On defining metrics for elasticity of cloud databases

Rodrigo F. Almeida[1], Flávio R. C. Sousa[1], Sérgio Lifschitz[2] and Javam C. Machado[1]

[1] Universidade Federal do Ceará - Brasil
rodrigo.felix@lsbd.ufc.br, {sousa,javam}@ufc.br
[2] Departamento de Informática, PUC-Rio - Brasil
sergio@inf.puc-rio.br

Abstract. In the database area, elasticity of cloud computing has required data systems to increase and decrease their resources on demand. However, traditional benchmark tools for data systems are not sufficient to analyze some specificities of these systems in a cloud. New metrics for elasticity are needed to provide an indicator both from consumer and provider perspective. In this work we present a set of metrics for elasticity of cloud data systems. In addition, we evaluate our model performing experiments and analyzing their results.

Categories and Subject Descriptors: H.2 [**Database Management**]: Miscellaneous; H.3 [**Information Storage and Retrieval**]: Miscellaneous

Keywords: Benchmarking, Elasticity, Databases, Cloud

1. INTRODUCTION

Scalability, elasticity, and pay-per-use pricing model 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]. Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible. [Herbst et al. 2013] Stateful systems, such as DBMSs, are hard to scale elastically because of the requirement of maintaining consistency of the database that they manage [Minhas et al. 2012]. Some DBMSs implement elasticity [HBase 2013] [Cassandra 2013] [MongoDB 2013]. However, consumers and providers want to measure the elasticity of a given database system. On one hand, consumers want to compare some cloud data systems to choose one that fits better to their needs. On the other hand, providers want to meet the Service-Level Agreement (SLA) of database systems with the minimum resources and cost. There are some studies on how to measure the elasticity [Cooper et al. 2010] [Islam et al. 2012] [Dory et al. 2011]. However, these studies do not address important issues like considering DBMS-specific aspects, analyzing over-provisioning scenario and measuring elasticity when the number of clients varies during workloads. Changing the number of clients during workload illustrates a more realistic scenario for many applications whose number of users goes up and down. Particularly, reducing the number of clients is a more challenging aspect, that is not addressed for most related works.

The major contributions of this article are (i) definition of a model with metrics to measure the elasticity of cloud database systems, (ii) presentation of two different perspectives (consumer and provider) for elasticity, and, finally, (iii) evaluation of our model through some experiments. In order to perform these experiments, we developed BenchXtend [Almeida 2013b].

2. OUR PROPOSED APPROACH

2.1 BenchXtend

We propose a tool called BenchXtend [Almeida 2013b] which is based on YCSB [Cooper et al. 2010] to provide a way to change the number of clients while running a workload, as well as to calculate the metrics proposed in this work. Varying the number of clients accessing a cloud data system is essential to properly evaluate the elasticity of that system, since the load will vary and then the system will have to act (adding or removing resources) to keep up the response time in an acceptable range that satisfies the SLA. In our tool, the number of clients is changed automatically according to a timeline defined in an input file. In our context, a timeline is simply a list of pairs $< time, number of clients >$ explicitly defined in a file, that describes the expected number of clients in such a moment.

Our tool sends queries to, what we call, a Cloud Database System that is composed of (i) an Instance Manager (composed of a Monitor and a Decision Taker), (ii) a Database Manager and (iii) a pool of instances (virtual machines) that are running or available to be started. The Instance Manager is responsible for monitoring the pool gathering statistics, as well as for taking decision on starting or on stopping instances of the pool. The Database Manager is the access point to where queries are sent. Depending on the DBMS, there is no central node to receive queries and distributed. Thus, queries are sent all over the pool and the Database Manager is a regular node. The pool of instances is only a set of pre-configured instances that can be started or stopped by the Instance Manager. Since our focus is not on how well (or badly) designed the cloud database system itself is, but on the benchmark tool, for a matter of simplicity, we implemented separately our own Instance Manager [Almeida 2013a] in Ruby calling Amazon EC2 API. Other authors [Konstantinou et al. 2012] [Cruz et al. 2013] propose more robust solutions to manage instances, even though they are not developed for Amazon EC2.

