Global Journal of Computer Science and Technology: B
Cloud and Distributed
Volume 14 Issue 4 Version 1.0 Year 2014
Type: Double Blind Peer Reviewed International Research Journal
Publisher: Global Journals Inc. (USA)
Online ISSN: 0975-4172 & Print ISSN: 0975-4350

A Novel Approach for Elastic Query Processing in the Cloud

By CH. Pavani, B.Naga Sai & Dr. K. V. Daya Sagar
K L University, India

Abstract- Cloud computing is a promising model of serviceoriented computing. One major advantage of cloud computing is its elasticity, i.e., the system's capability to supply and take away resources dynamically at runtime. For that, it's essential to design and implement a systematic and effective technique that takes complete advantage of the system's potential flexibility. This paper presents a non-invasive approach that monitors the performance of relational database management systems in cloud infrastructure, and dynamically makes choices to maximise the effectiveness of the provider's environment whereas still satisfying specified service level agreements" (SLAs).

Keywords: *elasticity, query processing, non-intrusive, service level agreement.*

GJCST-B Classification : *C.2.4 D.4.7*

Strictly as per the compliance and regulations of:

A Novel Approach for Elastic Query Processing in the Cloud

CH. Pavani[α], B.Naga Sai[σ] & Dr. K. V. Daya Sagar[ρ]

Abstract- **Cloud computing is a promising model of service-oriented computing. One major advantage of cloud computing is its elasticity, i.e., the system's capability to supply and take away resources dynamically at runtime. For that, it's essential to design and implement a systematic and effective technique that takes complete advantage of the system's potential flexibility. This paper presents a non-invasive approach that monitors the performance of relational database management systems in cloud infrastructure, and dynamically makes choices to maximise the effectiveness of the provider's environment whereas still satisfying specified service level agreements" (SLAs).**

Keywords: elasticity, query processing, non-intrusive, service level agreement.

I. Introduction

Cloud Computing is a platform consists of a commonly very large number of computers responsible for data computing and storage. Such large computing re-sources are combined to serve multiple consumers using a multi holder model, with different physical and virtual resources dynamically assigned and reassigned according to the users' demands. Consider a cloud computing environment as a set of virtual machines (VMs) which may be assigned to different physical machines (PMs), different VMs may return different performances due to many factors, e.g., concurrency with other processes on the PMs.

Cloud computing elasticity enables the system to provide and remove resources according to the application's needs in real-time. providing suitable cloud elasticity on demand is not a frivolous matter. A cloud computing environment is apt to several factors that may influence its performance, including different types of virtual systems provided by the service, different time zones, and demand variation .To providing cloud computing elasticity requires monitoring closely the system's demand for resources in order to decide when to add or to remove resources.

When users purchase figure out time from a cloud provider, commonly both sides, the user and provider agree on the quality of service, via a service level agreement (SLA), which may be composed of the following parameters:

Author α σ: Department of Electronic and Computer Engineering, B.Tech K L University. e-mail: pavanichirasani@gmail.com, bommareddysainani@gmail.com

Author ρ: Associate Professor, Department of Computer Science & Engineering, K L University. e-mail: sagar.tadepalli@gmail.com

Revenue: monetary value paid by the user to the provider for the computing time.

Operating Costs: monetary value paid by the user to the provider for the computing resources allocated for processing the user's workload.

Service Level Objective (SLO): It is associated with a user-defined metric which must be satisfied by the provider. e.g., response time, throughput, availability.

Penalty: monetary value paid by the provider to the user for not satisfying the SLO.

The advantage of cloud providers to accordingly monitor and scale their resources, e.g., VMs, in real-time as a function of the current workload, in order to lower their computing cost while satisfying all current SLAs and minimizing penalties as much as possible. This is the very problem we address.

In this paper we aim at continuously monitoring a DBMS's performance and automatically minimizing the VMs used for query processing while minimizing potential SLO violations. We use query processing time as the contracted SLO metric.

Our resource provisioning approach does not assume that the number of VMs is fixed, or that VMs yield the same performance, nor we assume we can conclude the workload beforehand.

Using our elastic solution to ensure those, the immigrate process of applications to the cloud can be made directly. Our approach uses the relational data model and works with full reproduction. Thus, each set up VM has a DBMS with a complete copy of the database. Our solution does not partition the data, but the query.

