should deliver the required infrastructure (hardware and software), measure the service usage, guarantee the dealt service quality and charge in per-use basis. We define a data management service provided by cloud systems as service consisting of a multiple database system (MDBS) running on several virtual machines (VMs) afforded by an infrastructure provider as a service (*Infrastructure as a Service* - IaaS). In this way, each VM hosts a single DBS and a database replica. Nonetheless, the set of DBSs in the different VMs behaves as a single logical database.

This work is partially funded by FUNCAP and CNPq.

Many companies expect cloud data management service providers to guarantee quality of service (QoS) using service level agreement (SLAs). However, dealing with unpredicted load patterns and elasticity is critical to ensure that the SLAs are met. There are different strategies to improve QoS, comply SLA and increase provider's profit, such as database replication, adaptive query processing, capacity planning, query scheduling and dispatching.

Still, a typical database workload consists of a mix of multiple query instances of different query types that run and interact with each other. A query type can be defined as a template for SQL queries, which consists of a SQL expression with parameter markers. Whenever a template is instantiated with a set of parameter values, one has a query instance. Query instances in a workload can have interactions with a significant impact on database performance, which can be positive or negative [Ahmad et al. 2009].

In this article, a query interaction-aware framework for providing data service in cloud systems, denoted QIDMaC (Query Interaction-aware Database Management in the Cloud), is presented. So, QIDMaC explores the query interaction concept to improve the data service provider's profit. The main feature presented by QIDMaC is to deliver required software infrastructure to make data service available in cloud computing environment. More specifically, QIDMaC implements efficient solutions for database query dispatching and scheduling by exploiting query interactions. The key goal of query dispatching mechanism is to allocate a given query to one VM v_k, ensuring that the service provider's profit obtained by executing q in v_k is the largest on w.r.t. the other VMs. In turn, a scheduling mechanism has to identify the most efficient schedule to execute several queries regarding query profit profile in order to maximize the service provider's profit. In other words, QIDMaC exploits query interaction in order to reduce the query response time and, consequently, increasing the likelihood of meeting SLA, reducing penalties and enhancing the provider's profit.

The proposed framework has been evaluated. For that, the TPCH benchmark has been used on PostgreSQL. The results show that QIDMaC has potential to improve the efficiency of database service and to increase the profit of cloud data service providers.

The rest of this article is organized as follows. Section 2 discusses related work. QIDMaC is described and analyzed in Section 3. Section 4 brings and analyzes experimental results. Finally, Section 5 concludes this work.

2. RELATED WORK

As already mentioned, in this work we aim at proposing an integrated solution for query scheduling and dispatching problems. In this sense, some approaches for coping with those problems are analyzed in this section.

Regarding query scheduling, Sharaf *et al* proposes in [Sharaf et al. 2009] a mechanism for scheduling I/O operations in order to reduce response time of queries which do not fulfill their SLAs. In this sense, that approach does not define the execution order of queries. It only schedules I/O operations resulting from query executions. In [Tilgner 2010] is proposed a query scheduler, which exploits SLA consistency metric as criterion for defining a query execution schedule. Thus, queries are executed concurrently by different DBSs, which in turn are running in different VMs. A similar approach is proposed by Costa and Furtado in [de Carvalho Costa and Furtado 2008]. However, the latter approach uses query SLOs for defining the amount of queries, which may be executed concurrently.

Next, the most referenced approaches on query dispatching are analyzed. Xion *et al* propose in [Xiong et al. 2011] a framework, which makes available a query dispatching mechanism for database service in cloud computing environment. The idea is to estimate the probability of a given query q to meet its SLO by means of machine learning techniques. According to the estimated probability, q is dispatched or not. Another approach for query dispatching in database service is proposed in [Schroeder and Harchol-Balter 2004]. In that approach, queries are classified in different groups. Thus, each server can only process queries of a given group.

In [Chi et al. 2011] is proposed a mechanism for supporting profit-oriented decisions regarding query dispatching in the cloud. Based on profit information and SLA of queries waiting in the server buffer to be dispatched for being executed, a data structure, called SLA-Tree is built. After the SLA-Tree is yielded, a set of "what-if" questions are "answered" by a component of the proposed framework. For example, for each available server S, the following questions should be answered: "What is the estimated profit change if a query q is dispatched to be executed in S?". Depending on the "answer", q is dispatched to be executed in S or not. The authors do not make clear how such "what-if" questions are modeled and processed ("answered").

Rogers et al proposes in [Rogers et al. 2010] a framework for resource provisioning and query dispatching. The proposed framework identifies a set of minimum-cost infrastructure resources (virtual machines), which are able to execute a workload assuring QoS expectations. The criterion for query dispatching is quite naive, since it is restricted to allocate queries to machines with sufficient resources to execute queries respecting QoS metrics. A query dispatching strategy based on utility functions is proposed in [Paton et al. 2009]. Thus, queries are dispatched to multiple servers according to the utility functions. An utility function $U(w,d)^P$ computes the benefit of a query dispatching policy d to available servers for a workload w regarding the property P. Thus, the approach is highly dependent on a correct definition of the utility function. Moreover, it does not consider the use of SLA and the workload should be known a priori.

3. QIDMAC: A QUERY INTERACTION-AWARE FRAMEWORK TO DATA MANAGEMENT IN THE CLOUD

In order to solve the problems of scheduling and dispatch satisfactorily, we propose a framework for data management in the cloud, called QIDMaC, which explores query interactions to increase performance. Additionally, QIDMaC has the following characteristics: non-intrusive, uses a generic cost model, based on SLAs, profit oriented and supports unexpected workloads.

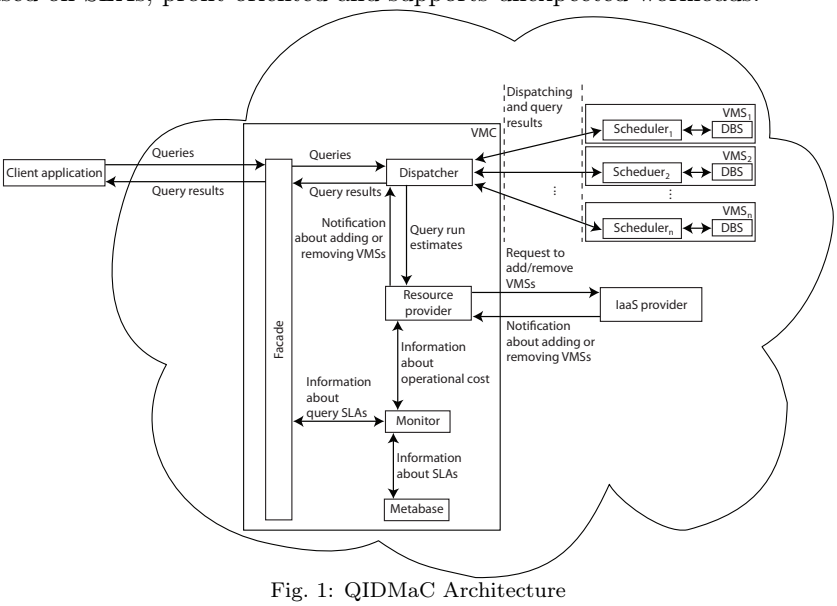


Fig. 1: QIDMaC Architecture

Figure 1 depicts QIDMaC architecture, which consists of five modules, divided into two different types of virtual machines. These kinds of virtual machines are:

—**VM Controller (VMC):** this VM hosts the modules responsible for solving the dispatching (dispatcher module) and resource provisioning (resource provider module) problems, as well as the user interface (facade module) and the statistics collector (monitor module). In addition, information concerning SLAs is stored in a component called metabase. It is noteworthy that the QIDMaC uses only one instance of VMC;

—**VM Scheduler (VMS):** each VMS hosts one instance of the scheduler module and a DBS complete replica. The scheduler module defines the execution order of the queries dispatched to it, and sends these queries to be performed on the DBS copy hosted locally. Note that there may be various VMSs, that is, multiple instances of the VM Scheduler.

