Efficient SQL Adaptive Query Processing in Cloud Databases Systems

Clayton Maciel Costa
High-Assurance Software Lab / INESC TEC
Instituto Federal do Rio Grande do Norte / Universidade do Minho
Ipanguaçu, Brasil / Braga, Portugal
clayton.maciel@ifrn.edu.br

Cicília Raquel Maia Leite
Software Engineering Lab
Universidade do Estado do Rio Grande do Norte
Mossoró, Brasil
ciciliamaia@gmail.com

António Luís Sousa
High-Assurance Software Lab / INESC TEC
Universidade do Minho
Braga, Portugal
als@di.uminho.pt

Abstract – **Nowadays, many companies have migrated their applications and data to the cloud. Among other benefits of this technology, the ability to answer quickly business requirements has been one of the main motivations. Thereby, in cloud environments, resources should be acquired and released automatically and quickly at runtime. This way, to ensure QoS, the major cloud providers emphasize ensuring of availability, CPU instance and cost measure in their SLAs (Service Level Agreements). However, the QoS performance are not completely handled or inappropriately treated in SLAs. Although from the user's point of view, it is considered one of the main QoS parameters. Therefore, the aim of this work consists in development of a solution to efficient query processing on large databases available in the cloud environments. It integrates adaptive re-optimization at query runtime and their costs are based on the SRT (Service Response Time) QoS performance parameter of SLA. Finally, the solution was evaluated in Amazon EC2 cloud infrastructure and the TPC-DS like benchmark was used for generating a database.**

Keywords - cloud computing; service level agreement; performance; service response time

I. Introduction

Nowadays, many companies have migrated their applications and data to the cloud due to the benefits of this technology. For example, the applications and data stored in the cloud can be accessed anywhere independent of local software platform. Another important benefit is the significant reduction of costs and time of experimentation and development when compared with local infrastructure because it eliminates the need of one or more physical servers in company increasing the space, minimizing the necessity of specialists for repairs.

In the cloud computing model, the cloud providers have to optimize their profits while servicing several customers. This is obtained recurring to some level of abstraction (virtualization) according to the type of service, such as: storage, processing, bandwidth and active user accounts [1]. To ensure QoS (Quality of Service), there are SLA (Service Level Agreements) associated to the service delivery. The SLA is a formal contract defined between a cloud service provider and its customers that describe the level of service expected from provider. SLAs are output-based in that their purpose is specifically to define what the customers expect to receive. The SLA is composed of several metrics on the levels of availability, functionality, performance, penalties, billing, etc [1]–[3]. This work focuses on the SRT (Service Response Time) performance parameter of SLA, which corresponds to the total time between time that the request/query arrives to the provider and at the time, it completes its execution in the system.

Following this context, adaptive query processing has the ability to dynamically and automatically allocate or release resources (elasticity of resources) during the query runtime. This technique is very important when statistical information about the services available may be minimal and the availability of physical resources may change. This is a typical scenario of cloud environments. However, traditional and adaptive query optimzers' main objective is to reduce response time. Moreover, in the context of cloud computing, users and providers of services expect to get answers in time to guarantee the service SLA.

The performance parameters of a SLA are the most important requirements for most customers when they decide to migrate their applications to the cloud. Because these parameters are directly related to the performance of their applications in the cloud. Therefore, from the user's point of view, they are considered one of the main QoS parameters [4]. Nowadays, one can see that the major cloud providers like Amazon [5] and Google [6] emphasizing availability, CPU instance and cost measure. Therefore, the SRT performance parameter is not completely handled or inappropriately treated in SLA.

The measuring of SRT parameter in SLA is a very complex task because it depends on many system variables, such as request type, database model and current rate system performance. Furthermore, it is common in a cloud environment that the requests rate is highly unpredictable. Therefore, guaranteeing a specific response time for any level of request rate is regarded as a significant challenge to the paradigm of cloud computing. Moreover, the growth of data stored in the cloud makes this challenge ever harder.

Therefore, the aim of this work consists in development of a solution to efficient query processing on large databases available in the cloud environment. It integrates adaptive re-optimization at runtime of the query and their costs are based on the SRT QoS parameter. Moreover, it is restricted to relational database access requests, it has not restriction of elasticity and/or scalability of their algorithms and a non-intrusive approach. Finally, it was evaluated utilizing Amazon EC2 cloud infrastructure small instances type and the TPC-DS [7] like benchmark was used for generating an OLAP database of structured data.

This paper is organized as follows. Section 2 presents related works. Section 3 presents the strategies for adaptive processing of different types of queries in cloud databases systems. Section 4 shows the experimental evaluation. Finally, Section 5 shows the conclusions and future works.

II. Related Works

Currently, several researches have been focused in search of techniques for efficient query processing in the cloud. As shown in Table I, the works in [8]–[17] do not use the strategy of monitoring during requests execution. The algorithm in [18] provides adaptive optimizing the response time of queries. The algorithm partitions and adaptively identifies the best level of parallelism for each query. The authors propose an adaptive provisioning algorithm for only select-range queries and consider variations in performance of VMs (Virtual Machines). However, it does not observe the SLA agreement and does not specify the frequency of the monitoring algorithm during query execution. The work in [4] presents an adaptive SLA-oriented resource manager. However, it only predicts the provisioning of resources and does not check DBMS variables for database access requests, addressing only the level of the application server layer. The approach in [19] uses the strategy of regular monitoring intervals during requests execution and therefore does not consider that VMs may have different performance. In addition, it limits its scope to single pipeline queries (queries without joins). The approach in [20] presents a non-intrusive framework for adaptive queries processing in database implanted in cloud environment. This work observes query response time of the SLA contract; makes adaptive monitoring considering the heterogeneous environment, and therefore, it considers that the VMs may have different performances. However, the scope is limited only to select-range queries.