We apply virtual partitioning to divide the query into sub queries to be processed on allocated virtual machines.

In that paper only *select-range* queries based on key attributes were investigated. This paper extends by showing how to address *select-range* queries on non-key attributes as well as *aggregation* queries.

Main contributions are :

- a non-intrusive, automatic and adaptive performance monitoring technique for DBMSs on the currently allocated VMs
- a pragmatic approach which dynamically aims at providing the smallest set of VMs capable of satisfying each query's SLO and thus the user's SLA in general.

One of the main advantages of the approach that we propose is that it may be easily applied to cases where the user already has its applications using relational database and ambition to deploy it in a cloud infrastructure, without the need of re architect ring the applications which are especially based on RDBMS technology.

II. Related Work

An adaptive approach for provisioning VMs for the use of a distributed stream processing systems (DSPS)in the cloud. The proposed provisioning algorithm uses a black box approach, i.e., it is independent of the specifics of the queries running in the DSPS. It scales the number of VMs used entirely based on measurements of input stream rates. It detects an overload condition when a decrease in the processing rate of input data occurs because of discarded data tuples due to load dropping. The algorithm is invoked periodically and calculates the new number of VMs that are needed to support the current workload demand however, the paper does not specify how often this algorithm is invoked. We do not focus only on resource provision, and our contribution is also an adaptive monitoring, and re provisioning if necessary, during querying processing.[1]

The authors of proposed Kingfisher, a cost-aware system that tries to minimize the customer-centric cost, the cost of renting servers while meeting the application's SLA. It solves an integer linear program to account for both the infrastructure and transition cost deriving appropriate elasticity decisions under each workload change. Kingfisher uses a proactive approach to know when to provision as well as an ideal workload predictor that uses statistics gathered by the monitoring engine to derive estimates of future workload. Our approach differs from this approach in many ways: we use a cloud-provider-centric approach, and we do not assume a workload predictor, rather than the system's performance is continuously monitored.[2]

The problem of provisioning resources in a public cloud to execute data analytic workloads is exited. The algorithm presented an lazes the space of possible structure for the input workload based on conclude costs of the structure. The algorithm tries to find a structure where resource costs are minimized while the SLA associated with the workload is met. The cost model presented is similar to ours. However, considering that the performance of a particular structure can therefore degrade and SLAs are violated, it might be necessary to change the resources allocated to the application. The paper does not present a dynamic provisioning as we have done in our work.[3]

This paper presents an adaptive method to optimize the response time of range queries in a distributed database. The algorithm partitions and adaptively identifies the best level of parallelism for each query, since choosing the maximum level of parallelism is not necessarily the best strategy to optimize a query's performance. If a query is sent to too many storage hosts, it can saturate a single client by returning results faster than the client can consume them. That work, similar to ours, proposes an adaptive provisioning algorithm for range queries and considers possible variation in VM performances, but it differs from our work as it does not have an SLA to observe and does not specify how often the algorithm is invoked.[4]

III. Proposed Work

Problem Formulation

Let TRQ denotes the estimated total time needed to process a query Q and let SLOQ be the agreed time to process a given query Q as per the SLA. If p is the cost, per unit of time, of failing to satisfy SLOQ, we can define the provider's penalty (cost) as:

$$pp_Q = \max\{(TR_Q - SLO_Q) * p, 0\} \quad (1)$$

Let c_{vm} be the cost, per unit of time, for using a VM which contains a full model of a relational database and let $n_Q(t)$ be the number of VMs allocated to execute Q at time t, where time is discretized in billable units. Then, we can define the computing cost for running Q as:

$$cc_Q = \sum_{t=1}^{TRQ} (nQ(t) * cvm)$$

allocating as many VMs as possible will output an optimally minimum provider's penalty cost, but will increase its computing cost. Like wise , allocating a single VM will minimize the provider's computing cost, but will very likely output a potentially large penalty cost. given a query Q, we need to obtain, for each time point t, the number of VMs (n_Q (t)) minimizing Q's cost

$$(pp_Q + cc_Q)$$

This problem definition captures the elasticity of the cloud environment, namely that in different points of time a different number of VMs may be sufficient to process a query with respect to its SLO.

a) Query Definition

A preliminary version of this paper appears in that paper only select-range queries based on Key attributes were investigated. A select-range query is defined as follows:

SELECT * FROM table T1 WHERE T1.attr >= Vs and T1.attr <Vf

where T1.attr denotes an attribute of table T1 and Vs and Vf are integer values.