The QIDMaC framework consists of five distinct modules (Figure 1). Next, we will describe each of these modules:

—**facade:** client applications send requests to the facade module requesting the execution of a particular SQL query.

—**dispatcher:** dispatcher module aims to decide which instance of the scheduler module will perform a given query.

—**scheduler:** each instance of the scheduler module generates the estimates used by the dispatcher module and defines the execution order of queries sent to it.

—**resource provider:** resource provider module decides, based on information provided by dispatcher module, about the VMSs instantiation/suspension need. This module will not be discussed in this article;

—**monitor:** the monitor module stores in the metabase information about SLAs compliance agreed with the client application and on operating costs arising from the VMs use.

3.1 Multi-Tenant Model

There are many ways to deploy a database as a service on a cluster with multi-tenancy: (1) all tenant data are stored together within the same database and the same tables with extra annotation such as "TenantID" to differentiate the records from different tenants; (2) tenants are housed within a single database, but with separate schemas to differentiate their tables and provide better schema-level security; (3) each tenant is housed in a separate database within the same DBMS instance (for even greater security); (4) each tenant has a separate VM with an OS and DBMS, which allows for resource control via VM management [Barker et al. 2012].

In multi-tenant strategies (1), (2) and (3), interference may occur between tenants. This is related to the DBMS type used and to the workload. In this work, we used option (4) to implement multi-tenancy, where each tenant database runs in its own virtual machine. This option uses more resources, but the level of sharing allows us to control the system resources allocated for each VM, or the corresponding tenant. Furthermore, option (4) provides better security and it is used in many works.

3.2 SLA

SLA is an agreement between two parties: the customer and the provider of a particular service [Comellas et al. 2010], whose purpose is to ensure the quality of the contracted service. The QIDMaC framework uses the SLA concept to ensure the cloud data service quality. The SLA concept used in QIDMaC has as a metric the SQL query response time.

We define the SQL response time, for a for a given query q_i, as the time elapsed since q_i arrives in the cloud data service (at facade module) until the delivery of the q_i execution results to the customer. Thus, the response time includes: i) dispatching time, the time required to select a VMS ms_j and dispatch q_i to ms_j; ii) scheduling time, the time necessary to choose a position p in the scheduling queue F of ms_j and put q_i in this position; iii) waiting time (or starting time), the time that q_i remains in the queue F waiting to start its execution; iv) runtime, time spent by the DBMS query processor to run q_i; and v) delivery time, time necessary to send the q_i execution results to the customer. For simplicity, dispatching time, scheduling time and delivery time are disregarded. So, SQL runtime is just a component of SQL response time. It's important to note that the waiting time for q_i depends of the runtime of the queries in the previous positions in the queue F.

The estimated runtime of a query instance q_i was obtained by running q_i with empty cache (memory free) 5 times and getting the average. Then, the estimated response time for q_i includes: i) estimated

dispatching time; ii) estimated scheduling time; iii) estimated waiting time (or estimated starting time); iv) estimated runtime and v) estimated delivery time. For simplicity, estimated dispatching time, estimated scheduling time and estimated delivery time are disregarded. It's important to note that the estimated waiting time for q_i can be calculated as the sum of the estimated runtime of the queries in the previous positions in the scheduling queue F of ms_j.

However, unlike [Sousa et al. 2012], SLO (e.g., response time less than 10 minutes), revenues and penalties are defined for each SQL query type Q_k. Thus, each query instance q_i of Q_k has the same values of SLO, revenue and penalties defined to the query type Q_k. In [Sousa et al. 2012] all SQL query instances, regardless of its type, has the same values for SLO, revenue and penalties. The choice of this new SLA definition was motivated by the fact that most of the applications currently available in the cloud have OLAP (Online Analytical Processing) characteristics which involves workloads whose queries have response times with different magnitude orders (e.g., while some SQL queries take 16 seconds others take 15 minutes).

To illustrate the SLA use in the proposed framework, consider a workload W. Let R_W the revenue for W, C_W the operating cost (cost to using VMs) to run W and P_W the penalties imposed to the data service provider after finishing the execution of W. The profit earned by performing W, represented by Pr_W, is defined by the following formula:

$$Pr_W = R_W - (C_W + P_W) \tag{1}$$

The revenue R_W is the sum of revenues from all queries $q_i \in W$, regardless of SLAs compliance. Let the query instance revenues r_{q_i} the monetary value agreed between the customer and the data service provider for a given query instance $q_i \in W$, R_W is obtained by the following formula:

$$R_W = \sum_{q_i \in W} r_{q_i} \tag{2}$$

Similar to [Silva et al. 2012], in this work, time was discretized into billable units (e.g., in Amazon EC2[1] revenue is calculated every hour). However, the operating cost model was simplified by assuming that each VM instance will have a fixed cost for each billable unit, depending on its type. For example, each instance of the type *Small* (a VM type in Amazon EC2[2]) will generate a fixed operating cost (e.g., U$ 1.00 per hour).

Let H the set of possible VM types, E_W the runtime for the workload W (using the n existing VMSs and the VMC), E'_W the time E_W converted to the used billing unit, $n_W(h,t)$ the amount of instantiated VMs of the type h in a time billing unit t and $c(h)$ the cost of using a VM of the type $h \in H$ per time billable unit. Thus, suppose that the time billable unit is one hour and $E_W = 45$ min. In this case, $E'_W = 1$ hour, since the payment is recorded every hour. Then, the operating cost C_W is calculated by the following formula:

$$C_W = \sum_{h \in H} \sum_{t=1}^{E'_W} n_W(h,t) \times c(h) \tag{3}$$

Not meeting the SLAs associated with the SQL query instances of W leads to the payment of penalties P_W by the data service provider to the client. Let p_{q_i} the penalty associated with a given query instance $q_i \in W$, P_W is defined by the following formula:

$$P_W = \sum_{q_i \in W} p_{q_i} \tag{4}$$

Let rt_{q_i} the response time of a given query instance q_i, SLO_{q_i} the service level objectives for q_i, that is, the maximum acceptable time to run q_i without any penalty and r_{q_i} the revenue associated