This way, we can observe that most works in the literature focus on shorter execution time of a query and on the prediction of resources to be used for query through the current system context. These works may not be suitable in highly unpredictable environments on the availability of resources. In turn, others works emphasize on adaptive query processing. However, they present limitations of elasticity and/or scalability in their algorithms, the absence of adaptive monitoring query processing, use of intrusive solutions and/or use proprietary technology and do not use formalisms in defining the QoS parameters in their solutions and as a result, the same service may have different understanding among cloud service providers.

III. Adaptive Query Processing in the Cloud

This Section presents the solution for efficient adaptive processing of different types of queries (database access requests) in cloud environment. This way, it will presented the definition of a request and the strategies of execution for each type of request.

A. Requests

In computational context, a request corresponds a task to be executed by a Web Service sent by a customer who has access to this service. This work focuses on database access requests on OLAP applications in the cloud environment. A request message is a SQL query composed by one or more tables and it can be of different types.

TABLE I. Related Works

Related Work	Adaptive Query Processing	Based on SRT on the SLA contract	Restriction of Query	Provisioning or Release of Resources
[8], [21]	No	Yes	Not restricted	Provisioning of Resources
[9]	No	No	Not restricted	Provisioning of Resources
[18]	Yes	No	Select-range	Provisioning of Resources
[4]	Yes	Yes	Not applied	Provisioning of Resources
[10]	No	Yes	Not applied	Provisioning of Resources
[19]	Yes	Yes	Select-range	Provisioning and Release of Resources
[14]	No	Yes	Not restricted	Not applied
[12], [11]	No	Yes	Not restricted	Provisioning of Resources
[15]	No	No	Not restricted	Provisioning of Resources
[13]	No	Yes	Not restricted	Provisioning of Resources
[16]	No	No	Not restricted	Provisioning of Resources
[20]	Yes	Yes	Select-range	Provisioning and Release of Resources
[17]	No	Yes	Not restricted	Provisioning of Resources

Therefore, to better understanding of proposed solution, the requests were classified between three types, according to level of complexity: (i) **Type 1 Requests** represent the select-range and/or select-aggregation requests. Select-range are the database access requests that will return only tuples that are in a given range of a table. An index can be used to select the tuples. The range is used when a column, key or not, is compared with a constant using: =, <>, >, > =, <, <=, IS NULL, <=>, BETWEEN or IN; (ii) **Type 2 Requests** represent the database access requests that uses one or more of the following operators: cross join, inner join, left outer join, right outer join or full outer join. Finally, (iii) **Type 3 Requests** represent the database access requests that use aggregation, joins, union, grouping and/or nesting operators. They can be UNION, INTERSECTION, EXCEPT, ANY, IN, UNIQUE, EXISTS, NOT EXISTS, GROUP BY, HAVING, ORDER BY or FETCH WITH.

B. Metadata and Performance

It is worth noting that before the effective execution of a request, it is replicated to a metadata server. The metadata has main objective extract, process and store information about the request that will be useful to its execution. Furthermore, the metadata monitors the real-time performance of each slave node with the aim to make estimates query execution. The following are presented main information of metadata:

Request Costs: To estimate the cost of a request, in this work was used the EXPLAIN command that shows query plan chosen by the DBMS optimizer. The query plan or query execution plan is the sequence of operations DBMS performs to run a request. The values obtained does not represent the correct estimated cost if the query is too complex, but it serves as a basis for estimating the request performance. The EXPLAIN command returns the variables: cost, rows and width. The cost estimates are measured in units of disk I/O. An operator that reads a single block of 8.192 bytes (8K) from the disk has a cost of one unit. CPU time is also measured in disk I/O units, but usually as a fraction. For example, the amount of CPU time required to process a single tuple is assumed to be 1/100th (0,01) of a single disk I/O. Finally, the rows variable corresponds the number of tuples to be returned of a request and the width variable corresponds the quantity of bytes of each returned tuple. Therefore, the total cost is the sum of the quantity of disk pages to access the data plus the quantity of returned rows times 0,01, i.e. cost=disk_pages +rows * 0,01.

Request Types: As defined previously, the requests that will executed are classified between three types, according to complexity level. The result of classification are used to trace a request-profile that will used by other requests in search of similar characteristics. Therefore, the EXPLAIN command of DBMS can too be used to obtain these information.

Probability of SRT Violation: Based on the requests of the similar characteristics that executed on the provider, it is calculated the probability of SRT violation. Let PV_{Ri} be the percentage of times that the response time of similar requests was bigger than SRT. If PV_{Ri} exceeds 50%, the query plan will take on a pessimistic approach, which consists to use more computational cost to decrease the probability of SRT violation. If PV_{Ri} does not exceeds 50%, the query plan of R_i request will take on an optimistic approach, which consists to use enough computational cost to execute R_i.

Performance Monitoring: To get the current performance of a slave node in this work were used the variables util and iowait of mpstat and iostat tools. They are very important to identify problems of CPU and device saturation and percentage of CPU time during which I/O requests were issued to the device (bandwidth utilization for the device). In metadata these values of each slave node are updated and stored in the metadata at regular intervals. Finally, whether this percentage is above 80%, the slave node is unavailable for executing requests, because there is too much risk of not meet the expectations of query response, else, on multiprocessor systems is used mpstat tool and through the iowait is checked the each CPU core availability. Case all CPU cores is above 70%, the slave node is unavailable, else, case at least one core is below 70%, the slave node is available to execute requests. In uniprocessor systems, it is checked only global iowait, not being necessary the use of mpstat tool.

