19th International Computer Science and Engineering Conference (ICSEC)
Chiang Mai, Thailand, 23-26 November, 2015

# Performance Measurement of SimpleDB APIs for Different Data Consistency Models

Pornpan Ampaporn and Sethavidh Gertphol
Department of Computer Science, Faculty of Science,
Kasetsart University, Bangkok, Thailand

***Abstract*—Cloud platform providers usually offer several APIs (Application Program Interface) to help facilitate programmers to utilize cloud resources effectively by hiding complex cloud structures and mechanisms. However, there are some aspects of distributed computing that cannot be hidden. Depending on data consistency models, two or more clients may not see the same current state of the data. Developers should understand the performance of available APIs and consistency models they provide. This paper explores the performance of a suite of APIs that can be used to implement two different data consistency models in SimpleDB, a distributed non-relational database-as-a-service provided by Amazon. Based on our experiment, read requests in two consistency models offered by Amazon SimpleDB performed almost identically, with median latency of 16 ms. Write performance was about 3 times slower, with 56 ms median latency. In addition, there were greater performance variations for writes than reads. Lastly, a strong consistency model worked as advertised, returning latest value with every read. On the other hand, the correctness of an eventual consistency read depended primarily on the elapsed time since last write operation.**

***Keywords—Cloud Computing; consistency models; Amazon Simple DB; performance measurement***

## I. INTRODUCTION

Cloud development platforms such as Google App Engine or Amazon Web Services are becoming a mature technology for convenient and rapid application development and deployment on the Cloud. Platform-as-a-Service cloud providers create several tools and API (Application Program Interface) that helps developers use, monitor, and manage cloud resources programmatically. These APIs include reading and writing distributed non-relational database, creating and scaling the number of virtual machine instances, managing users of cloud resources, etc.

While these APIs tremendously facilitate programmers to utilize cloud resources effectively by hiding complex cloud structures and mechanisms, there are some aspects of distributed computing that cannot be hidden. One of the issues that developers must understand is data consistency in distributed system [1]. Because information may be replicated in several data stores and updates may occur at any data store, two or more clients may not see the same current state of the data (depending on data consistency models). Cloud providers usually have APIs to implement some forms of data consistency, but it depends on each developer to choose an appropriate model (or a mix of models) for his application.

978-1-4673-7825-3/15/$31.00 ©2015 IEEE

This research was funded by Kasetsart University Research and Development Institute

Consistency is not the only concern when deciding on a consistency model. A software engineer must also take into account the performance of mechanisms used to implement different models, for example, the latency and throughput of read and write requests for each model. Performance variation may also be another factor to be considered; the developers may want an application that performs similarly every time rather than one that is very fast most of the time but may sometimes execute sluggishly.

This paper explores the performance of a suite of APIs that can be used to implement two different data consistency models in SimpleDB, a distributed non-relational database-as-a-service provided by Amazon. We also test the consistency models offered by SimpleDB to evaluate the correctness of data with respect to time after last write. In this way, a developer will have a complete information to judiciously make a trade-off between data consistency, acceptable inconsistent data probability, and performance.

The paper is organized as follows. Section 2 discusses related work in the field. Section 3 explains the SimpleDB and available consistency models while Section 4 outlines out test programs and experiment setup. Section 5 shows test results and discussions follows in Section 6. Section 7 concludes the paper.

## II. RELATED WORK

Data consistency model is an important issue in distributed computing. This topic is included in several text books regarding the issue, such as [1]. When cloud computing becomes popular, data consistency is considered to be a differentiating service for cloud platform providers, i.e., it could be an issue to be negotiated and included in SLA (service-level agreement) [7] or charged for extra service [6].

Performance measurement of operations that used to implement different data consistency models had been done in [5] and [8]. In [8] the authors developed an experiment system using Amazon S3 that could adapt consistency models during run time and measure its performance. Our work is similar to work in [5] which also measure performances of read and write operations in Amazon SimpleDB and Google Big Table, but we focus more on latency and its variation during a day.

## III. System Model

In this section, we explain our experiment platforms which are Amazon EC2 and SimpleDB briefly. More importantly, the consistency models available to developers are discussed.

### *A. Amazon EC2 and SimpleDB*

Amazon EC2 [2] is a cloud computing platform that provides virtual machine configuration, deployment, and management to customers. A user can choose VM capabilities (e.g. processing power and the amount of memory), an image to be used (which includes an operating system and/or other applications), and then deploy the VM in one of several regions around the world (e.g. East Coast North America, Asia-Pacific, or Europe). The user can then remotely access and manage the VM through the Internet.