For instance scan queries:

SELECT *FROM table T

can be commonplace rewritten as the following two queries:

SELECT MIN (T1.pk) into Vs, MAX (T1.pk) into Vf FROM table T1;

SELECT MIN (T1.pk) into Vs, MAX (T1.pk) into Vf FROM table T1;

where T1.pk denotes the primary key attribute of T1.

We can also handle aggregation queries and aggregation over range:

SELECT OPER (T1.attr)
FROM table T1;

where OPER is an aggregate operator (SUM, for example) and T1.attr denotes an attribute of a table T.

b) Motivating Example

i. *Select-Range Query*

Assume that the following single select-range queryQ with SLO_Q is received by the cloud provider.

SELECT * // <--- Q

FROM table T1 WHERE T.pk >= 0 and T.pk < 4000;

where T1.pk is the primary key of table T1.

Assuming that T's primary key has no gaps, Q_s=4000 tuples. Further let us assume that SLO_Q is 100 seconds and that our initial provisioning is one single machine vm_0 such that RR_0 = 30 and consequently NT_0^Q = 3000. Clearly, using only vm_0 will output a penalty to be paid by the provider, namely the cost of reading all the 4000 tuples, (TR_Q - SLO_Q) * p = (150-100)*p = 50*p. It is wise then to bring another VM (vm_1) up, to help. Let us assume that RR_1 = 10 and NT_1^Q = 2000.

At this point it may be seemed that those two VMs are enough to process Q and satisfy its SLO by rewriting Q into the following two (sub)queries, Q_1 and Q_2, running the first on vm0 and the second on vm_1, respectively. Note that we use virtual partitioning (i.e. partitioning the range over the primary key) to divide Q into Q_1 and Q_2.

SELECT * // <--- Q1

FROM table T1

WHERE T1.pk >= 0 and T1.pk < 3000;

SELECT * // <--- Q2

FROM table T1

WHERE T1.pk >= 3000 and T1.pk < 4000;

Our proposal to deal with this issue is to more partition queries so that the executing VMs' reading rate can be monitored often enough in such way that other VMs can be added as needed in order to enforce SLO_Q.

If monitoring is too constant that means the original queries would have to be partitioned in too many sub-queries, then the overhead added may hurt more than help. If hardly done it may be too late to make any corrections and to avoid potential penalties.

To address the partitioning process we build on historical data, i.e., how long it took to process a select-range query with a given selectivity using a certain number of partitions. We assume the existence of a table H for each VM where each entry has a 4-tuple: (Partitioning attribute, query's selectivity, number of partitions, average processing time).

Using H we can find the maximum number of partitions and we can divide a query on, thus allowing an as-frequent-as-possible monitoring, while still satisfying SLO_Q.

For the sake of argumentation, let us assume that H has enough information so that the following entries can be found when the query's selectivity is set to 4000and the partitioning attribute is pk as follows:

In this case, the larger number of partitions we can use is 4, maximizing the chance to monitor Q's performance while still satisfying SLOQ.

Thus, we divide query Q1 into four queries:

SELECT * // <--- Q1, 1

FROM table T1

WHERE T1.pk >= 0 and T1.pk < 500;

SELECT * // <--- Q1, 2

FROM table T1

WHERE T1.pk >= 500 and T1.pk < 1000;

SELECT * // <--- Q1, 3

FROM table T1

WHERE T1.pk >= 1000 and T1.pk < 1500;

SELECT * // <--- Q1, 4

FROM table T1

WHERE T1.pk >= 1500 and T1.pk < 2000;

SELECT * // <--- Q1, 5

FROM table T1

WHERE T1.pk >= 2000 and T1.pk < 2500;

SELECT * // <--- Q1, 6

FROM table T1

WHERE T1.pk >= 2500 and T1.pk <3000;

Even though it is not discussed here for the sake of brevity, a similar reasoning would be applied with respect to Q2 to be processed at vm1.