C. Query Processing: FlowChart

In summary, Figure 1 shows the flowchart of possible execution plans to execute a request. For **Type 1 Requests**, the requests are partitioned in the initial provisioning and its subqueries are distributed according to the current performance of each slave node in order to have an execution plan that ensures the SRT. For this, it will be used the metadata variables (statistical data). Therefore, during the execution of each partition, the monitoring checks the elapsed time and estimates the probability of SRT violation.

For **Type 2 Requests**, they are initially executed the partitioning of request according to its simple nested loops and if exists, its predicates. Then, each subquery is executed according to the **Type 1 Requests**. After all process, the result is unified in accordance with its joins.

For **Type 3 Requests**, they can be executed using a pessimistic or optimistic approach. The pessimistic approach is used when the PV_{Ri} of similar requests is greater than 50% and the optimistic approach when the PV_{Ri} is less than or equal to 50%.

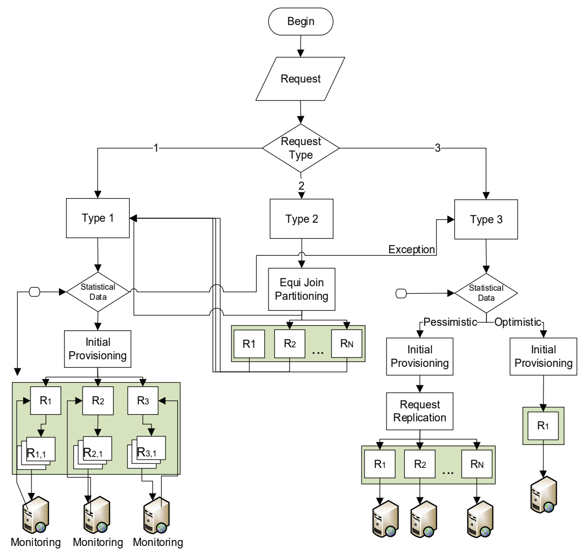

Figure. 1. Flowchart of query processing for each type of request.

The adaptive query processing algorithm (AQP Algorithm) is shown below. For each type of request is used a strategy of partitioning and execution. After its execution, a result of request is presented to customer and for the provider is presented request information, such as SRT violation, elapsed time of request etc. The information of SRT violation is important for the provider to understand the reasons of the violation and to make decisions to reduce the problem.

```
AQP ALGORITHM (R, TR, ET): RETURN RESULT
– ET; //Elapsed Time = RSRT - ET.
– R; //Request
– TR; //Type of Request
– METADATA; //Metadata Class
– SLAVE_NODE[0..i]; //Available Slave Nodes
1.  BEGIN
2.    SWITCH(TR)
3.    CASE 1: //Type 1 Request
4.        IF (R.hasPredicate("WHERE T.pk = <<value>>;")) //Exception
5.          AQP(R,3,ET);
6.        ELSE
7.          Partition[0..i] = METADATA.getSelectedSlaveNode(R, SLAVE_NODE[0..i]);
8.          FOR EACH Partition DO
9.            RESULT += DQM(Partition, ET, 1, SLAVE_NODE[j]);
10.         ENDFOR
11.         RETURN RESULT;
12.       ENDIF
13.       BREAK;
14.   CASE 2: //Type 2 Request
15.       Partition[0..i] = PartitionEquiJoin(R);
16.       FOR EACH Partition DO
```

```
17.             SubResult [0..i] = AQP (Partition,1,ET);
18.         ENDFOR
19.         RETURN JOIN(SubResult);
20.         BREAK;
21.     CASE 3: //Type 3 Request
22.
        SelectedSlaveNodes[0..i]=METADATA.getSelectedSlaveNodes(R,
        SLAVE_NODE);//all nodes > ET
23.         IF (METADATA.getProbability(R) == OPTIMISTIC)
24.             RESULT = DQM(R,ET,3,SelectedSlaveNodes[i]);
25.         ELSE //All slave nodes satisfies ET
26.             RESULT = DQM(R,ET,3,SelectedSlaveNodes[0..i/2]);
27.         ENDIF
28.         RETURN RESULT;
29.         BREAK;
30.     ENDSWITCH
31.     IF(ET > RSRT)
32.         METADATA.setViolation(TRUE);
33.     ENDIF
34. END
```

The monitoring algorithm verifies, periodically, the possibility of a query to be executed before a SRT. Therefore, the DQM (Dynamic Query Monitoring) algorithm reevaluates each subquery at runtime and checks the possibility of SRT violation, whether it is low, and the query continues its execution; otherwise, the query will be re-optimized in AQP algorithm.

The monitoring will check the request execution progress. Whether the performance of slave node decreases, the system can try recovering and meeting the recommended SRT or if the performance of slave node increases, the system can use that to its advantage. Therefore, monitoring is adaptive with non-regular intervals, because the framework uses a strategy is based on following variables: CPU, memory and processing and reading percentage in DBMS of each slave node used by request. Thus, this work considers that slave nodes can have different performance.

The challenge of monitoring algorithm is to monitor in the best time. It should not be so frequent, since original queries would be partitioned into many subqueries. Thus, the overload added can prejudice more than help. Moreover, it should not be infrequent, because if that happens, it may be difficult to make corrections in a timely manner and avoid possible penalties.