SimpleDB [3] is a type of non-relational distributed database-as-a-service provided by Amazon. A developer using SimpleDB does not have to administer the distributed database himself, he can just create a SimpleDB in one geographic region and read/write to it using provided APIs. Amazon will take care of data replication and distribution.

### *B. Consistency Model in Amazon SimpleDB*

While Amazon can help developers manage the backend management of SimpleDB, programmers still need to understand consistency models available when working with a distributed database. Amazon provides two consistency models for retrieving data from SimpleDB: eventual and strong consistency. A strong consistent read (select and get from a database) will always retrieve the latest write from the database. On the other hand, an eventual consistent read may retrieve the latest value, or a stale value, or even a previously written value [4]. Amazon states that strong consistent reads potentially has a higher read latency and lower throughput comparing to eventual consistent reads.

## IV. Experiment

The goal of our experiment is to evaluate the performance of SimpleDB in three operations: write, strong consistent read, and eventual consistent read. The experiment was also setup to evaluate the variation of SimpleDB performance during a day. We are also interested in the behavior and results of eventual consistent read to read the latest write data.

### *A. Test Program*

We developed a simple web application that will repeatedly write and read data from a SimpleDB database. The database is a just a single integer attribute. The test program consists of two parts: a writer and a reader. When activated, a writer starts writing a number to SimpleDB. This number is kept at the writer and initially starts at 0. Every 2000 milliseconds, the writer increments the number by 1 and writes it to SimpleDB. The writer stops after writing number 99. The start time and finish time of each write are recorded.

The reader starts execution after the first 2000 milliseconds has passed from when the writer starts. The reader repeatedly, and as fast as possible, reads a value from the SimpleDB database and records it along with the start time and finish time of each read. The reader can be configured to use either strong or eventual consistent read, and it will use the same method until it stops. The reader stops when it reads a value 99 or when five minutes has passed. Since the reader and the writer are located on the same machine, clock synchronization is not an issue in this test program.

### *B. Experiment Setup*

The experiment was conducted during the two-day period in April 2015. A SimpleDB domain was created in the US North East Region of Amazon Web Services. An m1.small instance which has 1 virtual CPU and 1.7 GiB memory installed with Ubuntu 12.04 and Tomcat 7.0 with JDK 1.7 was initiated also in the US North East Region. The test program was deployed to the VM instance.

During the first day, the reader was configured to use eventual consistent read. The test program was executed once every hour and the resulting performance and read records was saved. Then the reader was re-configured to use strong consistent read and the experiment repeated 24 times (once every hour) in the second day.

## V. Results

After the experiment concluded, we recorded 260,132 eventual consistent reads together with 2,400 writes and 225,801 strong consistent reads together with their 2,200 writes. There were two executions of strong reads that produced errors and we discarded those recordings. This is the reason why the numbers of strong read/write are less than eventual reads/writes. Since writes do not differentiated into strong or eventual, we combined all writes into one data set.

### *A. Write and Read performance*

From the records, we calculate a latency for each read and write. This is the time from sending request out until the value or acknowledgement comes back. Table I. shows the statistics for write latency and Fig. 2 shows the box plot. From the total of 4,600 writes, the median of write latency is 56 milliseconds while the mean is 114.4 seconds. The minimum write latency is 42 milliseconds and the maximum jumps to 2,391 milliseconds. The first quartile of write latency is 50 ms while the 3rd quartile is at 103 ms. The range of the whisker is 1.5 inter-quartile (IQR) range, and the lower whisker is at 29.5 ms and the upper whisker is at 182.5 ms. We did not plot outliers but the number of outliers above the upper whisker is 430 which is around 9.35% while there is no outliers below the lower whisker. It is clear that the write latency distribution is right-skewed.

The statistics for both eventual consistent read latency and strong consistent read latency are almost identical. They both have a median of 16 ms with 9 ms minimum and 14 ms as 1st quartile and 18 ms as 3rd quartile. The mean of eventual read latency is 18.78 compared to 18.19 for strong consistent read. The maximum of eventual read latency is 1,576 ms while it is 1,350 ms for strong consistent read. The whiskers of the box plot also calculated using 1.5 * inter-quartile range, and they are both at 8 ms for lower whiskers and 24 ms for upper

whiskers. Both reads do not have any outliers below the lower whiskers, but 7.81% of eventual reads and 8.66% of strong consistent reads took longer than 24 ms and are considered outliers. The read latency distribution shows negligible skewness.