This partitioning methodology using primary key as a partitioning attribute is the same for both select-range queries and aggregation queries which are presented in the next subsection. Let us assume that vm1's performance is stable though and it is able to finish its workload as planned.

When Q1; 1 finishes we have the first opportunity to monitor, in a non-invasive manner, the

VM's performance. Let us assume that it spent actually 40 seconds to finish. This leads us to reset the value of that VM's reading rate to $RR_0 = 20$ (500 tuples in 50 seconds), which leads to an expected completion time, for all five remaining sub-queries, of 200 seconds. This brings the expected completion time above SLOQ, and triggers a revision of the initial provisioning, so that SLAQ can still be satisfied. Note that before reviewing the initial provisioning, the three remaining partitions (Q1, 2, Q1, 3 and Q1, 4,Q1,5and Q1,6) are gathered in a single query.

Our elastic system would be able to process *Q* as follows: vm_0 would be used from time 0 to110 in order to retrieve tuples satisfying the primary key range [0, 2000], vm_1 would also be used from time 0 to 100 in order to retrieve tuples satisfying the range [3000, 4000], and vm_2 would be used from time 50 to70 to retrieve tuples in the range [2000, 3000]. The number of VMs used as a function of time to execute Q would require two VMs between time [0, 40], three VMs between time [40, 60] and again two VMs between time [60, 90] as we could see in Fig.1. On the 110thsecond, the VMs were deallocated.

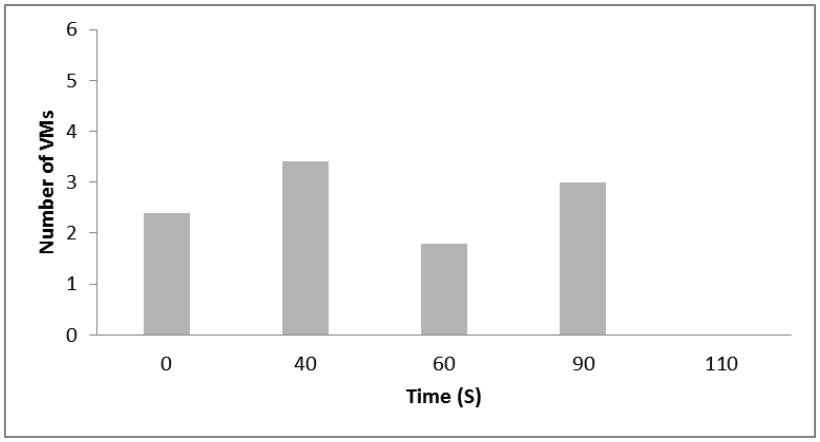

Figure 1: Variation in the number of nodes allocated by our approach.

c) Aggregation query

Assume that the following single aggregation query Q, with SLOQ equal to 100 seconds, is received by the cloud provider.

SELECT OPER(T.attr) // <--- Q

FROM table T1;

The difference between this example and the previous one is how to partition an aggregation query. For partitioning Q, we added a range over the primary key of T1 which is used as the partitioning attribute. The range value in this case is between the maximum and minimum value for the primary key. In order to simplify, let the maximum value be 2999 and the minimum be zero.

SELECT OPER(T1.attr) // <--- Q

FROM table T

WHERE T1.pk >= 0 and T1.pk < 4000;

We assume that the query's selectivity in this case is the range' selectivity over T.pk (Qs = 4000 tuples).

If the aggregation operator is a distributive function as COUNT, for example, the result of Q is the summation of the results collected for each partition. However, if the aggregation operator is an algebraic function as AVG, the original query result may not be easily obtained through the partitions. We have to transform to distributive functions; in the case of AVG, we could transform on SUM and COUNT, so that the original query result can be easily obtained through the summation of partitions' results.

If we consider the same scenario presented in the previous example, vm_0 has a reading rate $RR_0 = 30$ and consequently $NT_Q0 = 3000$. Using only vm_0 will yield a $((TR_Q\text{-}SLO_Q)*p = (150\text{-}100)*p = 50*p)$ to be paid by the provider. It is wise then to bring another VM ($vm1$) up, to help. Let us assume that $RR_1 = 10$ and $NT_Q1 = 2000$.

