Data Consistency Properties and the Trade-offs in Commercial Cloud Storages: the Consumers' Perspective

Hiroshi Wada[*] [†], Alan Fekete[‡], Liang Zhao[†] [*], Kevin Lee[*] and Anna Liu[*] [†]
[*] National ICT Australia – NICTA
[†] School of Computer Science and Engineering, University of New South Wales
[‡] School of Information Technologies, University of Sydney
[*] {firstname.lastname}@nicta.com.au
[‡] {firstname.lastname}@sydney.edu.au

ABSTRACT

A new class of data storage systems, called *NoSQL* (Not Only SQL), have emerged to complement traditional database systems, with rejection of general ACID transactions as one common feature. Different platforms, and indeed different primitives within one NoSQL platform, can offer various consistency properties, from Eventual Consistency to single-entity ACID. For the platform provider, weaker consistency should allow better availability, lower latency, and other benefits. This paper investigates what consumers observe of the consistency and performance properties of various offerings. We find that many platforms seem in practice to offer more consistency than they promise; we also find cases where the platform offers consumers a choice between stronger and weaker consistency, but there is no observed benefit from accepting weaker consistency properties.

1. INTRODUCTION

Cloud computing is attracting interest through the potential for low cost, unlimited scalability, and elasticity of cost with load [7, 8, 11, 33]. A wide variety of offerings are typically categorized as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). IaaS is exemplified by Amazon Web Services (AWS), and provides the capability to execute existing programs on a virtual machine that is essentially the same as a standard box with a standard operating system. The consumer has control over the virtual resources. Each PaaS system offers a distinctive set of functionalities as an API, that allow programs to be written specially to execute in the cloud; Google AppEngine (GAE) is an example of this approach.

In PaaS systems, a persistent and scalable storage platform is a crucial facility. In an IaaS environment, one could simply install an existing database engine such as MySQL in one's virtual machine instance, but the limitations (performance, scale, and fault-tolerance) of this approach are well-known, and the traditional database systems can become a bottleneck in a cloud platform [2, 27]; thus novel storage platforms are commonly offered within IaaS clouds too. These storage platforms operate within the cloud platform, and take advantage of the scale-out from huge numbers of cheap machines; they also internally have mechanisms to tolerate the faults that are inevitable with so many unreliable machines. Examples include Amazon SimpleDB[1], Microsoft Azure Table Storage[2], Google App Engine datastore[3], and Cassandra[4]. A term often applied to these storage platforms is *NoSQL* (Not Only SQL). NoSQL database systems are designed to achieve high throughput and high availability by giving up some functionalities that traditional database systems offer such as joins and ACID transactions. NoSQL data stores may offer weaker consistency properties, for example eventual consistency [32]. A client of such a store may observe values that are stale, not from the most recent write. This design feature is explained by the CAP theorem, which states that a partition-tolerant distributed system can guarantee only one of the following two properties: data consistency, or availability [17]. Many of NoSQL database systems aim for availability and partition tolerance as their primary focus and thus they relax the data consistency constraints.

It is a new challenge for developers to write applications that use storage offering weak consistency. For example, recent work by Hellerstein [19] has identified a class of monotonic programs that give correct results on eventual consistent data. The application designer therefore tries to express their computational task using only monotonic operations. Further complicating the programmer's task, there are variant consistency properties that may or may not be provided, such as read-your-writes, monotonic reads, or session consistency, each changing the set of possible situations, and thus what the code must be written to handle. The effort of coding for a weak consistency model is typically justified by pointing to corresponding tradeoffs, such as better availability, lower latency, etc [16].

We have experimentally investigated these issues, from the view of the consumer of the storage facilities. That is, we try to see what kinds of inconsistency are seen *in the results returned from operations*, and how frequently these situations arise. This contrasts with research on cloud-based

CIDR'11 Asilomar, California, January 2011

[1]aws.amazon.com/simpledb/
[2]www.microsoft.com/windowsazure/
[3]code.google.com/appengine/
[4]cassandra.apache.org/

storage platforms [9, 10, 12] that takes the view of the platform owner and focuses on algorithms and the properties of the data held in various replicas within the platform.