TABLE I. STATISTICS OF READ AND WRITE LATENCIES

| | Writes | Eventual Reads | Strong Reads |
|---|---|---|---|
| n (obs.) | 4,600 | 260,132 | 225,891 |
| Min (ms) | 42 | 9 | 9 |
| Q1 (ms) | 50 | 14 | 14 |
| Median (ms) | 56 | 16 | 16 |
| Q3 (ms) | 103 | 18 | 18 |
| IQR (ms) | 53 | 4 | 4 |
| Max (ms) | 2,391 | 1,576 | 1,350 |
| Outliers (%) | 9.35 | 7.81 | 8.66 |
| Mean (ms) | 114.4 | 18.78 | 18.19 |
| SD (ms) | 186.84 | 17.61 | 24.34 |

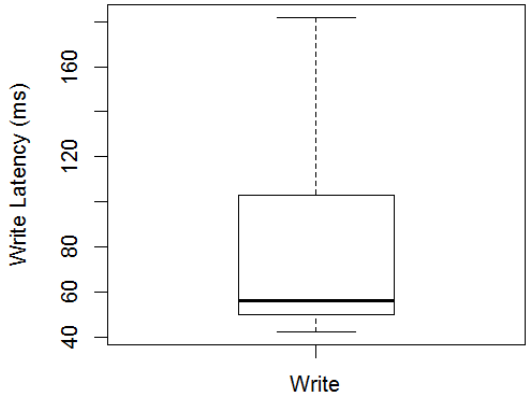


Fig. 1. Boxplot of write latency.

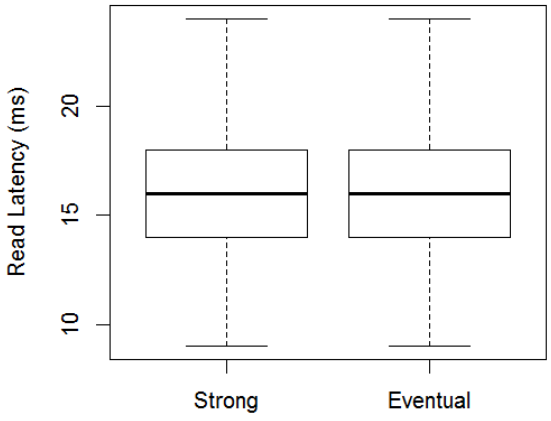


Fig. 2. Boxplot of strong and eventual read latencies.

TABLE II. STATISTICS OF READ AND WRITE THROUGHPUT

| | Q1 | Median | Q3 | IQR | Mean | SD |
|---|---|---|---|---|---|---|
| Eventual Reads per sec | 51 | 57 | 62 | 11 | 53.03 | 13.87 |
| Strong Reads per sec | 51 | 57 | 62 | 11 | 54.45 | 16.07 |

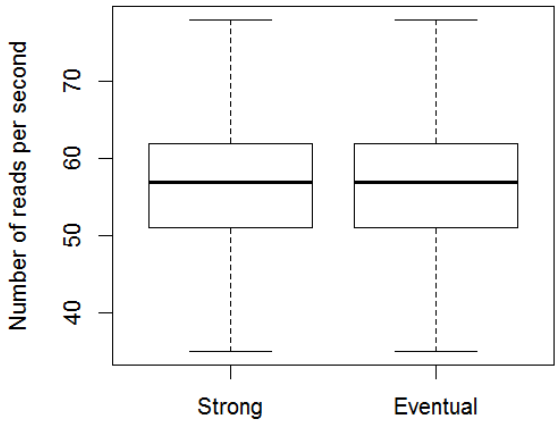


Fig. 3. Boxplot of reads throughput.

### *B. Write and Read performace by hour*

To evaluate read and write performance variations, we grouped the results by each test run (roughly 1 hour apart) and summarized latencies and throughput for each run. Table III. shows read and write latency for selected runs and Fig. 4-7 show box plot of read and write latency of all runs. There are two sets of write runs: one on the first day with eventual reads (Fig. 4), the other on the second day with strong consistent reads (Fig. 5). As noted, there were 22 runs only on the second day because of errors in the writer.