DQM uses historical data of similar requests to establish the most efficient number of partitions for monitoring. Thus, the algorithm checks the request selectivity and the current performance of the first slave node in the initial provisioning. When there are no statistical data, by default, if the request selectivity is less than 10.000 tuples, the component will fragment the request within 2 partitions. If it is between 10.000 and 100.000 tuples, the component will fragment the request up to 4 partitions. If the selectivity is greater than 100.000 tuples, the framework will fragment the request up to 8 partitions.

When there are statistical data, the number of partitions and the SRT used in the execution of similar requests are checked in metadata. Thus, the number of partitions for monitoring is chosen based on the similarity of request (selectivity) and SRT. It is important to note that the operations will be realized in the metadata and will be available now that is required by the request.

The summary of DQM component algorithm is shown below. As presented, Type 1 and 3 requests use different strategies. For **Type 1 Requests**, the DQM uses monitoring and adaptive query processing and for **Type 3 Requests**, it does not use adaptive query processing, it uses greedy algorithm in optimistic approach and the fastest execution in set of slave nodes in pessimistic approach. For the better understanding, the next section we present samples/examples of the scheduling, partitioning and monitoring algorithms.

```
DQM ALGORITHM (R, ET, TR, SLAVENODES): RETURN RESULT
–  R; //Request.
–  ET; //Elapsed Time: RSRT - ET
–  TR; //Type of Request.
–  SLAVENODES; //Slave Node to execute R.
1.  BEGIN
2.      SWITCH(TR)
3.      CASE 1: //Type 1 Request
4.          Partition[0..i] = Metadata.Partitioning(R);
5.          FOR EACH Partition DO
6.
    IF((RESULT+=EXECUTE(Partition,SLAVENODES[0])).getElaps
    edTime()>T2R)
7.              AQP (MERGE(Partition[j..i]),1, ET);
8.          ENDIF
9.          ENDFOR
10.         BREAK;
11.     CASE 3: //Type 3 Request
12.         //optimistic approach: SLAVENODES.getLength() returns 1.
13.         RESULT=EXECUTE(R,SLAVENODES[0..i]));
14.         BREAK;
15.     ENDSWITCH
16. RETURN RESULT;
17. END
```

D. Query/Subquery Scheduler

To scheduler of the query is responsible for distributing the partitions of a request to each slave node available based on its performance. To do this, Let $T2R_{SN}$ the *Tuple Read Rate*, the estimated time in seconds for a slave node to process a quantity of tuples:

$$T2R_{SN} = \frac{rows * 1000}{cost * Svctm} \tag{1}$$

where $rows$ corresponds the number of tuples to be returned of a request, $cost$ is estimated in units of disk I/O and $Svctm$ the average service time (in milliseconds) for I/O requests that were issued to the device of a slave node. This last parameter can be obtain through iostat tool. To better understanding, consider the R request with SRT received by a cloud provider:

```
Select * // ← R
From Table T;
```

Consider that SRT is 100 seconds and through the Explain command we have the cost = 368 and rows = 12.000. Moreover, consider that SN1 is an available slave node and it has Svctm = 13 milliseconds. Thus, $T2R_{SN1}$ of SN1 presents read rate of 250 tuples/seconds. Thus, SN1 ensures the SRT because it was estimated that SN1 in 100 seconds could process 25.000 tuples.

It is worth noting that equation do not consider CPU overhead as well as the use of DBMS cache. However, it presents an estimate used only in the initial provisioning. Thus, at query processing, the $T2R_{SN}$ is calculated by dividing the number of rows retrieved (RT) by the time to retrieve them (TRT):

$$T2R_{SN} = \frac{RT}{TRT} \qquad (2)$$

For complex queries, the strategy is similar to select-range queries. However, the rows variable is obtained by sum the quantity of tuples accessed by each query execution plan operator. Even if more than one operator uses these tuples and/or if these tuples are not part of the result. As well as select-range queries, this work considers that all access to a tuple block (on disk or temporary data pagination) is a cost I/O.

It is worth noting that this estimate does not consider the CPU overhead. However, the overhead of temporary data pagination is considered, since it does not distinguish the repetition of tuples during each step of the query execution plan.

Therefore, if we have the current speed of tuples read rate per second of a slave node, it is possible to partition a request in accordance with the estimated time to execute the request on each node. For not violate the SRT, the sum of the times for each partition to execute a subquery, according to the times of each slave node (SN), it has to be less than the SRT:

$$SRT_{Ri} \geq T2R_{SN1} + T2R_{SN2} + \cdots + T2R_{SNk} \qquad (3)$$

In this work, the partitioning strategies depends on the type of request and we consider that the all tables are clustered by primary key.

For example, assume that a cloud provider receives the following select-range request R with SRT:

```
SELECT * // ← R
FROM table T
WHERE T.pk >= 1000 and T.pk < 5000;
such that pk is the primary key of table T.
```

Considering that primary key values of T are sequential, without gaps between values, then we can extract rows = 4.000 tuples. Besides, consider that SRT is 100 seconds and that initial provisioning is a single slave node (SN1) such that the current moment $T2R_{SN1}$ = 20 tuples/sec.

Consequently, the initial provisioning using only SN1 will bring a penalty to be paid by the provider because it was estimated that SN1 in 100 seconds will process in 2.000 tuples. In this case, it is necessary to allocate a new slave node (SN2) to help. Assume that $T2R_{SN2}$ = 10 tuples/sec then only 1.000 tuples can be processed in 100 seconds. Then, a new slave node (SN3) is required to process the request. Then, consider $T2R_{SN3}$ = 10 tuples/sec.