Our main contributions are detailed measurements over several storage platforms, that show how frequently, and in what circumstances, different inconsistency situations are observed, and what impact the consumer sees on performance properties from choosing to operate with weak consistency mechanisms. The overall methodology of our experiments, for measuring consistency as seen by a consumer, is another contribution. In Section 2 we report on the experiments that investigate how often a read sees a stale value. For several platforms, data is always, or nearly always, up-to-date. For one platform (SimpleDB), we often see stale data, and so in Section 3 we investigate more deeply the consistency properties of this platform, covering issues such as consistency among multiple data elements, and cases where operations on one element impact on reads of another element. Section 4 then explores the performance of different consistency options; in particular, we investigate whether the consumer is offered any tradeoff in cost or performance, to compensate for using weak consistency operations. Section 5 discusses some limitations to generalising our results. In Section 6 we connect and contrast our work with other research related to this topic. Section 7 gives some conclusions and suggests directions for further study.

2. STALENESS OF DATA

We first investigate the probability of a consumer observing stale data in an item.

Figure 1 illustrates the architecture of the benchmark applications in this study. There are three cloud-deployed roles: the data store, and two computations, *writer* and *reader*. A writer repeatedly writes 14bytes of string data into a particular data element; the value written is the current time, so that we can easily check which write is observed in a read. In most of the experiments we report, writing happens once every three seconds. A reader role repeatedly reads the contents from the data element and also notes the time at which the read occurs; in most experiments reading happens 50 times every second. In some of our experiments, we use one thread for the writer role and one or multiple threads each implementing the reader role, in other experiments we have a single thread that takes both roles. We refer to one "measurement" of the experiment as running the writing and reading for 5 minutes, doing 100 writes and 15,000 reads. We repeated the measurement once every hour, for at least one week, in October and November 2010. In a post-processing data analysis phase, each read is determined to be either fresh (if the value observed has a timestamp from the closest preceding write operation, based on the times of occurrence) or stale; also each read is placed in a bucket based on how much clock-time has elapsed since the most recent write operation. By examining all the reads within a bucket, from a single measurement run, or indeed aggregating over many runs, we calculate the probability that a read which occurs a given time after the write, will observe the freshest value. Repeating the experiment through a week ensures that we will notice any daily or weekly variation in behavior.

2.1 Amazon SimpleDB

SimpleDB is a distributed key-value store offered by Amazon. Each key has associated a collection of attributes, each with a value. For these experiments, we take a data element to be a particular attribute kept for a particular key (a key identifies what SimpleDB calls an item). SimpleDB supports (among other calls) a write operation (`PutAttributes`) and two types of read operations, distinguished by a parameter in the call to `GetAttributes`: *eventual consistent read* and *consistent read*. The consistent read is supposed to ensure that the value returned always comes from the most recently completed write operation, while an eventually consistent read does not give this guarantee. Our study investigates how these differences appear to the consumer of data.

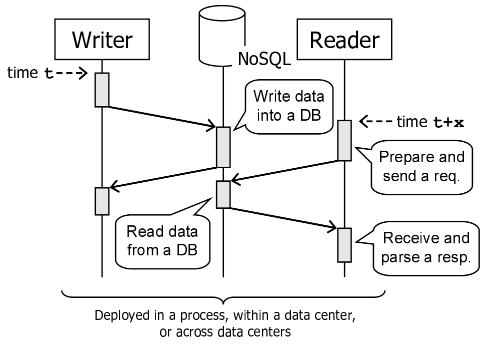

Figure 1: The Architecture of Benchmark Apps

SimpleDB is currently operated in four geographic regions independently (i.e., US West, US East, Ireland and Singapore) and each of them offers a distinct URL as its access point. For example, `https://sdb.us-west-1.amazonaws.com` is the URL of SimpleDB operated in US West. We used this as the data store for our experiments. Writer and reader are implemented in Java and run in EC2; they access SimpleDB in US West through its REST interface.

2.1.1 *Accessing from a Single Thread*

In the first study, we run the writer and reader in the same single thread on an `m1.small` instance provided by EC2 with Ubuntu 9.10. The writer/reader process is deployed in US West. We cannot be sure that the SimpleDB data store will be in the same physical data center as the computation [5], but using the same geographic region is the consumer's best mechanism to reduce network latency.