[1]http://aws.amazon.com/ec2/pricing/
[2]http://aws.amazon.com/ec2/instance-types/

with the query instance q_i. So, the following equation describes how p_{q_i} is calculated. It is noteworthy that other penalty models can be used.

$$p_{q_i} = \begin{cases} r_{q_i}, \text{ if } rt_{q_i} > SLO_{q_i} \\ 0, \text{ otherwise} \end{cases} \tag{5}$$

Derived from the penalty concept, we also use the concept of estimated penalty. Let q_i a query instance, the estimated penalty ep_{q_i} refers to the monetary value that will be paid by the data service provider to the client, if the estimated response time for q_i, denoted ert_{q_i}, is greater than the SLO defined for q_i. Thus, the estimated penalty $ep_{q_i}(ert_{q_i})$, according to the estimated response time ert_{q_i} for q_i is defined by the following formula:

$$ep_{q_i}(ert_{q_i}) = \begin{cases} r_{q_i}, \text{ if } ert_{q_i} > SLO_{q_i} \\ 0, \text{ otherwise} \end{cases} \tag{6}$$

The estimated response time of a given query q_i can be calculated according to the estimated time instant est to start running q_i. So, consider $ert_{q_i}(est)$ estimated response time for q_i. Let $erunt_{q_i}$ the estimated runtime for a query instance q_i (elapsed time since start running q_i until its end) and at_{q_i} the arrival time of q_i in the data service. Thus, the estimated response time for query q_i is calculated by the following formula:

$$ert_{q_i}(est) = (est + erunt_{q_i}) - at_{q_i} \tag{7}$$

An additional concept, derived from the definitions of [Sousa et al. 2012], used in the proposed solution is the balance, which consists of revenues minus the penalties. The balance is defined on two levels: by SQL query instance (Equation 8) and by workload (Equation 9). So, consider that b_{q_i} represents the balance of the query instance q_i and B_W, the balance of the workload W. Suppose that r_{q_i} and p_{q_i} are the revenue and the penalty associated with q_i, respectively, and R_W and P_W are the revenue and the penalty related with W, respectively. So, we have:

$$b_{q_i} = r_{q_i} - p_{q_i} \tag{8}$$

and

$$B_W = R_W - P_W \tag{9}$$

Based on the balance concept, QIDMaC also uses the estimated balance concept, which is the revenue r_{q_i} minus the estimated penalty $ep_{q_i}(ert_{q_i})$ (Equation 6). The estimated balance $eb_{q_i}(est)$ for q_i, if q_i starts its execution at time instant est, can be calculated by the Equation 10. This formula considers the revenue r_{q_i} and the estimated penalty $ep_{q_i}(ert_{q_i}(est))$ (Equation 6). In this case, set is the estimated starting time for q_i and $ert_{q_i}(est)$ (Equation 7), the estimated response time, if q_i starts its execution at time est.

$$eb_{q_i}(est) = r_{q_i} - ep_{q_i}(ert_{q_i}(est)) \tag{10}$$

For calculating the cloud data service profit, some information about SLA are stored in the metabase. Next, we will describe these information: i) k, query type identifier; ii) SQL text; iii) SLO_{Q_k}, maximum response time for any query instance of the query type Q_k; and iv) r_{Q_k}, represents the monetary value agreed between the customer and the service provider to perform a query instance q_i of the type Q_k.

After the q_i execution, the response time rt_{q_i} for q_i is obtained. All these information (r_{q_i}, SLO_{q_i} and rt_{q_i}) are stored in the metabase. Thus, the equations 5 and 8 can be applied to calculate the penalty and the balance of q_i, respectively.

3.3 Query Interaction

Nowadays, most databases are stored in hard disks. Data access rate in hard disks is several magnitude orders lower than in main memory, specially w.r.t. random accesses.

To execute a query q_i, the buffer manager may load data pages into the buffer pool, which are used by another query q_k. Such a scenario characterizes a query interaction between q_i and q_k. Of course,

executing q_i before q_k (or vice-versa), q_k can profit from the fact that pages, necessary to process it, are already in the buffer pool.

Based on this observation, [Siqueira et al. 2012] presents three approaches to model and measure query instance and query type interactions. These approaches, denoted intercalation strategy (IS), data retrieving rate (DRR) and greedy bidimentional array (GBA), do not require any prior assumptions on internal aspects of the database system.

Besides, these approaches use the concept of *interaction factor*, which is a number between 0 and 1. The interaction factor quantifies the interaction between two query instances or between two query types. Values close to 1 indicate strong interaction and close to 0 weak interaction. Then, given two queries q_i and q_j as input, IS, DRR and GBA produce as output the *interaction factor* between q_i and q_j. Next, based on the *interaction factor* concept, we define the following terms:

Definition 3.1 *Interaction gain.* Let q_i a query to run at a given DBMS. The interaction gain for q_i represents the estimated gain provided by running q_i after a query q_j and before a query q_{j+1}.

Equation 11 supposes that $f(q_i, q_j)$ represents the interaction factor between q_i and q_j, where q_i and q_j are query instances. Besides, it supposes that q_i runs before q_j.

$$ig(q_i, q_j, q_{j+1}) = \begin{cases} f(q_i, q_{j+1}), & \text{if } q_j \text{ is null} \\ f(q_j, q_i), & \text{if } q_{j+1} \text{ is null} \\ (f(q_j, q_i) + f(q_i, q_{j+1})) - f(q_j, q_{j+1}), & \text{otherwise} \end{cases} \quad (11)$$

Definition 3.2 *Balance gain.* Let q_i a query instance, sm_k a scheduler module instance, F the set of queries in the scheduling queue of sm_k, EB_F a profit estimate of the balance provided by the execution of queries belonging to the queue F, and $EB_{F \cup \{q_i\}}$ the balance estimate provided by running q_i in sm_k, the balance gain (cash value) to execute the query q_i in the instance sm_k, represented by $bg(q_i, F)$, is calculated according to the Equation 12.

$$bg(q_i, F) = EB_{F \cup \{q_i\}} - EB_F \quad (12)$$

3.4 Dispatching

The dispatching problem consists in to allocating (assigning) each query q_i, belonging to a given workload W, to one of the available scheduler instances, in order to provide the highest possible profit to the data service provider. Initially, the client application sends a message to the facade module requesting the execution of a given query q_i. Then, facade module redirects this request for the dispatcher module. Then, the dispatcher sends a request, containing the query q_i, for each scheduler instance me_k, requesting an estimate (to be stored in the variable e) of the profit that the execution of the query q_i in the instance me_k would bring to the data service. This benefit is measured by using two metrics: interaction gain (Definition 3.1) and balance gain (Definition 3.2). These two metrics are computed by a procedure called hypothetical scheduling. After all scheduler instances send their estimates to the dispatcher, this module selects the scheduler instance that provides the greatest balance gain. Let bg_{max} the highest value returned to the balance gain. If two or more instances return the same value bg_{max} for the balance gain, the dispatcher will select the scheduler instance with highest value to interaction gain estimate. Finally, the dispatching module will send the query q_i for the selected scheduler instance.