TABLE III. WRITE AND READ LATENCY BY TEST RUN

| | Run | Q1 | Median | Q3 | IQR | Mean | SD |
|---|---|---|---|---|---|---|---|
| Write (first day) | 7 | 48 | 51 | 53.3 | 5.3 | 69.21 | 144.91 |
| | 11 | 50 | 206.5 | 407.8 | 357.8 | 343.40 | 365.38 |
| | 17 | 126.5 | 136.5 | 150.2 | 23.7 | 207.50 | 212.96 |
| Eventual read | 7 | 14 | 16 | 17 | 3 | 18.71 | 24.13 |
| | 11 | 14 | 16 | 18 | 4 | 17.15 | 10.05 |
| | 17 | 15 | 16 | 18 | 3 | 20.93 | 34.28 |
| Write (second day) | 7 | 49 | 52 | 56 | 7 | 77.82 | 144.51 |
| | 11 | 50 | 59 | 271 | 221 | 255.4 | 350.52 |
| | 16 | 141.8 | 167.5 | 188 | 46.2 | 211.20 | 188.37 |
| Strong read | 7 | 13 | 15 | 17 | 4 | 15.8 | 7.49 |
| | 11 | 17 | 21 | 26 | 9 | 26.48 | 23.45 |
| | 16 | 14 | 16 | 18 | 4 | 17.76 | 10.25 |

From the plots, there were roughly 3 types of write performance. First was a low latency, low IQR one that had a median about 50 ms and inter-quartile range of less than 10 ms. Most test runs exhibited this type of quick and reliable latency. The second group had a high latency between 100 and 200 ms but a relatively small IQR of 50 ms. This was a group that had lower performance but still not too much variation. The boxplot of this group also did not show much skewness. The

third group was a high variance one that had IQR around 100 ms or greater. A few of test runs displayed this performance with extreme right skewness. Overall, the write performance was highly uneven.

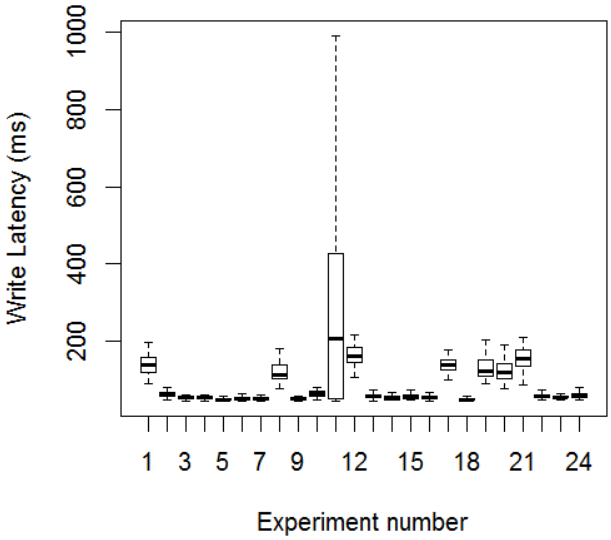


Fig. 4. Write latency by test run on the first day with eventual read.

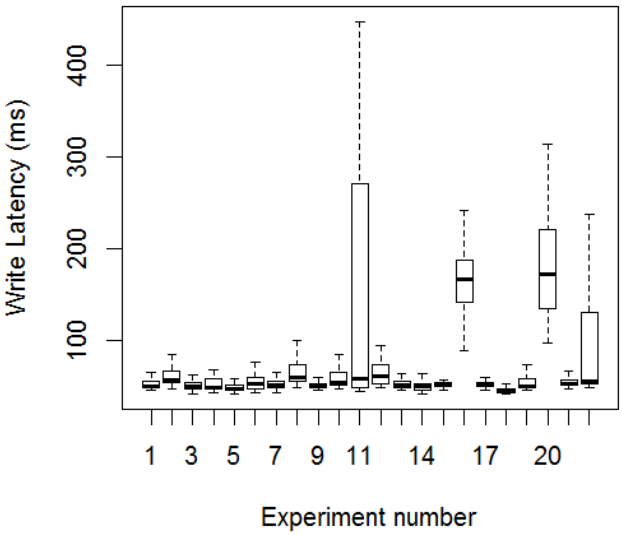


Fig. 5. Write latency by test run (second day) with strong consistent read.

On the other hand, read performances of both strong and eventual consistency remains stable during virtually all runs with median less than 20 ms and IQR less than 10 ms. There was only one single run than the median of read latency went up to 21 ms. Fig. 6 and Fig. 7 shows eventual and strong consistent read performance respectively. From these two figures, it would be very difficult to differentiate eventual read and strong consistent read from one another. There was one anomaly in run number 11 in strong consistent read data. That run was the same run where write latency IQR was almost 200 ms, so it may be an especially bad SimpleDB performance.