We executed a measurement run once every hour for 11 days from Oct 21, 2010. In total 26,500 writes and 3,975,000 reads were performed. Since we use only one thread in this study, the average throughput of reading and writing are 39.52 per second and 0.26 per second, respectively. (Each measurement runs at least five minutes.) The same set of measurements was performed with eventual consistent read and with consistent read.

2.1.1.1 *Read-Your-Write Consistency.*

Figure 2 and Table 1 show the probability of reading the fresh value plotted against the time interval that elapsed from when the write begins, to the time when the read is submitted. Each data point in the graph is an aggregate over all the measurements for a particular bucket containing all time intervals that agree to millisecond granularity; in the Table we aggregate further, placing all buckets whose time is in a broad interval together, and here we also show actual numbers as well as percentages. With eventual consistent read the probability stays about 33% from 0ms to 450ms. It jumps up sharply between 450ms and 500ms, and it reaches 98% at 507ms. A spike and a valley in the first 10ms are perhaps random fluctuation due to a small number of data points. With consistent read, the probability is 100% from about 0ms onwards.

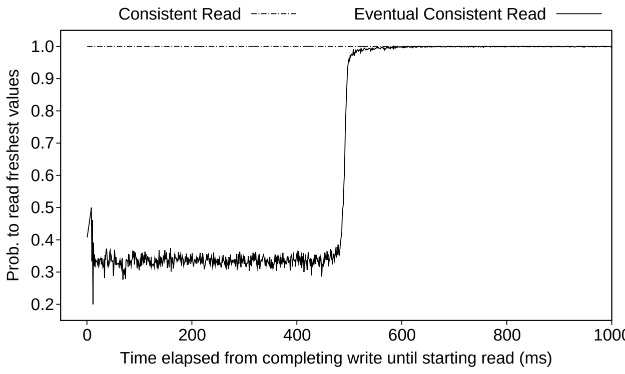

Figure 2: Probability of Reading Freshest Value

Table 1: Probability of Reading Freshest Value

Time Elapsed from Starting Write Until Starting Read	Eventual Consistent Read	Consistent Read
[0, 450)	33.40% (168,908/505,821)	100.00% (482,717/482,717)
[500, 1000)	99.78% (1,192/541,062)	100.00% (509,426/509,426)

A relevant consistency property is "read-your-writes", which says that when the most recent write is from the same thread as the reader, then the value seen should be fresh. As we find that stale eventual consistent reads are possible with SimpleDB within a single thread, so we conclude that eventual consistent reads do not satisfy read-your-writes; however, consistent reads of course do have this property.

We now consider the variability of the time when freshness is possible or highly likely, among different measurement runs. For eventual consistent reads, Figure 3 shows the first time when a bucket has freshness probability that is over 99%, and the last time when the probability is less than 100%. Each data point is obtained from a five minutes measurement run, so there are 258 data points in each time-series. The median of the time to exceed 99% is 516.17ms and coefficient of variance is 0.0258. There does not seem to be any regular daily or weekly variation, rather the outliers seem randomly placed. Out of the 258 measurement runs, 2 runs (0.78%) and 21 runs (8.14%) show a non-zero probability of stale read after 4000ms and 1000ms, respectively. Those outliers are considered to be generated by network jitter and similar effects.

2.1.1.2 Monotonic Read Consistency.

One consistency property that has been considered important [32] is "monotonic read", where a following operation sees data that is at least as fresh as what was seen before. This property can be examined across multiple data elements, or for a single element as we consider here. We find that consistent reads are monotonic as they should be, since each read should always see the most recent value. However, eventual consistent reads are not monotonic, and indeed the