We rewrite *Q* into the two following (sub)queries, *Q*1 and *Q*2, running the first on vm_0 and the second on vm_1, respectively.

// *Q*1:

SELECT OPER(T1.attr)

FROM table T1

WHERE T1.pk >= 0 and T1.pk < 3000;

// *Q*2:

SELECT OPER(T1.attr)

FROM table T1

WHERE T1.pk >= 3000 and T1.pk < 4000;

We could rely on the same historical data *H* of the previous example since we use the same partitioning attribute. The larger number of partitions in

this case is4, because it maximizes the chance to monitor Q's performance while still satisfying SLO_Q.

Thus, we divide query Q_1 into four queries:

SELECT OPER(T1.attr) // <--- Q_1,1

FROM table T1

WHERE T1.pk >= 0 and T1.pk < 500;

SELECT OPER(T1.attr) // <--- Q1,2

FROM table T1

WHERE T1.pk >= 500 and T1.pk < 1000;

SELECT OPER(T1.attr) // <--- Q1,3

FROM table T1

WHERE T1.pk >= 1000 and T1.pk < 1500;

SELECT OPER(T1.attr) // <--- Q_1,4

FROM table T1

WHERE T1.pk >= 1500 and T1.pk < 2000;

SELECT OPER(T1.attr) // <--- Q_1,4

FROM table T1

WHERE T1.pk >= 1500 and T1.pk < 2000;

SELECT OPER(T1.attr) // <--- Q_1,5

FROM table T1

WHERE T1.pk >= 2000 and T1.pk < 2500;

SELECT OPER(T1.attr) // <--- Q_1,6

FROM table T1

WHERE T1.pk >= 2500 and T1.pk < 3000;

A similar thinking would be applied with respect to Q_2 to be run at vm1. As in the previous example, we assume that vm_1's performance is stable yet and it is able to finish its workload as planned.

After $Q_{1.1}$finishes we have the first opportunity to monitor the VM's performance in a non-invasive manner. For the sake of brevity, we could suppose the same what happened in the previous example where $Q1,1$ spent 40 seconds to finish, therefore the VM's reading rate decreases to $RR_0 = 20$ and leads to an expected completion time, for all three remaining partitions ($Q1,2$, $Q1,3$ and $Q1,4$, $Q1,5$and $Q1,6$), of 200 seconds.

Let us allocate a new VM (vm2) with RR2 = 40 to satisfy the SLOQ and to o²oad some query processing. Then all remaining 1000 tuples might be read by vm2 in 20 seconds, in the best case, which does not lead to a violation of SLOQ (recall that at this point 40 seconds have already been spent on Q1;1). We have the same elastic behaviour (two VMs between time [0,40], three VMs between time [40, 60] and again two VMs between time [60, 90]) presented in Fig.1.

The fact that a single basic approach can be easily adapted/ applied to different types of queries is a significant advantage. For instance, any optimization that can be done to the basic underlying (*select-range*) query can benefit other types of queries.