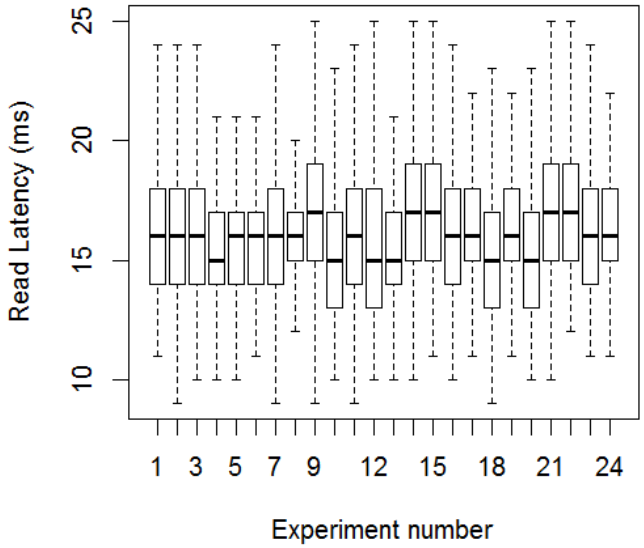


Fig. 6. Eventual read latency by test run.

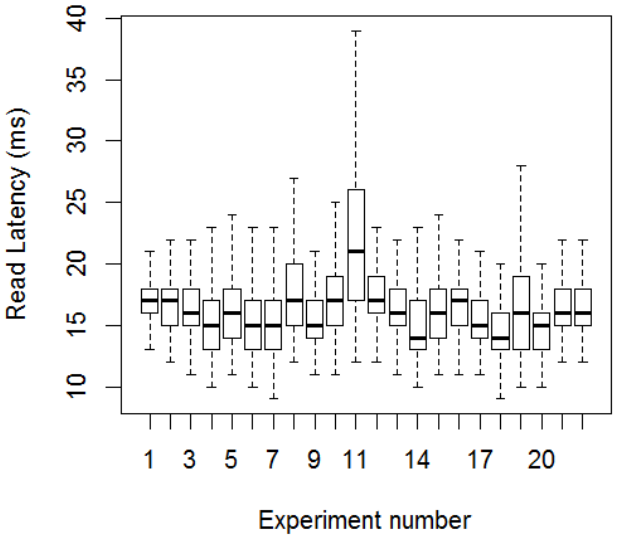


Fig. 7. Strong consistent read latency by test run.

### C. *Read correctness*

We want to analyze the probability of read returning the latest write value with regards to the elapsed time between that read and latest write. For each read, we determine the latest write that happened before the read was initiated. The time elapsed between the finish of latest write and the start of that read was calculated. Basically we find how long the read was requested after the latest write finished. We also note the correctness of the read; that is whether the read value is the same as the latest write value or not.

Next we grouped all reads by their elapsed time from latest write into bins. Each bin covers a 10 milliseconds period. For example, two reads with elapsed time from latest write of 52 ms and 57 ms belong to a same bin. Then we calculated the ratio between correct reads and the total reads in each bin.

Fig. 8 shows the resulting correctness probability with respect to elapsed time from latest write. Strong consistent reads perform as advertised by Amazon and return the latest write value 100% of the time in every elapsed time period. On the other hand, eventual consistent reads return the latest value around 30% of the time when elapsed time from latest write is less than 500 ms. The correctness jumps to around 80% when elapsed time is between 500 and 600 ms. Finally after 600 ms the correctness of eventual reads is almost 100%.

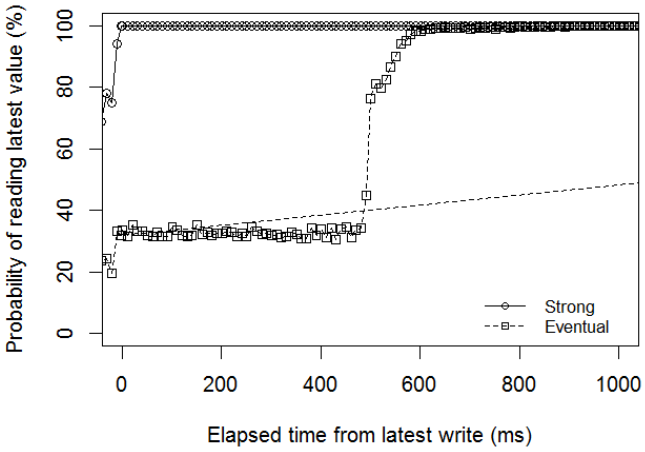


Fig. 8. Probability of reading latest value for each read consistency model.