Let F the scheduling queue and n the number of queries in F. The position with index 0 (first position) in F is reserved for the last query submitted to execution. The position with index 1 (second position), in turn, stores the next query to be executed. So, the position with index $n-1$ stores the last query to be executed and the position with index n is the first free position in F. For example, suppose $n = 10$. Thus, the queue F has 9 queries ($n-1$ queries) to run and the first free position in F is the position 10 (position n). In addition, new queries can be inserted between positions 1 and n,

inclusive, during the scheduling process. Moreover, whenever a query q_i is inserted in a given position of the queue F, each query located after q_i in the queue F will have its position incremented by one. In this article, we assume that $|F| = n$, i.e., they will have $|F|$ queries in the queue F.

Let sm_k a scheduler instance, F the scheduling queue at sm_k and $q_j \in F$ a query inserted in the queue F, such that $1 \leq j \leq |F| - 1$. The balance gain definition (Definition 3.2) uses the balance estimate EB_F provided by the execution of queries in F. This estimate is calculated by the Equation 13, which consists of the sum of the estimated balance (Equation 10) for each query q_j, considering the estimated initial time est_{q_j} previously defined for q_j. Another estimate used in the balance gain definition (Definition 3.2), represented by $EB_{F\cup\{q_i\}}$ refers to the balance provided by running q_i at sm_k (Equation 14). $EB_{F\cup\{q_i\}}$ is the maximum estimated balance obtained by inserting q_i at each of the possible positions of the queue F (between positions 1 and $|F|$, inclusive). Thus, $EB_{F\cup\{q_i\}}(j)$ is the estimated balance obtained by inserting q_i in the position j of F. So, $EB_{F\cup\{q_i\}}(j)$ is calculated by the Equation 15 and consists of the sum of the following terms:

(1) sum of the estimated balance of each query q_l, such that $1 \leq l \leq j - 1$, considering the estimated starting time est_{q_l} previously defined for q_l;
(2) estimated balance for query q_i, considering the estimated starting time est_{q_j} previously defined for q_j. This balance is calculated based on the time est_{q_j} since the estimated balance $EB_{F\cup\{q_i\}}(j)$ refers to the insertion of q_i in the position j of the queue F. Thus, the estimated starting time for q_i would be the same estimated starting time previously defined for q_j;
(3) sum of the estimated balance of each query q_l, such that $j \leq l \leq |F| - 1$, considering that each query q_l would be delayed according to the estimated runtime $erunt_{q_i}$ of q_i, that is, the estimated starting time est_{q_l} of q_l would be delayed according to $erunt_{q_i}$ time.

$$EB_F = \sum_{j=1}^{|F|-1} eb_{q_j}(est_{q_j}) \tag{13}$$

$$EB_{F\cup\{q_i\}} = max\{EB_{F\cup\{q_i\}}(j) | 1 \leq j \leq |F|\} \tag{14}$$

and

$$EB_{F\cup\{q_i\}}(j) = \sum_{l=1}^{j-1} eb_{q_l}(est_{q_l}) + eb_{q_i}(est_{q_j}) + \sum_{l=j}^{|F|-1} eb_{q_l}(est_{q_l} + erunt_{q_i}) \tag{15}$$

Next, we will describe in detail all the steps and algorithms involved in the proposed solution for the dispatching problem.

The dispatching process starts when the client application sends a message to the facade module requesting the execution of a given query q_i. Upon receiving this request, the facade module performs Algorithm 1 (main algorithm at facade module), which takes as input the query q_i and uses dm as a global variable referencing the dispatcher module. Then, the facade module creates a data structure q'_i to represent the query q_i (lines 2 to 6 at the Algorithm 1). This data structure, whose type was called SLAQueryDS, contains the query itself q_i, besides some additional information: the time instant at which the query q_i was received at the facade module, i.e., the arrival time of q_i at the data service (line 2), the q_i identifier (line 3), the revenue and the SLO (maximum response time) from q_i (line 4) agreed between the customer and the data service provider, and the estimated runtime for q_i, which must be previously obtained (line 5). In the experiments, we defined one SLO for each query type Q_k. Thus, all instances of the same query type have the same SLO. The estimated runtime of a query instance q_i was obtained by running q_i with empty cache (memory free). Next, the facade module redirects the received request, now containing a data structure that represents the query q_i (that is, q'_i), to the dispatcher module (line 7 at Algorithm 1, calling Algorithm 2 from dispatcher module). Subsequently, the response time $rt_{q'_i}$ is calculated (line 8) and q'_i is sent to the monitor module, in

order to update the statistics regarding q'_i (line 10). Algorithm 2 will return the result of the query q'_i, which is stored in the variable *result*. Finally, Algorithm 1 returns the variable *result* to the client application (line 11).

Algorithm 1: Main algorithm of facade module

```
   input  : query instance q_i
   output : query result
   Data: dm reference to dispatcher module
 1 begin
 2     at_{q_i} ← getCurrentTime();
 3     id_{q_i} ← generateQueryId();
 4     sla_{q_i} ← getSLA(q_i);
 5     erunt_{q_i} ← getEstimatedRunTime(q_i);
 6     q'_i ← createSLAQueryDS(id_{q_i}, q_i, sla_{q_i}, at_{q_i}, erunt_{q_i});
 7     result ← dm.Algorithm2(q'_i);
 8     rt_{q_i} ← getCurrentTime() − at_{q_i};
 9     q'_i.setResponseTime(rt_{q_i});
10     sendStatisticsToMonitor(q'_i);
11     return result;
12 end
```

The dispatcher module, upon receiving a message from the facade module requesting the execution of a query q_i, runs Algorithm 2 (algorithm for receiving query request at dispatcher module), which receives as input the data structure q'_i, which represents the query q_i. Furthermore, the global variable D, which represents the dispatching queue, is used by Algorithm 2. The dispatching queue is used to store the queries to be dispatched by the dispatcher module. Initially, the Algorithm 2 inserts the query q'_i into the dispatching queue D (line 2) and the dispatcher notifies himself informing that a new query was added in the dispatching queue (line 3). Algorithm 3 (main algorithm for query dispatching) is responsible for receive such notification and dispatch the query q'_i. While q'_i does not finish its execution, the dispatcher module waits for a notification informing that q'_i results is available (between the lines 4 and 6). Finally, the q'_i results is returned to the facade module (line 7).