IV. Our Adaptive Approach

a) *Prototype Architecture*

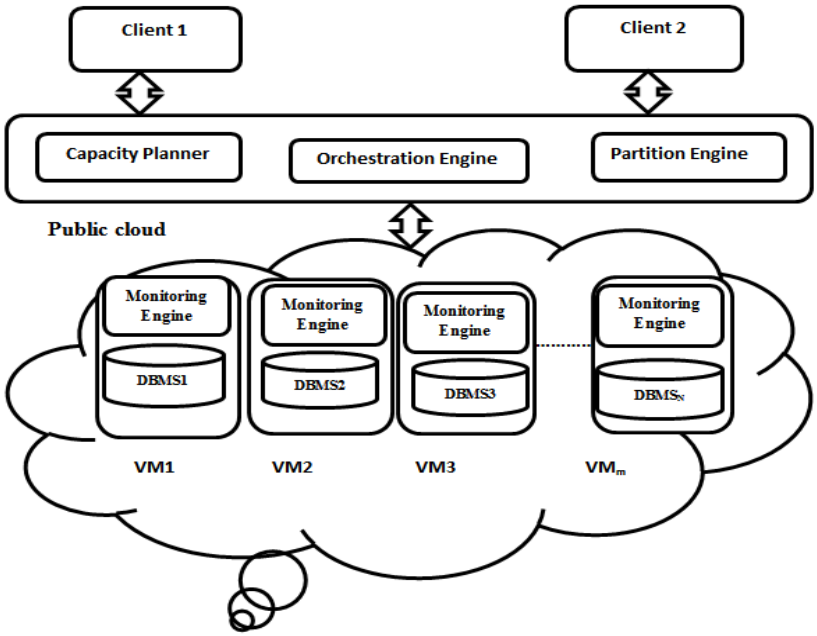

Figure 2 : Architecture

- The partition engine uses table *H* and is responsible for partitioning the query aiming at respecting the query's SLO.
- The monitoring engine is executed within each VM vm_i allocated to process a query *Q* and aims at making sure each VM keeps within the expected SLO. VMs can request" and offer" help. Both the partition engine and monitoring engine form the core of our approach.
- The capacity planner initially provisions a number of VMs to process a query *Q* within the agreed SLO_Q, minimizing the computational cost and penalty. It also has to make some decisions when the monitoring engine warns that the SLO_Q is about to be violated.
- The orchestration engine communicates with the capacity planner to obtain a provisioning, and with the partition engine to obtain the partitions and afterwards gives them to the monitoring engine.

V. Implementation

a) Partition Query Algorithm

Step1: Start/begin

Step2: Read the input elements H, Q, SLO_Q, V= {vm_0, vm_1,… ,vm_m}.

Step3: For each vm_i that belongs to V do the following four steps

i. Calculate Q_i which is sub query of Q with NT_i^Q selectivity.
ii. Calculate n_i^Q which is the query in H, the maximum number of partitions for Q_i satisfying SLOQ .
iii. Calculate P_i i.e. divide Q_i in n_i^Q partitions.
iv. Call monitoring algorithm by passing Pi and SLO_Q as input.

Step4: Stop/end

Partition Query algorithm describes how partitions are created from a query Q taking into account SLO_Q, performance of all vm_i that belongs to V and data table H.

The *working of this algorithm* is as follows:

For each vmi allocated to process Q, the partition engine rewrites Q into a sub query Q_i which has selectivity equal to the number of tuples that vm_i can read without violating SLO_Q, i.e., NT_i^Q and then Q_i is partitioned using data table H by constructing the largest set of partitions Pi to be processed by vmi, such that the average Pi processing time does not exceed SLO_Q.

How to obtain the value NT_i^Q for each vm_i is discussed in the next section. After built Partitions, Algorithm 2 i.e. monitoring algorithm is applied inside each vmi.

b) VM's Monitoring Algorithm

Step 1: Start/begin

Step 2: Read the input elements P_i , SLO_Q which are obtained from partitioning algorithm.

Step 3: $P_s := \text{selectivity}(P_i)$;

Step 4: until $P_i \neq 0$ do the following steps

i. $Q := P_i.\text{remove}()$;
ii. $T_{start} := \text{timer}()$;
iii. $TP := \text{execute}()$;
iv. $T_{end} := \text{timer}()$;
v. $T_q := T_{end} - T_{start}$;
vi. $T_{spent} := T_{spent} + Tq$;
vii. $P_s := Ps - |TP|$;
viii. $T_{estimated} := P_s / RR_i$;
ix. $TR_Q := T_{spent} + T_{estimated} + TM_i$;
x. Check

if $TR_Q - SLO_Q > 0$ then do the following two steps

a. $T_{remain} := SLO_Q - (T_{spent} + TM_i)$;
b. return make Decision (Vm_i, P_i, T_{remain});

else if $TR_Q - SLO_Q < 0$ then do the following two steps

a. $ST_i := SLO_Q - TR_Q$;
b. HasSlackTime(ST_i);

xi. remove VM(vm_i);

Step 5: Stop/end.

Monitoring algorithm, monitors query partition processing, is carried on at each virtual machine allocated to this process.