Please note that the result of this part of our experiment confirms an earlier test done by [5] and verifies the claim by Amazon that strong consistent reads will not return stale value [4]. In addition, from the figure, it can be seen that there are some observations that the elapsed time is negative. This is because that read was initiated after a write began but before the write finished (the read was requested in the middle of a new write) AND the read received the value from the new write. This behavior is entirely possible according to Amazon [4]. Work done by [5] defined elapsed time differently than ours so this problem may not happen with them. However, since the number of observations with negative elapsed time is around 1%, they have negligible effects on the result.

## VI. Discussions

From the analysis, it is clear that eventual and strong consistent reads have the same performance. Each read generally takes less than 24 milliseconds to complete, only 8% of the reads may take longer. However, the worst read time can goes up to one second range. Write time on the average is 3 times longer than read time but its variance is also greater. Most writes will complete in 200 milliseconds but the worst writes may take up to 2 seconds.

Throughput of the two types of read also shows similar performance, with identical median, IQR and almost equal mean. Please note that the throughput experiment was not as thorough as it should have been. We cannot determine right now whether the throughput achieved was limited by SimpleDB or by the capability of the VM used. The experiment showed similarity between eventual and strong consistent read performance, but should not be used to gauge the SimpleDB actual throughput. In the future we plan to investigate this issue further.

When we breakdown write and read performance by test run, where each test run was approximately 1 hour apart, it becomes clear that write performance was highly unpredictable with some runs that had median write latency in hundreds. Comparing with read performance by test run which was much more consistent with median around 20 ms in all runs. Looking at read performance alone, it is very difficult to tell whether write performance during the same run was good or bad.

Please also note that the start time of the experiment (start of run number 1) on the two days were different. It is probably pure co-incidence that bad write performance happened on run 11 on both days. (We checked and found that run 11 did not occur during the same time of day on both days.) The data set is also too few to make any analysis on performance pattern.

Another topic that we are going to discuss briefly is the cost of operations. Amazon stated that the cost of reading and writing to SimpleDB did not depend on the number of reads and writes, but on "virtual CPU time" used to service those operations. With only this information it is not clear how much cost a web application will incur when using SimpleDB. Amazon however provides cost explorer application that can summarize cost by each API call. According to this application, the cost of eventual read for the duration of the experiment was $27.15. Since we issued 260,132 eventual read requests, the cost comes to $0.1 per 1,000 request. We can only reliably report the cost of eventual read, because there were errors with strong consistent read and the experiment executed longer than it should have been and write operations were too few to be sure of cost of each write.

## VII. Conclusion

This paper evaluate the performance of three APIs for interfacing with Amazon SimpleDB, which are write, eventual read, and strong consistent read. These APIs together provide developers with either eventual or strong data consistency model. From the results of our experiment, it is shown that the latency and throughput of both read APIs were identical, even though they guarantee consistency differently. The performance of write API was about 3 times worse than reads and had greater variations too.

Our work also confirmed an earlier work about consistency in SimpleDB which showed that the probability of reading the latest write value depends on the time since last write.

## References

[1] A. S. Tanenbaum and M. van Steen, "Distributed Systems: Principles and Paradigms", Second Edition, Prentice Hall, 2006.

[2] Amazon Elastic Compute Cloud , retrieved from http://aws.amazon.com/ec2

[3] Amazon SimpleDB, retrieved from http://aws.amazon.com/simpledb/

[4] Amazon SimpleDB Developer Guide, retrieved from http://docs.aws.amazon.com/AmazonSimpleDB/latest/DeveloperGuide/ConsistencySummary.html

[5] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data Consistency Properties and the Tradeoffs in Commercial Cloud Storages: the

Consumers' Perspective", Proceeding of the 7th Conference on Innovative Data Systems Research (CIDR), 2013.

[6] I. Fetai and H. Schuldt, "Cost-Based Data Consistency in a Data-as-a-Service Cloud Environment", in Proceeding of the Fifth International Conference on Cloud Computing, 2012.

[7] D.B. Terry et. al., "Consistency-Based Service Level Agreements for Cloud Storage", in Proceeding of the ACM Symposium on Operating Systems Principles", 2013.

[8] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: pay only when it matters. Proceedings International Conference on Very Large Data Bases (VLDB), August 2009.