Algorithm 2: Algorithm for receiving query request at dispatcher module

```
   input  : data structure q'_i
   output : query result
   Data: dispatching queue D
 1 begin
 2     D.add(q'_i);
 3     notifyDispatcher(q'_i);
 4     while not hasResult(q'_i.getId()) do
 5         waitForResultNotification();
 6     end while
 7     return getResult(q'_i.getId());
 8 end
```

Algorithm 3: Main algorithm to query dispatching

```
   Data: list V of available instances of scheduler module
 1 begin
 2     while true do
 3         q'_i ← getNextQuery();
 4         foreach sm_k ∈ V do
 5             e ← sm_k.Algorithm4(q'_i);
 6             sm_k.updateEstimates(e);
 7         end foreach
 8         sm_k ← Algorithm5(q'_i);
 9         sm_k.Algorithm7(q'_i);
10     end while
11 end
```

Upon being notified that a new query was added in the dispatching queue, the dispatcher runs Algorithm 3 (main algorithm for query dispatching), which uses a global variable V, which represents

the list of available scheduler module instances. This algorithm runs indefinitely by repeating the loop between lines 2 and 10. Within this loop, the first step consists in obtaining the next query to be dispatched (line 3), which is stored in the variable q'_i. Next, the loop between lines 4 and 7 is executed. This loop runs the hypothetical scheduling of q'_i in each scheduler module instance $sm_k \in V$ (line 5, call to the Algorithm 4). That is, the dispatcher module sends a message, containing query q'_i, for each scheduler module instance $sm_k \in V$ requesting an estimate of the profit that the execution of query q'_i at the instance sm_k would bring to the data service (line 5). As mentioned earlier, this profit is measured by using two metrics: interaction gain (Definition 3.1) and balance gain (Definition 3.2). Besides these two metrics, variable e stores the estimated response time associated with a given scheduler module instance sm_k, which is the time instant estimated to complete the execution of the last query in the scheduling queue at sm_k subtracted from the current time instant. This information is used to identify if the instance sm_k is overloaded compared to other scheduler instances. The balance gain, interaction gain and estimated response time will be used by Algorithm 5 to select the scheduler instance for which the query q'_i will be dispatched. After an instance sm_k to answer the request sent by the dispatcher, the balance and interaction gains (if q'_i would dispatched to sm_k) estimates, as well as the estimated response time associated with sm_k are updated (line 6). At line 8, the algorithm selects the scheduler instance sm_k with highest interaction gain among those with higher balance gain (call to Algorithm 5). This instance will be responsible for executing q'_i. Next, the dispatcher will send the query q'_i to the selected scheduler instance (line 9, call the Algorithm 7, main algorithm at scheduler module).

Algorithm 4 (the algorithm for hypothetical scheduling is performed by a scheduler instance) describes the hypothetical scheduling of query q'_i in a particular scheduler instance sm_k. Thus, this algorithm runs at scheduler instances. Algorithm 4 has as input a query q'_i. The output of this algorithm, represented by the variable e, consists of the estimated response time needed to run all queries in the scheduling queue of sm_k, as well as an estimate of the benefit that the execution of the query q'_i at sm_k would bring to the data service (more precisely, this benefit is measured by the estimates balance gain and interaction gain). This variable consists of a data structure whose type, denoted EstimatesDS, consists of: i) id_{sm_k},the scheduler instance identifier, ii) balance gain provided by q'_i; iii) interaction gain provided by q'_i, and, iv) $t_{response}$, estimated response time needed to run all queries in the scheduling queue of sm_k. In addition, Algorithm 4 uses the global variable F, which represents the scheduling queue of sm_k. Initially, it runs a call to Algorithm 8 (line 2). Algorithm 8 aims to estimate the balance gain and the interaction gain that would be afforded by the execution of query q'_i at the instance sm_k. Algorithm 8 will be described in detailed later. A data structure containing these estimates is stored in the variable g. Next, the estimated response time for the execution of all queries in F is stored in the variable $t_{response}$ (line 3). This information is used to assist in selecting the scheduler instance responsible for running q'_i, at Algorithm 5. Finally, a data structure of type EstimatesDS containing the estimates generated by the hypothetical scheduling, is created (line 4) and returned (line 5).

Algorithm 4: Algorithm to hypothetical scheduling used by scheduler module instances

```
   input  : data structure q'_i
   output : data structure e
   Data: scheduling queue F, identifier id_{sm_k} of scheduler module instance sm_k
1  begin
2  |  g ← Algorithm8(q'_i);
3  |  t_response ← q_{|F|-1}.getFinalTime() - getCurrentTime();
4  |  e ← newEstimates(id_{sm_k}, g, t_response);
5  |  return e;
6  end
```

The algorithm for selecting a scheduler instance, called by Algorithm 3, is described by Algorithm 5 (algorithm to select a scheduler instance to run a given query q'_i). Algorithm 5 has as input query q'_i to be executed and as output the scheduler instance selected to run q'_i. In addition, the algorithm

uses a global variable V, which represents the list of scheduler instances. Moreover, algorithm uses a constant c in order to define a upper limit to $V_{interaction}$. So, in our experiments, we used $c = 0.5$.

Algorithm 5: Algorithm to choose which scheduler module instance will run a query instance q'_i

```
   input  : data structure q'_i
   output : reference sm_chosen to scheduler module instance
   Data: available instance list V of scheduler module, constant c
1  begin
2  |  bg_max ← max{bg_sm_k | sm_k ∈ V};
3  |  V_balance ← {sm_k ∈ V | bg_sm_k = bg_max};
4  |  erunt_min ← min{erunt_sm_k | sm_k ∈ V_balance};
5  |  erunt_max ← (1 + c) × erunt_min;
6  |  gi_max ← max{gi_sm_k | sm_k ∈ V_balance and erunt_sm_k ≤ erunt_max};
7  |  V_interaction ← {sm_k ∈ V_balance | gi_sm_k = gi_max and erunt_sm_k ≤ erunt_max};
8  |  sm_chosen ← getElement(V_interaction);
9  |  return sm_chosen;
10 end
```

Let sm_k a scheduler instance, F the scheduling queue of sm_k, bg_{sm_k} the balance gain of running the query q'_i at sm_k, gi_{sm_k} the interaction gain provided by running query q'_i at sm_k and $erunt_{sm_k}$ the estimated time interval needed to run all queries in F. Besides, bg_{sm_k}, gi_{sm_k} and $erunt_{sm_k}$ are the estimates obtained in the main dispatching algorithm (Algorithm 3). Thus, initially, algorithm 5 gets the set $V_{balance}$ of schedules instances with the higher balance gain, considering all available scheduler instances in V (lines 2 and 3). Next, the algorithm obtains $erunt_{min}$, the minimum value of $erunt_{sm_k}$, considering each $sm_k \in V_{balance}$ (line 4). So, $erunt_{min}$ is used to calculate the upper limit for the runtime ($erunt_{max}$) of a given instance $sm_k \in V_{balance}$ (line 5). Subsequently, the algorithm calculates the maximum interaction gain gi_{max}, considering the instances $sm_k \in V_{balance}$ with estimated runtime ($erunt_{sm_k}$) less or equal to $erunt_{max}$ (line 6). Thus, this algorithm defines the set $V_{interaction}$ of scheduler instances with interaction gain equal to gi_{max}. The restriction $erunt_{m_k} \leq erunt_{max}$ (lines 6 and 7) aims to avoid the overload of a scheduler instance sm_k. Finally, the scheduler instance sm_{chosen} returned (line 9) can be any instance in the set $V_{interaction}$ (line 8).