The *working of this algorithm* is as follows:

At each vmi, monitoring algorithm starts monitoring the VMs' reading rate and estimates how long it takes to finish all partitions in P_i. For each partition P_i, monitoring algorithm calculates the time spent to process the partition. Therefore, it is possible to know how many tuples of P_i remain to be retrieved, and also the estimated time to do that. This estimated time is based on the VM's reading rate, i.e., RR_i. If the estimated total time needed to process Q is greater than SLO_Q , this indicates that SLO_Q may be violated. Note that we also consider the time TM_i spent to monitor vm_i's performance. Hence, continuing to process Pi only with vmi will yield a penalty. In order to use the remaining time to process the remaining partitions in Pi without violating SLO_Q, we have to make some decision .At this point P_i should be recomputed for vm_i, using make decision algorithm.

Monitoring algorithm should also work in the situation when the SLO_Q could be satisfied faster than expected. In such case, we may use the slack time for processing other incoming query partitions or to reduce the number of VMs allocated. ST_i represents the slack time value of vm_i that can be further allocated without violating SLO_Q. The VMs with slack time are used by provisioning algorithms, for reusing overloaded machine resources. The selectivity of each partition is calculated in accordance to the range selectivity of partitioning

attribute. Recall that in all queries we chose the primary key as the partitioning attribute.

c) Initial Dynamic Provisioning Algorithm

Step 1: Start/begin

Step 2: Read input elements Q, SLO_Q .

Step 3: V := ø ;

Step 4: Q_s:= Selectivity(Q);

Step 5: Until $Q_s > 0$ ^ S≠ 0 do the following steps

i. (vm_i ,ST_i) :=S.remove( );
ii. Check
   a. if (ST_i> SLO_Q) then NT_i^Q :=SLO_Q * RR_i ;
   b. else NTiQ :=ST_i * RR_i ;
iii. Q_s := Q_s - NT_i^Q;
iv. V := V U {vm_i };

Step 6: Until Qs >0 do the following steps

i. Vm_i :=new( );
ii. NT_i^Q :=SLO_Q * RR_i;
iii. Q_s := Q_s - NT_i^Q;
iv. V := V U {vm_i };

Step 7: Call Partioning Query algorithm by passing the variables H, Q, SLO_Q, V as input.

Step 8: Stop/end.

This algorithm implements the initial provisioning approach. The purpose of this algorithm is to compute the smallest set V of virtual machines (V = {vm_0, ……,vm_n}) that should be initially dedicated to processing Q while satisfying SLO_Q. For each vm_i allocated it is required to know the amount of Q's tuples that vm_i can process without violating SLO_Q i.e., NT_i^Q. NT_i^Q is computed agreeing to the vm_i's reading rate capacity ($RR_{i)}$ within SLO_Q. It computes the Q's selectivity and computes the adequate amount of tuples (NT_i^Q) that should be provided to each vm_i that belongs to V .We can get the query's selectivity by using the query plan statistics presented by the using the query plan statistics presented by the Database Management Systems.

Here we take the advantage of VM's slack time. This algorithm finds in S the largest set of vm_i whose ST_i can be devoted to processing Q. The choice of the largest set is to allow the provider to serve more clients using less computing resources (VMs) and meeting the SLA. The choice is made using a greedy method, following the order of S. Let S be an ordered set of pairs (vm_i, ST_i), which contains slack time ST_i for each vm_i, such that there is only one element for each vm_i and ST_i is greater than 0 for all set elements. The set S is ordered by the descendent order of ST_i value.

This algorithm calculates Q's selectivity. If S =0, the algorithm removes the first vm_i from S and computes the number of tuples vm_i can retrieve. If ST_i> SLO_Q, then NT_i^Q is calculated based on SLO_Q and RR_i. Otherwise, NTQ_i is computed based on ST_i and RR_i.

The second loop in this Algorithm is responsible for distributing remaining tuples in Q to new VMs. When instantiating a new virtual machine vm_i, the algorithm calculates its NT_i^Q . NT_i^Q is calculated using the VM's reading rate (RR_i) and the value of SLO_Q. Algorithm 3 terminates when there is no tuples in Q to be distributed. After that, our strategy partitions Q (Algorithm 1 discussed in the previous subsection) is called to monitor the performance of each provisioned VM during Q execution.