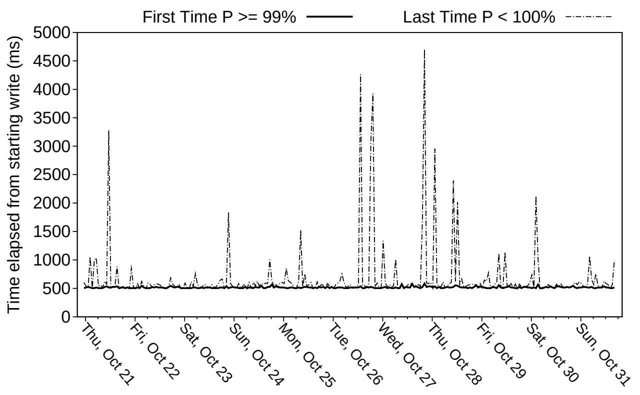

Figure 3: Time to See Freshness (Eventual Consistent Read)

freshness of a successive operation seems essentially independent of what was seen before. Thus eventual consistent reads also do not have stronger properties like causal consistency.

Table 2 shows the probability of observing fresh or stale values in each pair of successive eventual consistent reads performed during the range from 0ms to 450ms after the time of a write. The table also shows the actual number of observations out of 475,575 of two subsequent reads performed in this measurement study. The monotonic read condition is violated (that is, the first read returns a fresh value but the second read returns a stale value) in 23.36% of pairs. This is reasonably close to what one would expect of independent operations, since the probability of seeing a fresh value in the first read is about 33% and the probability of seeing a stale value in the second read is about 67%. The Pearson correlation between the outcomes of two successive reads is 0.0281, which is very low, and we conclude that eventual consistent reads are independent from each other.

Table 2: Successive Eventual Consistent Reads

Second Read / First Read	Stale	Fresh
Stale	39.94% (189,926)	21.08% (100,1949)
Fresh	23.36% (111,118)	15.63% (74,337)

2.1.2 Accessing from Multi Threads and Processes

In the previous results, all read and write requests originate from the same thread. We did measurements for four other configurations:

1. A writer and a reader run in different threads in the same process,
2. A writer and a reader run in different processes on the same virtual machine in the same geographic domain as the data storage (US West),
3. A writer and a reader run on different virtual machines in US West, or
4. A writer and a reader run on different virtual machines, one in US West and one in Ireland.

In the first two cases, read and write requests are originated from the same IP address. In the third case, requests are originated from different IP addresses but from the same geographical region. In the last case, requests are originated from different IP addresses in different regions.

Each experiment was run for 11 days as well. In all four cases the probability of reading updated values shows a similar distribution as in Figure 2. We conclude that consumers of SimpleDB see the same data consistency model regardless of where and how clients are placed.

2.2 Amazon S3

A similar measurement study was conducted on Amazon S3 for 11 days. In S3, storage consists of objects within buckets, so our writer updates an object in a bucket with the current timestamp as its new value, and each reader reads the object. In this experiment, we did measurements for the same five configurations as SimpleDB's case, i.e., a write and a reader run in a single thread, different threads, different processes, different VMs or different regions. Amazon S3 two types of write operations: *standard* and *reduced redundancy.* A standard write operation stores an object so that its probability of durability is at least 99.999999999%, while a reduced redundancy write aims at giving at least 99.99% probability of durability. The same set of measurements was performed with standard write and reduced redundancy write.

Documentation states that Amazon S3 buckets provide eventual consistency for overwrite put operations [4]; however, no stale data was ever observed in our study regardless of write redundancy options. It seems that staleness and inconsistency might be visible to a consumer of Amazon S3 only in executions such that there is a failure in the particular nodes of platform where the data is stored, during the time of their access; this seems a very low probability event.

2.3 Azure Table and Blob Storage

The experiment was also conducted on Windows Azure table and blob storages for eight days. Since it is not possible to start more than one process on a single VM (Web Role in this experiment), we did measurements for four configurations: a write and a reader run in a single thread, different threads, different VMs or different regions. On Azure table storage a writer updates a property of a table and a reader reads the same property. On Windows blob storage a write updates a blob and a reader reads it.

The measurement study observed no stale data at all. It is known that all types of Windows Azure storages support strong data consistency [24] and our study confirms it.