2.2 Elasticity metrics

Our approach to define metrics for elasticity extends the work proposed on a previous work [Almeida 2012]. We use a penalty model approach to measure imperfections in elasticity for database systems. Similarly to [Islam et al. 2012], our elasticity model is composed of two parts: penalty for over- and under-provisioning. Unlike [Islam et al. 2012], we explore database system features, like response times, and present both the consumer and provider perspectives. We consider a scenario where a consumer accesses a database service in a Platform-as-a-Service (PaaS) provider. In a PaaS scenario, database systems are usually exposed as services and the applications deployed by the consumers can execute queries on them, abstracting the complexity of replication, consistency, and so on.

2.3 Consumer perspective

Due to the large number of PaaS providers and to the so-claimed buzzword elasticity, consumers need to have a model to evaluate and compare elasticity of database systems. From a PaaS-consumer perspective, a database system is elastic if, regardless the number of queries submitted to the system, it satisfies the SLA. We assume that upper and lower bounds for response times are defined by *operation type* in the SLA. Queries whose response times are between these bounds are not considered either under- nor over-provisioned. YCSB provides five operation types (read, insert, update, delete, scan) but, due to paper space restrictions, we analyze in our experiments only scan and insert.

Under-provisioning penalty (*underprov*): The strategy for *underprov* is compliant to the concept of penalties for database services in the cloud. In this case, the resources allocated to the consumer are under-provisioned, i.e. insufficient to keep up the quality, generating a response time increase and consequently causing a penalty. This penalty is for violating the upper bound of SLA. Penalties due to under-provisioning are directly related to bad elasticity, since, if the system had a good one, it would have scaled up to address the increase of demand to avoid penalties. We define this metric

(shown in Equation 1) as the average of the ratio execution time by expected time of those n queries whose response times are greater than the upper bound defined in the SLA and that are not outliers. In order to remove discrepant values, we use interquartile range analysis to identify extreme outliers on the response times data distribution and then we remove them from the calculation of all metrics of our model. Outliers are those response times that are out of the range $[Q_1 - 3 * R, Q_3 + 3 * R]$, where Q_1 is the 1st quartile, Q_3 is the 3rd quartile and R is the interquartile range. Expected response time ($expected_{rt}$) is defined by operation type in the SLA. This time represents the maximum time (or upper bound) a query should take without disrespecting the SLA. The violated execution time ($violated_{et(i)}$) represents time spent by a query i that did not meet the SLA and that is not an outlier.

$$underprov = \frac{\sum_{i=1}^{n} \frac{violated_{et(i)}}{expected_{rt(i)}}}{n} \quad (1)$$

The higher *underprov* is, the less elastic the database system is, because more queries violates the SLA. $\frac{violated_{et(i)}}{expected_{rt(i)}}$ is always greater than 1, since it is applied only for violated queries.

2.4 Provider perspective

From a customer perspective, we presented *underprov* metric. For a PaaS provider, besides measuring that, it is also essential to evaluate how efficient the database system is to use only the minimum amount of resources to meet the SLA. Thus, from a provider perspective, our approach proposes *underprov* exactly as showed in the last section, based on observed SLA, and *overprov*, based on the charged level of resources. Finally, we put *underprov* and *overprov* together to get a single dimensionless value for elasticity of cloud database systems.

Over-provisioning penalty (*overprov*): When there is over-provisioning, the provider offers more resources than necessary to meet a demand. Thus, the provider may be subject to a higher operating cost than the necessary to meet the SLA. In this situation, the database system has a number of resources (i.e. nodes) that are running a given workload, but this amount may be higher than necessary. Unlike *underprov*, this metric does not make sense from a consumer perspective, since for a PaaS consumer there is no problem to have more available resources if that does not imply in a cost increase. *overprov* considers the execution time of queries performed when the database system is over-provisioned. We define this metric (shown on Equation 2) as the average of the ratio lower bound time by execution time for those m queries that are considered over-provisioned. *overprov* moves the $expected_{rt}$ to the numerator since in an over-provisioning scenario the execution time is supposed to be lower than the expected one. Similarly to *underprov* outlier values are disconsidered. The query execution time ($query_{et}$) is the time spent by a query i that is considered in the *overprov* calculation.

$$overprov = \frac{\sum_{i=1}^{m} \frac{expected_{rt(i)}}{query_{rt(i)}}}{m} \quad (2)$$