Suppose that a query q'_i has been dispatched to the scheduler instance sm_k. In this case, at the time that q'_i finishes running, sm_k sends the results of q'_i to the dispatcher. This result is sent by calling the Algorithm 6 at the dispatcher. Algorithm 6 has as input the query q'_i and the variable *result*, which represents the results of q'_i. Moreover, the Algorithm 6 uses the global variables V (list of scheduler instances) and R (a key-value mapping, where key is the query identifier and value is the query result). At Algorithm 6, the first step is to store the result of q'_i in R (line 2). The mapping R is used in Algorithm 2 to send the result of q'_i from the dispatcher to the facade module. Then, in line 3, the dispatcher notifies itself in order to continue the execution of the Algorithm 2 (which sends the results from dispatcher to facade module).

Algorithm 6: Algorithm for receiving by dispatcher module the result of executing a query q'_i on an instance sm_k of scheduler module

```
   input  : data structure q'_i, variable result
   Data: available instance list V of scheduler module, key-value map R
1  begin
2  |  R.addResult(q'_i.getId(), result);
3  |  notifyResult();
4  end
```

3.5 Query Scheduling

Let sm_k a scheduler instance, F the scheduling queue of sm_k and q'_i a query instance dispatched to sm_k, which must be inserted into the queue F. The query scheduling problem consists in choosing the best query execution sequence (for queries in F) in order to maximize the profit of the data service provider. In this work, this sequence is defined when a query q'_i is inserted in F. Thus, the problem can be redefined as follows: given a scheduling queue F and a query instance q'_i, in which position

of F should we must insert q'_i in order to maximize profit of the data service provider? Thus, a scheduler instance sm_k has as one of its functions to define the execution sequence for the queries in its scheduling queue. This sequence is profit-oriented and based on query interactions.

Let sm_k a scheduler instance, F the scheduling queue of sm_k, $|F|$ the number of queries in F and q'_i a query instance dispatched to sm_k. The execution flow of the scheduling process starts with the arrival q'_i at sm_k. The next step is to add q'_i in F, between positions 1 and $|F|$, inclusive. When q'_i is at the position 1 of F and sm_k is notified that can run a new query, q'_i is moved to position 0 of F, and later performed in the DBMS copy hosted in sm_k. After the execution of q'_i, its result is sent to sm_k. So, sm_k send this result to the dispatcher.

When q'_i arrives at sm_k, its position in F is defined by means of Algorithm 7 (main algorithm of scheduler module), which has q'_i as input and F (the scheduling queue of sm_k) as a global variable. So, let q'_i a query dispatched to sm_k and q'_j the query at the j position in F, such that $1 \leq j \leq |F|-1$. Algorithm 7 starts getting the data structure g, of the type GainsDS, (line 2) which comprises: i) balance gain estimate, ii) interaction gain estimate and iii) the position j of F selected to host q'_i, such that $1 \leq j \leq |F|$. The values of g for the query q'_i are defined by Algorithm 8. Then, let j the position of F selected to host q'_i (line 3), the estimated starting time $est_{q'_i}$ of q'_i is set with the same value of the estimated finishing time of q'_{j-1} (line 4). The next step of this algorithm consists in postponement the queries q'_l, such that $j \leq l \leq |F|-1$, according with the estimated runtime $erunt_{q'_i}$ of q'_i (line 5). Finally, q'_i is inserted at position j of F (line 6).

Algorithm 7: Main algorithm of scheduler module

```
   input  : data structure q'_i
   Data: scheduling queue F
1  begin
2      g ← Algorithm8(q'_i);
3      j ← g.getChosenPosition();
4      q'_i.defineInitialTime(getFinalTime(q'_{j-1}));
5      postponeQueries(j, erunt_{q'_i});
6      F.add(q'_i, j);
7  end
```

Algorithm 8 defines the position j of F to host q'_i. Besides, algorithm 8 estimates balance gain and interaction gain provided by scheduling q'_i at j. This algorithm has q'_i as input, F as a global variable and g as output. Initially, Algorithm 8 supposes that q'_i will be inserted at the position $|F|$ (first free position) of F. Thus, balance and interaction gains are calculated by the formulas 16 and 11, respectively (lines 2 and 3). Furthermore, the variable j is initialized with the position $|F|$ of F (line 4). Let bg_{max} the maximum estimated balance gain provide by scheduling q'_i in F (at one of the F's positions). Algorithm 8 checks each position l, such that $1 \leq l \leq |F|-1$, and identifies which position of F generates the highest interaction gain, considering only the positions with balance gain equal to bg_{max} (lines 5 to 14). If more than one position is found, anyone can be chosen as j. Finally, the data structure g is returned (line 15).

Algorithm 8 calculates the balance gain using the Equation 16, which uses EB_F and $EB_{F\cup\{q'_i\}}(j)$. EB_F is the estimated balance gain provided by running the queries in F. $EB_{F\cup\{q'_i\}}(j)$ is the estimated balance gain provided by running the queries in F after the insertion of q_i at the position j of F (Equation 15).

$$bg(q'_i, F, j) = EB_F - EB_{F\cup\{q'_i\}}(j) \tag{16}$$

Let sm_k a scheduler instance, q'_i the next query to be run at sm_k and F the scheduling queue of sm_k. Algorithm 9 runs q'_i. This algorithm uses the global variable dm, which represents the dispatcher module. Algorithm 9 runs continually by repeating the loop between lines 2 and 6. Initially, it obtains the query q'_i (next to run) (line 3). Next, the query q'_i is executed and its result stored in the variable *result* (line 4). Finally, Algorithm 6 (algorithm for receiving the q'_i's result at the dispatcher) is called (line 5).

Algorithm 8: Algorithm to choose the position of scheduling queue to q'_i be added and the estimates of gains in balance and interaction provided for scheduling q'_i at the selected position

```
   input    : data structure q'_i
   output   : data structure g
   Data: scheduling queue F
 1 begin
 2     bg_max ← bg(q'_i, |F|);
 3     gi_max ← gi(q'_i, q'_{|F|-1}, null);
 4     j ← |F|;
 5     for l ← |F| − 1 to 1 do
 6         if bg_max < bg(q'_i, F, l) then
 7             bg_max ← bg(q'_i, F, l);
 8             gi_max ← gi(q'_i, q'_{l-1}, q'_l);
 9             j ← l;
10         else if bg_max = bg(q'_i, F, l) and gi_max < gi(q'_i, q'_{l-1}, q'_l) then
11             gi_max ← gi(q'_i, q'_{l-1}, q'_l);
12             j ← l;
13         end if
14     end for
15     return newGain(bg_max, gi_max, j);
16 end
```

Algorithm 9: Algorithm to run queries in scheduling queue

```
   Data: reference dm to dispatcher module
 1 begin
 2     while true do
 3         q'_i ← getNextQuery();
 4         result ← runQuery(q'_i);
 5         dm.Algorithm6(q'_i.getId(), result);
 6     end while
 7 end
```

4. EXPERIMENTAL EVALUATION

In order to evaluate QIDMaC, we have chosen a scenario in which queries are executed in a sequential way. The choice for such a scenario was motivated by the discussion about the need of predictability in the current paradigm of optimization problem [Florescu and Kossmann 2009]. The idea is to ensure that the queries are executed in an acceptable time, but not as quick as possible. According to this approach, optimizing for the 99 percentile is more important than for the average. Thus, for instance, in the case that each query has its deadline (SLO), a solution that is optimized by 99 percentile is more appropriate. In this case, each query has to be run in a defined period of time to generate profit to the service provider, otherwise penalties can be applied.