2.4 Google App Engine Datastore

Similar to SimpleDB, Google App Engine (GAE) datastore keeps key-accessed entities with properties, and it offers two options for reading: strong consistent read and eventual consistent read. However, the behavior we observed for eventual consistent read in GAE datastore is completely different from that of SimpleDB. It is known that the eventual consistent read of GAE datastore reads from a secondary replica only when a primary replica is unavailable [18]. Therefore, it is expected that consumer programmers see consistent data in most reads, regardless of the consistency option they choose.

We ran our benchmark application coded in Java and deployed in GAE. In GAE applications are not allowed to create threads; a thread automatically starts upon an HTTP request and it can run no more than 30 seconds. Therefore, each measurement on GAE runs for 27 seconds and we run measurements every 10 minutes for 12 days. The same set of measurements was performed with strong consistent read and eventual consistent read. Also, GAE offers no option to control the geographical location of applications. Therefore, we did measurements for two configurations: a writer and a reader run in the same application (i.e., thread), or a writer and a reader run in different applications. Each measurement consists of 9.4 writes and 2787.9 reads on average, and in total 3,727,798 reads and 12,791 writes happened on average for each configuration.

With strong consistent read no stale value was observed. With eventual consistent read and both roles in the same application, no stale value was observed. However 11 out of 3,311,081 readings ($3.3E^{-4}\%$) observed stale values when a writer and an eventual consistent reader are run in different applications. We cannot conclude for certain whether stale values might sometimes be observed when a writer and a reader run in the same application; however, it suggests the possibility that GAE offers read-your-writes eventual consistency. In any case, consistency errors are very rare.

3. CONSISTENCY MODEL OF SIMPLEDB

We do not have a public description of the implementation approach used by SimpleDB. However our data from Section 2 shows that eventual consistent read of SimpleDB does not support monotonic reads or read-your-writes. This section investigates in more detail, what the consumer can determine about the data consistency model of SimpleDB eventual consistent read. Section 3.1 investigate the support of monotonic write consistency and Section 3.2 investigates the consistency among multiple data elements.

This section discusses the observations obtained in various studies; we speculate in Section 4.3 on mechanisms that might lead to these observations.

3.1 Monotonic Write Consistency

Vogels [32] has advocated the importance of the "monotonic write" property, because programming is notoriously hard if this is missing. The monotonic write property guarantees to serialize the writes by one process. It is not clear whether a consumer can test this, through looking at the values received in reads. To gain some insight, we ran a similar benchmark to the one which is described in Section 2, except it has only one thread which performs 100 repetitions of a small *cycle*, each of which updates a data element twice in a row and then reads the element repeatedly for three seconds; each read is placed in a bucket depending on the time interval from the start of the cycle till the read is submitted. We ran a measurement once every hour for nine days, and aggregated all the buckets from a given time after the cycle starts.

We refer to the value in the element before the cycle starts as `v0`, the value placed there in the first write as `v1` and then `v2` is written immediately afterwards. Figure 4 shows the probability of reading `v0`, `v1` or `v2` against the time from the start of the cycle. The total probability of reading `v0`, `v1` or `v2` at times between 50ms to 400ms are 0.097%, 66.339% and 33.564%, respectively. It appears that the second write is enough to ensure that no replica any longer contains the value from before the cycle started, even though we are still well below the period of 500ms from the first write, which our earlier data showed was the time till that write would be visible in all replicas. We say that the second write "flushes" the first write to consistency.

When we modify the experiment to write three different values v1, v2 and v3 consecutively, the probability of reading v0, v1, v2 or v3 is 0.051%, 0.028%, 66.650% and 33.272%, respectively. Again, each write except the last seems to have been flushed, and this measurement suggests that each replica almost always contains a value that is no more than one write behind the latest.