At this point, it is possible that three slave nodes are sufficient to process R and ensure the SRT. R is rewritten in three subqueries: R_1, R_2 and R_3, the first one is executed in SN1, the second one in SN2 and the third one in SN3, respectively. Note that in this case a virtual partitioning is used (i.e. we partition using the predicate of the primary key) to divide R in R_1, R_2 and R_3.

```
SELECT * // ← R1
FROM table T
WHERE T.pk >= 1000 and T.pk < 3000;
```

```
SELECT * // ← R2
FROM table T
WHERE T.pk >= 3000 and T.pk < 4000;
```

```
SELECT * // ← R3
FROM table T
WHERE T.pk >= 4000 and T.pk < 5000;
```

Using only three slave nodes do not guarantee that the quality defined in SRT will be met, because the cloud environment is unstable and the performance of nodes can change during the queries execution. Therefore, a proactive approach based on statistical data in metadata indispensable use. For this, the query are partitioned in such a way that the performance of the nodes can be monitored at a frequency that allows other nodes to be added when necessary in order to ensure the SRT.

An important issue is the monitoring frequency. If too frequent, the original queries would have to be partitioned into many subqueries. Thus, the overload added could prejudice more than help. If monitoring is infrequent, it may be difficult to make corrections in a timely manner and avoid possible penalties.

The partitioning process uses historical data about the request containing information about how long it was necessary to process similar requests (same type of request), including the number of partitions used. From this information, it is possible to monitor efficiently the request execution.

Consider for example, in similar requests, 2 partitions were used for each partition of the initial provisioning. Then, R_1 is partitioned in two requests:

```
SELECT * // ← R1,1
FROM table T
WHERE T.pk >= 1000 and T.pk < 2000;
```

```
SELECT * // ← R1;2
FROM table T
WHERE T.pk >= 2000 and T.pk < 3000;
```

When $R_{1,1}$ is done, it have the first opportunity to monitor the query execution performance in a non-intrusive way. Consider that 70 seconds were spent to execute $R_{1,1}$. This means that the performance $T2R_{SN1}$ was below of predicted, which leads to a completion time with the expected processing of the next subquery of 140 seconds. However, this value is above the SRT. Thus, it starts a revision of the initial provisioning for that SRT can be satisfied. Before reviewing, the remaining partitions will be merged in a single query.

In this case, a solution relocates remain subquery to another slave node. Suppose a new slave node (SN4) is such that $T2R_{SN}$ = 30. Thus, all the 1.000 remaining tuples can be read by SN4 in 30 seconds in the best-case scenario, and that does not lead to a violation of SRT. To monitor the request execution, it is again partitioned into two, each of the following way:

```
SELECT * // ← R1,2,1
FROM table T
WHERE T.pk >= 2000 and T.pk < 2500;
```

```
SELECT * // ← R1,2,2
FROM table T
WHERE T.pk >= 2500 and T.pk < 3000;
```

Consider that performance is stable and it is able to finish their workload on schedule. Thus, the same strategy can be applied in the processing of R_2 in SN2 and R_3 in SN3. This partitioning method using the primary key as the partitioning attribute is the same for similar select-range requests and similar requests with aggregation.

For requests with joins (**Type 2 Requests**), it rewrites the query separating all tables of FROM clause. Consider that a cloud provider receives the following trivial select-join request R with SRT:

```
SELECT * // ← R
FROM table T1, table T2
WHERE T1.fk = T2.pk;
such that T1.fk the foreign key referenced by the primary key T2.pk.
```

The request R is rewritten in two subqueries, R_1 and R_2:

```
SELECT * // ← R1
FROM table T1;

SELECT * // ← R2
FROM table T2;
```

In this case, R_1 and R_2 will be executed utilizing strategies of **Type 1 Requests**. Thus, it is used the partitioning methodology described for **Type 1 Requests**. As well as the monitoring and provisioning of slave nodes to execute the rewritten query is made the same way. Finally, after the execution of all partitions the slave node that executed R_1 makes the join to present the result.

For complex requests and others not shown here (**Type 3 Requests**), it adopts the strategy of seeking the set of available slave node with $T2R$ enough to process the request that ensures the SRT. This strategy are adopted before due to the highly complexity of estimates costs and making partitioning. Therefore, this type of request does not use monitoring nor adaptive partitioning during query execution. Consider the following a complex request with SRT is 100 seconds and rows = 200.000 tuples.

In the optimistic approach, the greedy strategy is adopted, in which only one slave node executes the request and it is expected that it ensure the SRT. Now consider three slave nodes, SN1, SN2 and SN3, with $T2R_{SN1}$ = 4.000 tuples/sec, $T2R_{SN2}$ = 2.000 tuples/sec, $T2R_{SN}$ = 1.000 tuples/sec, respectively. In this case, the algorithm using greedy strategy chooses SN1 because it was the first and enough in such a way to execute R within the SRT. In pessimistic approach, the algorithm strategy is to choose half the number of slave nodes available with the highest $T2R$ to execute request R. Consider four available slave nodes, SN1, SN2, SN3, SN4, with $T2R_{SN}$ = 4.000 tuples/sec, $T2R_{SN2}$ = 2.000 tuples/sec, $T2R_{SN3}$ = 1.000 tuples/sec and $T2R_{SN4}$ = 1.500 tuples/sec, respectively. Then, the algorithm replicates the request R for SN1 and SN2, in such a way that least one can ensure the SRT.