When each query has a maximum time to be run, the use of a concurrent solution can be an inefficient way to schedule queries because of the complexity in predicting their runtime if compared with sequential approaches. Many factors, such as operating system scheduling algorithm and resource utilization impact of each query during its execution, meddle in concurrent solution effectiveness. In fact, they can decrease the workload response time, but it is harder than in sequential solutions to control the query deadline fulfillment or to guarantee the profit in an SLA environment that uses query response time as metric, for example. Therefore, a sequential solution shows to be more feasible in some situations than a concurrent one. As a future work we intend to extend QIDMaC to concurrent scenarios.

4.1 Environment

We implemented a prototype of our framework in Java, which runs on Amazon's EC2 cloud infrastructure in the same availability zone (us-west-1c). Each machine in our system runs on a small instance of EC2. This instance is a virtual machine with a 2.4 GHz Xeon processor, 1.7 GB memory and 160

GB disk capacity. We use the Ubuntu 12.04 operating system and the PostgreSQL 9.3 DBMS with default settings. We used TPC-H benchmark (with approximately 2.4 GB dataset). Five machines were used in these experiments, i.e. one VM controller (VMC) and four VM schedulers (VMSs). Each VMS had a fully replicated dataset and a PostgreSQL instance.

4.2 Experimental Results

With the aim to demonstrate the advantages of using query interactions to improve the decision-making in cloud data management services, QIDMaC framework was evaluated using three approaches proposed in [Siqueira et al. 2012] to measure the queries interactions: IS, DRR and GBA.

We used 20 workloads, which consisted of 5 instances of each TPC-H query type. Thus, one workload had 110 query instances, i.e. 5 instances x 22 query types. During the experiments, different strategies for dispatching and scheduling of queries were evaluated, as described following: **Roundrobin/FIFO** - dispatcher module executes the *roundrobin* algorithm and scheduler module uses *FIFO* approach; **SLA-Tree** - both dispatcher and scheduler modules use the approach proposed in [Chi et al. 2011]; **QIDMaC/IS** - both dispatcher and scheduler modules use QIDMaC together with IS strategy; **QIDMaC/DRR**- both dispatcher and scheduler modules use QIDMaC together with DRR strategy; **QIDMaC/GBA** - both dispatcher and scheduler modules use QIDMaC together with GBA strategy. IS, DRR and GBA are different strategies, proposed in [Siqueira et al. 2012], to calculate the interaction factor.

To measure the strategy effectiveness, it was analyzed the workloads and the balance (revenue - penalty), which is greater than the value provided by the strategy RoundRobin/FIFO. Thus, effectiveness is determined by the ratio between the balance and the total of workloads. Note that the operating cost was the same for all strategies. So, we decided to disregard it for comparison purposes. For this reason, we use the balance and not profit. The balance obtained was measured in a fictitious monetary unit. However, it is possible to use other monetary unit, e.g. dollar.

In the first experiment, the effectiveness was evaluated. Figure 2 shows the effectiveness of each strategy. We can observe that SLA-Tree presented 60% of effectiveness, i.e., the balance of SLA-Tree strategy was greater than that Roundrobin/FIFO strategy in 60% of workloads. Thus, SLA-Tree was better than that the Roundrobin/FIFO in 12 in total 20 workloads executed. This is due to the fact that the SLA-Tree strategy did not exploit the benefits provided by query interactions in dispatcher and scheduler modules. QIDMaC/IS, QIDMaC/DRR and QIDMaC/GBA strategies shows 95%, 90% and 85% effectiveness, respectively.

In the second experiment, performance was measured, i.e., the total time required to process all workloads. Figure 3a shows the performance of each strategy. Roundrobin/FIFO strategy executed all workloads in 2,93 hours and SLA-Tree strategy performed all workloads in 2,85 hours (reduction of 2,5%). The strategies proposed in this work improved the response time: QIDMaC/IS performed all workloads in 2.44 hours (16.71% reduction); QIDMaC/DRR in 2.38 hours (18.64% reduction) and QIDMaC/GBA in 2,53 hours (13.42% reduction).

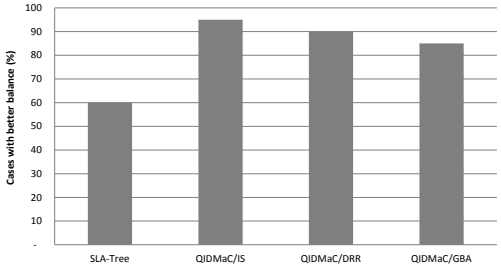
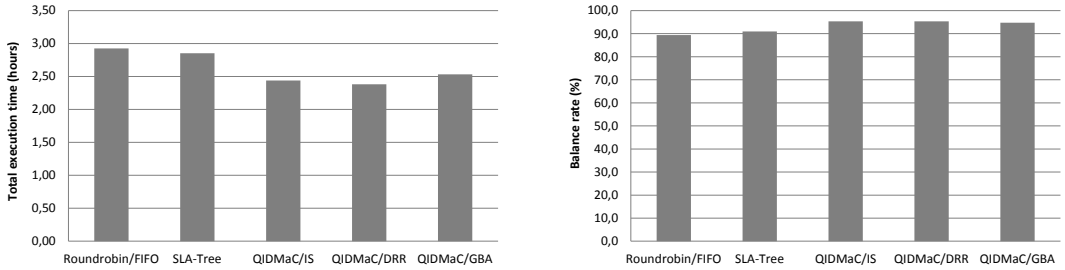


Fig. 2: Comparative of strategies - Effectiveness

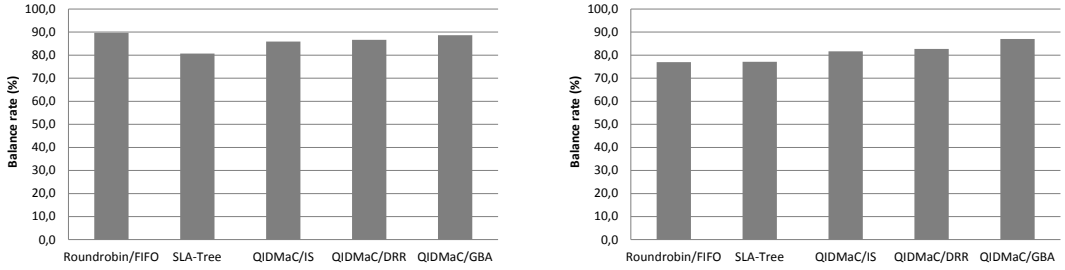
In the last experiment, balance was measured. For each strategy, the total balance was defined, which consists of adding the balance provided by each workload. For comparison, it calculated the maximum balance that could be obtained. This value can be calculated by adding the revenue from each query instance that composes the workloads. This value is present in the SLA defined for the query instance. It was assumed that all query instances were performed according to SLAs. Thus, no strategy had penalties. To calculate the balance for a particular strategy, the total balance of this strategy was divided by the maximum balance. Figure 3b shows the balance of each strategy. Roundrobin/FIFO and SLA-Tree, strategies that do not address query interactions, showed 89.4% and 91% of balance, respectively. QIDMaC/IS, QIDMaC/DRR and QIDMaC/GBA strategies obtained 95,3%, 95,4%, 94,8% of balance, respectively. These results indicate that the use of the query interactions concepts improves processing workloads, providing better performance and, consequently, larger provider's profit.