Elasticity for Cloud Database System ($elasticity_{db}$): Since the provider is affected both on under- and over-provisioning scenario, from his perspective, elasticity can be given by the weighted average of metrics from both scenarios. Worthwhile noticing that the penalty due to over-provisioning should have a lower weight than the under-provisioning one, since that affects only one of the parties. To provide a dimensionless metric that quantifies elasticity from the provider perspective, we suggest the formula 3 and provide the variable x to set how bigger *underprov* weight is when compared to *overprov* weight (fixed to 1, for a matter of simplicity). Numerical domain of x is $[1, \infty)$.

$$elasticity_{db} = \frac{x * underprov + overprov}{x + 1} \quad (3)$$

For instance, x weight could be defined based on costs. In this case, x could be defined by the cost of paying for underprovisioning penalties and *overprov* weight (set to 1) by the cost that could be saved if some resources had been released when environment was overprovisioned. Lower values of $elasticity_{db}$ indicates a more elastic cloud database system, since it makes better use of resources while meeting the SLA. Our metric meets tests of reasonableness, such as (i) elasticity is non-negative and (ii) elasticity captures both over- and under-provisioning.

3. EXPERIMENTS

3.1 Environments

Since our goal is to validate our metrics and not compare elasticity among database systems, we considered one database system in our experiments and compared metrics in two scenarios: (i) with elasticity, where the Decision Taker scales in and out and (ii) without elasticity, where no machine is added or removed during the workload. Decision Taker is the module responsible for adding or removing a node depending on monitored CPU usage [Almeida 2013a]. Cassandra (version 1.1.5) was chosen as the database system, due its wide adoption both by academy and industry. The following machines were used in our experiments: 1 EC2 instance (m1.medium) to run BenchXtend tool; 1 EC2 instance (m1.small) to run the Instance Manager. In addition, we set up 2 instances (m1.large) as seeds and 2 instances (m1.large) for data nodes. Seeds are started before the workload execution while data nodes keep turned off until the Decision Taker starts one of them.

3.2 Execution

We run YCSB Core Workload E (Short ranges - 95% scan, 5% insert), populating the database with 4 millions 1-KB 10-field records. Workload was executed for 4 hours, number of clients was changed according to a timeline (Figure 1) and a Linear function was chosen to interpolate timeline entries. We start Cassandra seeds and then perform the BenchXtend load phase. After that, we restart seeds and then start Monitor and Decision Taker. Before starting the run phase, we check if no compaction of Cassandra SSTables is being performed. During compaction [DataStax 2013] there is a temporary spike in disk space usage and disk I/O that may affect our results. Each started instance had Cassandra already configured, but the database is empty. As soon as we add or remove an instance, Instance Manager updates the *hosts* property of the YCSB workload file in order to let it know about the cluster change. In the SLA file, we set expected response time by operation type. For insert operations we set $200000\mu s$ and $100000\mu s$ as upper and lower bounds, respectively. For scan operations the values were $700000\mu s$ and $350000\mu s$. These values were defined after some experiments in our environment and there is no rule of thumb to define them. x weight was empirically set to 5.

3.3 Results

The first chart of Figure 1 plots when machines are actualy added or removed and compare them with client variation. The time to add (bootstrap) a Cassandra node may be considerably high and may vary a lot. 5min were taken to add the 1st machine, while 35min to the 2nd and 14min to the 3rd one. A reason for this is that Amazon EC2 presents performance issues depending on the time the experiments are executed and on the CPU model of the physical machines where the VMs are placed. Another reason is the cost of data streaming among Cassandra nodes. To remove (decommission) Cassandra nodes the variation was lower (about 2min30s for each of 3 decommission operations).

As we can see on Figures 1 and 2, response times increase as number of clients goes up and reduces as number of clients goes down. Thus, it is fairly reasonable to assert that changing number of clients directly affects response time. Although the number of clients is pretty much the same on first and second peaks, on the second one (from 7000s to 11000s) we can notice on Figure 1 higher values of