In the worst-case scenario, if there is no slave nodes that meets the SRT, the closest node to meet the SRT in terms of $T2R$ is selected. Monitoring the slave node to process this type of request is made after its processing, when it is checked violation or not of SRT and metadata updates its information.

IV. Experimental Evaluation

A. Experimental Environment

The strategies presented was implemented in using the Java language and concurrent programming with threads and API based on OpenMP (Open Multi-Processing) [22]. It was deployed in the Amazon EC2 cloud infrastructure using small instances. Due to the limitations of Amazon, it was used 20 VMs, each one with an Intel Xeon Processor with turbo up to 3.3GHz, 1.7 GB of main memory and 160 GB of disk storage.

It was created an AMI (Amazon Machine Image) of a VM with the database. This image allows startup new VMs quickly. The Amazon EBS (Elastic Block Store) was used to storage the AMI. Each VM runs the Ubuntu 12.04 operating system and PostgreSQL 9.3 DBMS. This work focuses on OLAP applications with very large and complex database. Thus, the TPC-DS like benchmark was used to generate a database of approximately 13 GB, fully replicated in all VMs. Therefore, the database generated represents the customer data.

B. Methodology

The experiments aim at showing the efficiency of queries processing strategies proposed in this work. This way, it will check the ability to avoid penalties associated with SRT violation and the elasticity of the algorithm in according to the number of VMs allocated when processing queries. For **Type 1** and **Type 2 Requests**, the experiments consisted to stress the system using 10 workloads and each workload having 10 queries of the same type. For **Type 3 Requests**, as the strategy is predictive and queries are complex, 5 workloads were used, each workload having 5 queries of the same type. Finally, the experiments were performed using 10 workloads and each workload having 10 queries of several types of requests.

The minimum amount of required machines is a complex task. Therefore, previous tests were performed using a fixed number of VMs. Thus, the minimum number of machines was found for the workload of the experiments. However, if new workloads arrive to the system, it will be necessary to perform extensive experiments again to obtain a new configuration of service provider. The arrival time of the queries workloads was disposed uniformly varied distribution (non-uniform distribution): each workload arriving at a random time intervals between 10 and 60 seconds. This distribution is closer to real environments, since the unpredictability of workloads arriving to the system and performance variation are characteristics of cloud environments. Moreover, it was used different values of SRT, from the most restricted to the most relaxed.

Seeking more accurate results for each type of request, experiments were repeated 10 times. Finally, to eliminate any possible interference between successive experiments, in particular, effects of other queries already executed, the OS cache was deleted and the DBMS has been restarted before executing the queries workloads again.

For each experiment, the number of virtual machines used are observed in accordance with time. To calculate the computational cost it was enough to observe the number of virtual machines used by each query. Finally, the query runtime is measured according to the strategies described in previous Section.

C. *Results*

For **Type 1 Requests**, the Figure 2 shows experiments of select-range queries with the arrival of workloads following the non-uniform distribution. The graphs present the number of VMs used by time in seconds and the SRTs used were 80, 100 and 120 seconds. The queries predicate may be on a non-key attribute or on a key attribute.

It is important to emphasize at this point that when the attribute is not a primary key, our strategy scans all tuples of the table, and i.e. all tuples are checked to verify whether they satisfy the predicate. Thus, this type of queries requires more processing time than the select-range that have a predicate on key attribute. Consequently, this causes the increase of VM computational cost, since the response time is higher. According to the results, it can see the increase and decrease of the workloads on the system and the elasticity on the number of virtual machines allocated to execute the queries. When the SRT is more restricted, the computational cost is higher or equal to the computational cost of the most relaxed SRT, this happens to avoid penalties. Moreover, the computational cost is higher when the workloads arrive at random times (non-uniform distribution) if compared to uniform distribution. We believe that the system may not recover quickly when there is an unexpected overload resource, and seeking quick reaction to execute the queries, the algorithm allocates more VMs to execute the workload in SRT time. Consequently, the computational cost increases.

For **Type 2 Requests**, experiments were realized in queries with or without SQL predicate. Then, the primary difference between these types (**Type 1** and **Type 2 Requests**) is the query partitioning and merge of their results. Figure 3 shows the experiments with the arrival of workloads following the non-uniform distribution. As in previous experiments, the SRT was varied, the most restricted SRT was 130 seconds and the most relaxed SRT was 180 seconds. The partitioning time of queries was not considered because the low complexity of queries used in the experiments. However, it was observed that the merging of the results cause a higher time to execute queries, approximately 10% more than the select-range requests. Finally, it is important to observe that in the most restricted SRT, the ninth workload reached the limit of the infrastructure service provider. For **Type 3 Requests**, the experiments were realized with complex queries obtained from the TPC-DS. As shown in previous sessions, the following graphs show the number of VMs allocated by time in seconds and the SRT was varied, the most restricted SRT was 800 seconds and the most relaxed SRT was 1200 seconds. However, due to the complexity of these queries and the limit of VMs available in service provider, it was used only 5 workloads and each workload having 5 complex queries. According to proposed strategy of this work, the experiments stressed the system searching a VM that could execute successfully a query in SRT time (optimistic approach) or executing a query over a set of VMs that one VM could execute successfully the query in SRT time (pessimistic approach). Therefore, it is not used monitoring nor adaptive partitioning during query execution. Figure 4 shows the results of experiments following the non-uniform distribution of workloads. We can observe that due to the strategy used in this work a large number of VMs are used since the first query workload. In addition, in accordance to the strategy presented, the algorithm chooses through the metadata the optimistic or pessimistic strategy for executing a query and after its execution the metadata are updated. However, we believe that the decrease in the use of virtual machines after the third workload happened due to the algorithm starts to use more often the optimistic approach. Consequently, the queries were being executed successfully. Moreover, it can observe that due to the complexity and selectivity of the queries, there is a greater overhead for the ending its results.