(a) Performance (b) Balance

Fig. 3: Comparative of strategies - Performance and Balance

In order to analyze the potential and limitations of interaction-aware approaches, we elaborated other two experiments similar to the previous one (Figura 3b). Each one was composed of 20 workloads, which consisted of 5 instances of 9 selected TPC-H query types. Thus, each workload had 45 query instances, i.e., 5 instances x 9 queries. The difference between these two experiments was the chosen query types. In the first one (Figure 4a), almost all data accessed from disk fit in memory. Whereas in the second experiment the data accessed from disk to memory have an expressive size, in GB, if compared with memory size.



(a) Wokload 1 (b) Workload 2

Fig. 4: Comparative of strategies - Benefits and limitations regarding to balance

According to Figure 4a, when all data accessed from disk fit in memory, the compared approaches was worse or equal to Roundrobin/FIFO. This happened because scheduling of all approaches have similar query interaction, since there were few page swapping and, in general, the common data were in memory, independent of query execution order. On the other hand, the second experiment (Figure 4b) had more page swapping because of the size of data accessed by all workloads. In this scenario, interaction-aware dispatching and scheduling approaches defined query execution order in a way to decrease page swapping. So, data service improved query interactions in these approaches and, hence, got a higher profit, if compared with SLA-Tree and Roundrobin/FIFO.

5. CONCLUSION AND FUTURE WORK

In this article, a query interaction-aware framework for providing data service in cloud systems, denoted QIDMaC, has been presented. The main functionality presented by QIDMaC is to deliver required software infrastructure to make data service available in cloud computing environment. More specifically, QIDMaC implements efficient solutions for database query dispatching and scheduling. Moreover, QIDMaC is able to support non-predictable workloads and to take advantage of using query interaction. The proposed framework has been evaluated. For that, the TPC-H benchmark has been used on PostgreSQL. The results show that QIDMaC has potential to improve the efficiency of database service and to increase the profit of cloud data service providers. As future work we intend to extend QIDMaC to solve the resource provisioning problem.

REFERENCES

Abadi, D. J. Data Management in the Cloud: Limitations and Opportunities. *IEEE Data Engineering Bulletin* 32 (1): 3–12, 2009.

Ahmad, M., Aboulnaga, A., and Babu, S. Query Interactions in Database Workloads. In *Proceedings of the 2nd International Workshop on Testing Database Systems*. DBTest, Rhode Island, USA, pp. 11:1–11:6, 2009.

Barker, S., Chi, Y., Moon, H. J., Hacigümüş, H., and Shenoy, P. "Cut me some slack": Latency-aware Live Migration for Databases. In *Proceedings of the 15th International Conference on Extending Database Technology*. EDBT '12. pp. 432–443, 2012.

Chi, Y., Moon, H. J., Hacigümüs, H., and Tatemura, J. SLA-tree: a Framework for Efficiently Supporting SLA-based Decisions in Cloud Computing. In *Proceedings of the 14th International Conference on Extending Database Technology*. EDBT, Uppsala, Sweden, pp. 129–140, 2011.

Comellas, J. O. F., Presa, I. G., and Fernández, J. G. SLA-driven Elastic Cloud Hosting Provider. In *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. PDP, Pisa, Italy, pp. 111–118, 2010.

de Carvalho Costa, R. L. and Furtado, P. A QoS-Oriented External Scheduler. In *Proceedings of the 2008 ACM symposium on Applied computing*. SAC, Ceara, Brazil, pp. 1029–1033, 2008.

Elmore, A. J., Das, S., Agrawal, D., and El Abbadi, A. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. SIGMOD, pp. 301–312, 2011.

Florescu, D. and Kossmann, D. Rethinking Cost and Performance of Database Systems. *SIGMOD Record* 38 (1): 43–48, 2009.

Paton, N. W., Aragão, M. A. T., Lee, K., Fernandes, A. A. A., and Sakellariou, R. Optimizing Utility in Cloud Computing through Autonomic Workload Execution. *IEEE Data Engineering Bulletin* 32 (1): 51–58, 2009.

Rogers, J., Papaemmanouil, O., and Çetintemel, U. A Generic Auto-provisioning Framework for Cloud Databases. In *Proceedings of the IEEE 26th International Conference on Data Engineering Workshop*. ICDE, California, USA, pp. 63–68, 2010.

Schroeder, B. and Harchol-Balter, M. Evaluation of Task Assignment Policies for Supercomputing Servers: The Case for Load Unbalancing and Fairness. *Cluster Computing* 7 (2): 151–161, 2004.

Sharaf, M. A., Chrysanthis, P. K., Labrinidis, A., and Amza, C. Optimizing I/O-Intensive Transactions in Highly Interactive Applications. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. SIGMOD, Rhode Island, USA, pp. 785–798, 2009.

Silva, T. L. C., Nascimento, M. A., Macêdo, J. A. F., Sousa, F. R. C., and Machado, J. C. Towards Non-Intrusive Elastic Query Processing in the Cloud. In *Proceedings of the 4th International Workshop on Cloud Data Management*. CloudDB, Hawaii, USA, pp. 9–16, 2012.

Siqueira, M., Monteiro, J. M., Macedo, J., and de Castro Machado, J. Approaches to Model Query Interactions. In *Proceedings of the 27th Brazilian Symposium on Databases (Short Paper)*. SBBD, São Paulo, Brazil, 2012.

Sousa, F. R. C., Moreira, L. O., Santos, G. A. C., and Machado, J. C. Quality of Service for Database in the Cloud. In *Proceedings of the 2nd International Conference on Cloud Computing and Services Science*. CLOSER, Porto, Portugal, pp. 595–601, 2012.

Tilgner, C. Declarative Scheduling in Highly Scalable Systems. In *Proceedings of the 2010 EDBT/ICDT Workshops*. EDBT, Lausanne, Switzerland, pp. 41:1–41:6, 2010.

Xiong, P., Chi, Y., Zhu, S., Tatemura, J., Pu, C., and Hacigumus, H. ActiveSLA: a Profit-oriented Admission Control Framework for Database-as-a-Service Providers. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. SOCC, Cascais, Portugal, pp. 15:1–15:14, 2011.