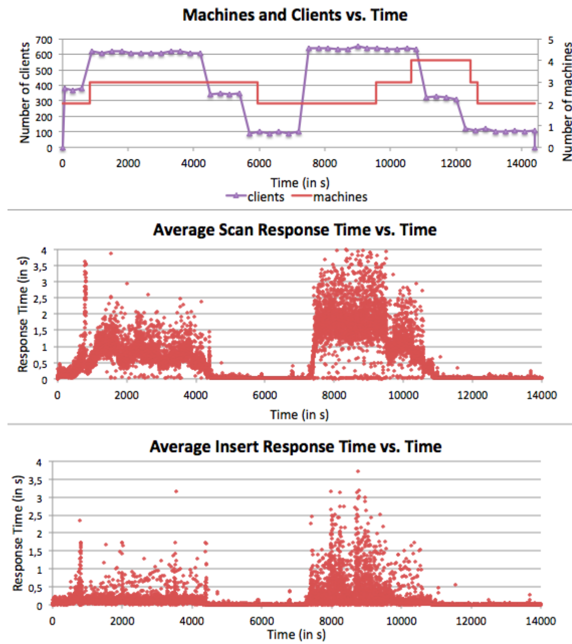

Fig. 1. Experiments with elasticity

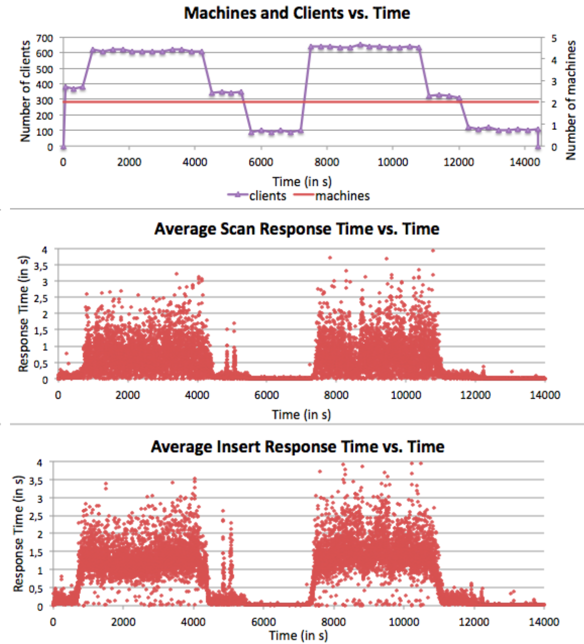

Fig. 2. Experiments without elasticity

	Scan		Insert	
	with elasticity	**w/o elasticity**	**with elasticity**	**w/o elasticity**
Total of underprov queries	514719	987528	16302	33116
Underprov metric	**2.019751**	**3.084927**	**1.439694**	**2.613020**
Total of overprov queries	2426529	1767091	137090	93563
Overprov metric	**16.859536**	**15.631292**	**22.129353**	**23.521127**
Elasticitydb metric	**4.493048**	**5.175988**	**4.887970**	**6.097704**

Table I. Metrics gathered in both experiments

response time, due to the long bootstrap (that is an I/O-intensive operation) time, about 35min, to add the 2nd machine. Scan operations require multiple I/Os that are considerably impacted by bootstrap. Insert operations performed well since Cassandra is write-optimized. In addition, we can see that insert operations perform better when new Cassandra nodes are added.

Metrics results are shown on Table I. For *underprov* of scan operations our model could capture the improvement on adding machines when system was overloaded. In this case, the number of violated queries is 52% lower than number of non-elastic experiment and *underprov* had also a lower value. Unlike what was expected, *overprov* of scan operations showed a higher value for the elastic experiment. This may suggest the strategy adopted by Decision Taker should be improved to drop earlier machines when the system is overprovisioned. $elasticity_{db}$ of scan operations is lower in elastic scenario (4.493048) than in the non-elastic one (5.175988), even with a higher value of *overprov*. For insert operations, metrics values showed lower values on elastic experiment, as expected.