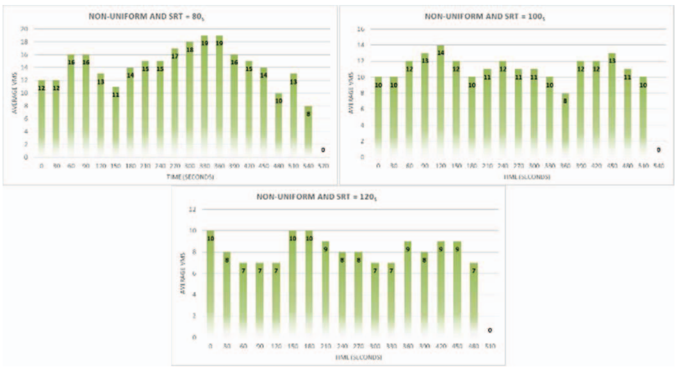

Figure. 2. Type 1 Requests (Select-Range): average virtual machines used for workloads randomly arriving between 10 and 60 seconds for the SRTs: 80, 100 and 120 seconds.

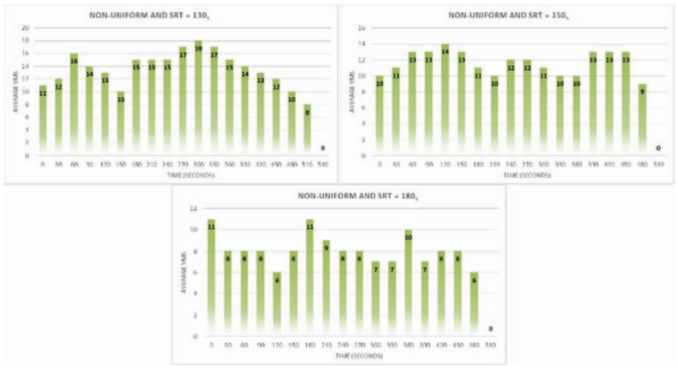

Figure. 3. Type 2 Requests: average virtual machines used with workloads randomly arriving between 10 and 60 seconds for the SRTs: 130, 150 and 180 seconds.

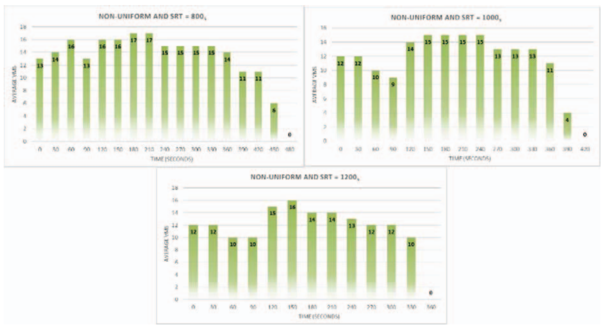

Figure. 4. Type 3 Requests: average virtual machines used with workloads randomly arriving between 10 and 60 seconds for the SRTs: 800, 100 and 1200 seconds.

Finally, experiments were realized using **all requests types** over the same queries workload. According to the strategies proposed in this paper, it was obtained similar results to previous ones. The main overhead is of the algorithm having to classify each query to be executed. After classifying the query, the query is executed according to the already mentioned strategies. Figure 5 shows the experiments with the arrival of workloads non-uniform distribution. The graph show the number of VMs used by the time in seconds. However, unlike previous experiments, each query after classification has a different SRT, according to their type of request. For several moments, it can see the limit of the provider's infrastructure is reached; however, it has not been exceeded. Thus, it can see the increase and decrease in workloads due to elasticity in the number of allocated virtual machines to execute all queries. It is important to observe that no penalty occurred with all queries. Finally, the results of all experiments shown that the proposed solution reacts to the resources variation of the environment and to different sizes of workloads. The strategies ensured that the SRT was satisfied in a non-intrusive and automatic way.

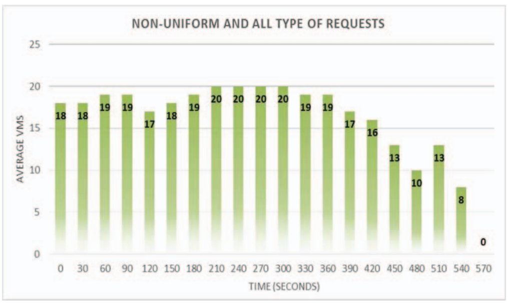

Figure. 5. All Type Requests: average virtual machines used with workloads randomly arriving between 10 and 60 seconds.

V. CONCLUSIONS

In this work, it was presented partitioning, monitoring and provisioning strategies for adaptive processing of different types of queries (database access requests) in cloud environment. The strategies were implemented in a framework and the experiments were evaluated in Amazon EC2 cloud infrastructure. This work focuses on OLAP applications because this kind of environment the adaptive processing produces positive effects at query runtime.