Recall that in Algorithm 2 we have to make some decision to continue query processing while satisfying SLO. We therefore propose an elastic solution (described in Algorithm 4), which dynamically provisions VMs. At the monitoring stage, Algorithm 4 is called for each vmi that has the possibility of violating SLO_Q. Suppose that vmi has a set of remaining partitions Pi, which should be processed within T remain units of time without violating the SLO_Q. Algorithm 4 recalculates the number of VMs to help vm_i to finish the processing of Pi within T remains, aiming at satisfying SLO_Q. First, Algorithm 4 recalculates how many tuples vmi may retrieve while satisfying SLO_Q. After that, Algorithm 4 provisions a new set of VMs and allocates to each of them an amount of tuples to be processed.

d) Decision Making Algorithm

Step 1: Start/begin

Step 2: Read input elements vmi, Pi, Tremain

Step 3: V := 0 ;

Step 4: Q' := gather Partitions (Pi);

Step 5: Q's := selectivity(Q);

*Step 6:*NTi^Q := Tremain* RR_i

*Step 7:*Q's :=Q's-NT_i^Q;

Step 8: V := V U {vm_i};

while Q's >0 ^ S ≠ 0 do the following steps

i. (vm_i ,ST_i) :=S.remove( );
ii. Check
   a. if (ST_i> SLO_Q) then NT_i^Q :=Tremain* RR_j ;
   b. else NT_i^Q :=ST_j * RR_j ;
iii. Q's := Q's – NT_j^Q ;
iv. V := V U {vm_j };

Step 9: Until Q's >0 do the following steps

i. Vm_i :=new( );
ii. NT_j^Q :=Tremain * RR_j ;
iii. Q's := Q's – NT_j^Q ;
iv. V := V U {vm_j};

Step 10: Call Partioning Query algorithm by passing the variables H, Q, Tremain, V as input.

Step 11: Stop/end.

At the monitoring stage, Algorithm 4 is called for each vmi that has the possibility of violating SLO_Q.

At the monitoring stage, Algorithm 4 is called for each vmi that has the possibility of violating SLO_Q. Suppose that vmi has a set of remaining partitions P_i, which should be processed within Tremain units of time without violating the SLO_Q. Algorithm 4 recalculates the number of VMs to help vmi to finish the processing of P_i within Tremain, aiming at satisfying SLO_Q. First, Algorithm 4 recalculates how many tuples vmi may retrieve while satisfying SLO_Q. After that, Algorithm 4 necessities a new set of VMs and allocates to each of them an amount of tuples to be processed.

From Algorithm 4, the remaining partitions of P_i are gathered into a new query Q'. Then $NT_i^{Q'}$ is figured, i.e., the amount of Q's tuples that vm_i

If there still are tuples to be processed in Q' and S = 0 we have to allocate new VMs. When instantiating a new machine vm_j, the algorithm calculates its $NT_j^{Q'}$ which is using the VM's reading rate (RR_j) and the value of Trema in. Algorithm 4 terminates when there are no more tuples of Q' to be distributed. After that, our partitions Q' (Algorithm 1 discussed in the previous subsection is called) to monitor the performance of each provisioned VM during Q' execution, trying to ensure the Q' execution with in Tremain.

VI. Conclusion

A cloud-based, non-intrusive, automatic and adaptive performance monitoring technique for DBMSs for *select-range* queries and *aggregation* queries, as well as an approach that dynamically minimizes the number of VMs needed to satisfy the queries' SLO.

VII. Future Scope

Future work will focus on queries that use joins. Future work is to study the other parameters that can be used beyond the reading rate of each VM, such as CPU and effective memory available in each VM.

References Références Referencias

1. Coelho da Silva T L, Nascimen to M A, de Mace do J A F, Sousa F R C, and Machado J C "Towards non-intrusive elastic query processing in the cloud". In Proc. the 4th International Workshop on Cloud Data Management, Oct. 29-Nov. 2, 2012.
2. Sharma U, Shenoy P, Sahu S, Shaikh A," A cost-aware elasticity provisioning system for the cloud" In Proc. the 31stInternational Conference on Distributed Computing Systems, June 2011.
3. Mian R, Martin P, Vazquez-Poletti J L," Provisioning data analytic workloads in a cloud" Future Generation Computer Systems, 2013,
4. Rogers J, Papaemmanouil O, Cetintemel U,"A generic auto provisioning framework for cloud databases" In Proc. the 26th IEEE International Conference on Data Engineering Workshops, March 2010.