4. RELATED WORK

[Klems et al. 2012] discuss about elastic scalability but do not present a metric for that and mention some metrics that do not consider SLA rules. [Shawky and Ali 2012] provide metrics inspired on elasticity definition from physics, focus on network bandwidths instead of database-specific aspects and did not use real data in their experiments. [Islam et al. 2012] [Dory et al. 2011] [Cruz et al. 2013]

discuss elasticity but do not propose metrics to compare their results with other works. [Cruz et al. 2013] presents improvements when compared to [Konstantinou et al. 2012] but no common indicator is used to analytically measure and compare elasticity of cloud data systems. [Islam et al. 2012] propose ways to quantify the elasticity concept in a cloud. They define a measure that reflects the financial penalty to be paid to a consumer, due to under- or over-provisioning. However, it does not take into account DBMSs features like response time. [Dory et al. 2011] provide definitions of elasticity for database and a methodology to evaluate the elasticity. However, these definitions deal only with under-provisioning scenarios and do not consider aspects like SLA, penalties, and resources. [Cooper et al. 2010] present *elastic speedup* and *scaleup*. The first metric illustrates the latency variation as new machines are instantiated. The second one is a traditional metric and does not encompass elasticity aspects. Even though these metrics are useful, they do not illustrate the consumer perspective.

5. CONCLUSION AND FUTURE WORK

In this work, we presented metrics to measure the elasticity of cloud databases. Our model presented metrics based on SLA and from two different perspectives. According to the analysis of experiments, we could validate our metrics since our model could capture the correct variation of elasticity between a scenario with elasticity and another without this. These metrics can now be used to compare elasticity of cloud database systems and to help providers on tuning strategies to add or remove resources on demand. As future work, we intend to execute experiments to compare elasticity of MongoDB and Cassandra and perform a statistical analysis to try to reduce the number of parameters of our model.

REFERENCES

Almeida, R. Benchxtend: a tool to benchmark and measure elasticity of cloud databases. In *Simpósio Brasileiro de Bancos de Dados - SBBD 2012 - Workshop de Teses e Dissertações*. pp. 93–98, 2012.

Almeida, R. F. rodrigofelix/benchxtend-monitor. `https://github.com/rodrigofelix/benchxtend-monitor`, 2013a.

Almeida, R. F. rodrigofelix/YCSB. `https://github.com/rodrigofelix/YCSB/`, 2013b.

Cassandra. The apache cassandra project. http://cassandra.apache.org/, 2013.

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *SoCC*. New York, NY, USA, pp. 143–154, 2010.

Cruz, F., Maia, F., Matos, M., Oliveira, R., Paulo, J., Pereira, J., and Vilaca, R. Met: Workload aware elasticity for nosql. In *EuroSys 2013*, 2013.

DataStax. About Writes in Cassandra. `http://www.datastax.com/docs/1.1/dml/about_writes`, 2013.

Dory, T., Mejías, B., Roy, P. V., and Tran, N.-L. Measuring elasticity for cloud databases. In *Proceedings of the The Second International Conference on Cloud Computing, GRIDs, and Virtualization*. Berlin, Heidelberg, 2011.

Elmore, A. J., Das, S., Agrawal, D., and El Abbadi, A. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD '11*. New York, NY, USA, pp. 301–312, 2011.

HBase. Apache hbase. http://hbase.apache.org/, 2013.

Herbst, N. R., Kounev, S., and Reussner, R. Elasticity in Cloud Computing: What it is, and What it is Not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 2013), San Jose, CA, June 24–28*, 2013. Preliminary Version.

Islam, S., Lee, K., Fekete, A., and Liu, A. How a consumer can measure elasticity for cloud platforms. In *ICPE'12 - Second Joint WOSP/SIPEW International Conference on Performance Engineering*. New York, NY, USA, 2012.

Klems, M., Bermbach, D., and Weinert, R. A runtime quality measurement framework for cloud database service systems. In *Proceedings of the 8th International Conference on the Quality of Information and Communications Technology*. IEEE, Conference Publishing Services (CPS), pp. 38–46, 2012.

Konstantinou, I., Angelou, E., Tsoumakos, D., Boumpouka, C., Koziris, N., and Sioutas, S. Tiramola: elastic nosql provisioning through a cloud management platform. In *Proceedings of the 2012 ACM SIGMOD*. New York, NY, USA, pp. 725–728, 2012.

Minhas, U. F., Liu, R., Aboulnaga, A., Salem, K., Ng, J., and Robertson, S. Elastic scale-out for partition-based database systems. In *SMDB '12, ICDE Workshops*. Washington, DC, USA, pp. 281–288, 2012.

MongoDB. Mongodb. http://www.mongodb.org/, 2013.

Shawky, D. and Ali, A. Defining a measure of cloud computing elasticity. In *Systems and Computer Science (ICSCS), 2012 1st International Conference*. Lille, FR, pp. 1–5, 2012.