Given the increase and decrease of the workloads, it can see the elasticity in the number of virtual machines allocated by the methods proposed to execute queries. Furthermore, results show that the solution reacts to the resources variation of the environment and to different sizes of workloads. A solution ensures that the SRT is satisfied in a non-intrusive and automatic way. Finally, our proposal was effective to avoid the penalties in the execution of queries and the SRT was satisfied in all experiments without incurring penalties. As future work, we intend to develop adaptive strategies for more types of queries. Moreover, we intend to improve the cost model involving others SLA parameters, such as resiliency, throughput and efficiency, since they are important measures to evaluate the performance of services in cloud infrastructures.

ACKNOWLEDGMENT

This work was supported by Federal Institute of Rio Grande do Norte, University of Minho and Institute for Systems and Computer Engineering, Technology and Science – INESC TEC.

REFERENCES

[1] CSMIC, "Service Measurement Index Introducing the Service Measurement Index (SMI)," no. July, pp. 1–8, 2014.
[2] S. K. Garg, S. Versteeg, and R. Buyya, "A framework for ranking of cloud computing services," *Futur. Gener. Comput. Syst.*, vol. 29, no. 4, pp. 1012–1023, 2013.
[3] V. C. Emeakaroha, M. a S. Netto, R. N. Calheiros, I. Brandic, R. Buyya, and C. a F. De Rose, "Towards autonomic detection of SLA violations in Cloud infrastructures," *Futur. Gener. Comput. Syst.*, vol. 28, no. 7, pp. 1017–1029, 2012.
[4] W. Iqbal, M. Dailey, and D. Carrera, "SLA-Driven Adaptive Resource Management for Web Applications on a Heterogeneous Compute Cloud," in *1st International Conference on Cloud Computing (CloudCom '09)*, 2009, pp. 243–253.
[5] "AWS EC2 Service Level Agreement," 2015. [Online]. Available: http://aws.amazon.com/ec2-sla. [Accessed: 15-Jun-2015].
[6] D. Sanderson, *Programming Google App Engine:*, 2nd ed. O'Reilly Media | Google Press, 2012.
[7] Transaction Processing Performance Council, "Tpc Benchmark^{TM} Ds," 2012.
[8] J. Guitart, D. Carrera, V. Beltran, J. Torres, and E. Ayguadé, "Dynamic CPU Provisioning for Self-managed Secure Web Applications in SMP Hosting Platforms," *Comput. Netw.*, vol. 52, no. 7, pp. 1390–1409, 2008.
[9] Amazon Web Services, "Auto Scaling: Developer Guide," 2015.
[10] J. Rogers, O. Papaemmanouil, and U. Cetintemel, "A generic auto-provisioning framework for cloud databases," in *IEEE 26th International Conference on Data Engineering (ICDE'10)*, 2010, pp. 63–68.
[11] H. Kllapi, E. Sitaridi, M. M. Tsangaris, and Y. Ioannidis, "Schedule Optimization for Data Processing Flows on the Cloud," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, 2011, pp. 289–300.
[12] J. Zhao, X. Hu, and X. Meng, "ESQP : An Efficient SQL Query Processing for Cloud Data Management," in *2nd International Workshop on Cloud Data Management (CloudDB'10)*, 2010, pp. 1–8.
[13] Y. Chi, H. J. Moon, H. Hacıgümü, J. Tatemura, H. Hacıgümüş, and J. Tatemura, "SLA-Tree: A Framework for Efficiently Supporting SLA-based Decisions in Cloud Computing," in *14th International Conference on Extending Database Technology (EDBT/ICDT '11)*, 2011, p. 129.
[14] C. Curino, E. P. C. Jones, S. Madden, and H. Balakrishnan, "Workload-aware Database Monitoring and Consolidation," in *ACM SIGMOD International Conference on Management of Data*, 2011, pp. 313–324.
[15] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, "A Cost-Aware Elasticity Provisioning System for the Cloud," in *31st International Conference on Distributed Computing Systems (ICDCS'11)*, 2011, pp. 559–570.
[16] J. Cerviño, E. Kalyvianaki, J. Salvachúa, and P. Pietzuch, "Adaptive provisioning of stream processing systems in the cloud," in *IEEE 28th International Conference on Data Engineering Workshops (ICDEW)*, 2012, pp. 295–301.
[17] R. Mian, P. Martin, and J. L. Vazquez-Poletti, "Provisioning data analytic workloads in a cloud," *Futur. Gener. Comput. Syst.*, vol. 29, no. 6, pp. 1452–1458, 2013.
[18] Y. Vigfusson, A. Silberstein, R. Fonseca, B. F. Cooper, and R. Fonseca, "Adaptively Parallelizing Distributed Range Queries," in *VLDB Endowment*, 2009, vol. 2, no. 1, pp. 682–693.
[19] D. Alves, P. Bizarro, and P. Marques, "Deadline Queries: Leveraging the Cloud to Produce On-Time Results," in *IEEE 4th International Conference on Cloud Computing*, 2011, pp. 171–178.
[20] T. L. Coelho Da Silva, M. a. Nascimento, J. A. F. De Macêdo, F. R. C. Sousa, and J. C. Machado, "Non-intrusive elastic query processing in the cloud," *Comput. Sci. Technol.*, vol. 28, no. 6, pp. 932–947, 2013.
[21] Í. Goiri, F. Julià, J. O. Fitó, M. Macías, and J. Guitart, "Supporting CPU-based guarantees in cloud SLAs via resource-level QoS metrics," *Futur. Gener. Comput. Syst.*, vol. 28, no. 8, pp. 1295–1302, 2012.
[22] OpenMP.org, "About the OpenMP ARB," 2013. [Online]. Available: http://openmp.org/wp/about-openmp/.