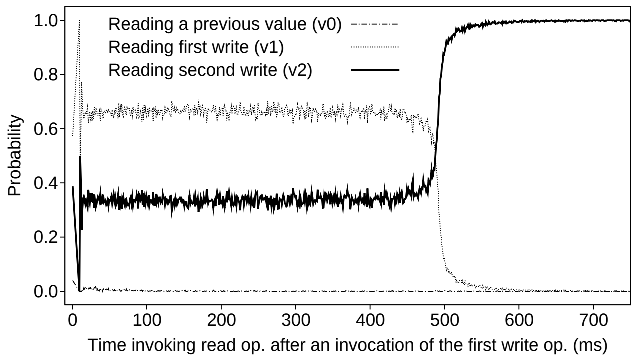

Figure 4: Consecutive Writes to One Item

Another study shows even more complexity in the consistency model that the consumer sees for SimpleDB, because flushing behavior varies depending on the content being written. This is just like the previous experiment, except that the same value is written in both writes of a cycle, that is v1 = v2. Figure 5 shows the probability of reading v0 or v1 over time. The total probability of reading v0 or v1, through the period between 50ms to 400ms after the write, is 66.526% and 33.474%, respectively. Note that this is quite different from what one would see if one just combined the curves for v1 and v2 in Figure 4. In particular, when v1 = v2, after the second write some replicas clearly continue to hold the value from before the cycle, which is not so when v1 ≠ v2. This suggests that the second write is ignored, and does not cause a flush of previous writes, if v1 = v2.

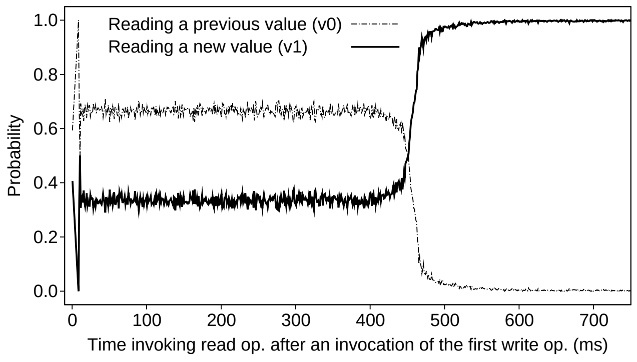

Figure 5: Writing the Same Value Twice

The SimpleDB data model is comprised of domains, items, attributes and values [5]. A domain is a set of items. An item is a set of attribute-value pairs. The write operation of SimpleDB can update multiple attribute-value pairs in an item at once. We use this fact to explore more closely the cases when a second write flushes the value in an earlier write. Our experiment is just like the previous one, except that the first write puts v1 in two attributes, A1 and A2, at once, and the second write puts v2 in only A1. As in the previous experiments, one experiment writes different values, v1 ≠ v2, in the first and second writes. The other experiment writes the same value, v1 = v2, in both writes.

Figure 6 shows the probability of reading v1 or v2 when v1 ≠ v2. The probability of reading a previous value (v0) is not zero; however, they are very small (less than 0.1%) and not shown in this figure. A1 shows similar probability to the one shown in Figure 4. However, A2 shows the complexity in the consistency model. Although A2 is updated only once, by the first write, the probability of reading v1 is very close to 100%. It indicates that the second write (i.e., writing v2 on A1) affects the data consistency of A2, flushing the earlier write, despite the fact that it does nothing in A2.

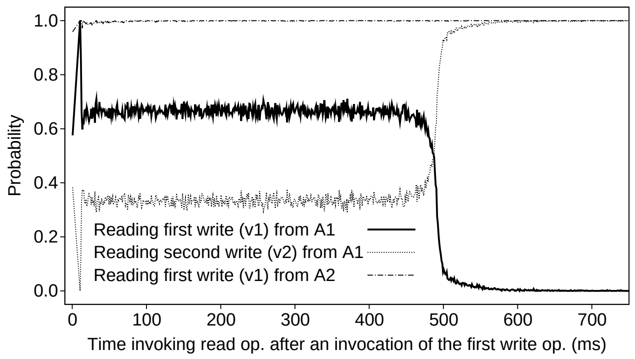

Figure 6: Writing in Two Attributes then in One

When v1 = v2, the probability of reading v1 from A1 and that of reading v1 from A2 draw similar curves as the one in Figure 5. This suggests that the second write is ignored as it is redundant, delivering a subset of the information in the first write.

3.2 Inter-Element Consistency

NoSQL database systems usually provide limited support of transactions; in particular, there is typically avoidance of two-phase commit, and so the platform will typically not allow transactional update to elements that might be stored on different physical nodes. We explore this by having a writer thread modify two data elements (placing the current timestamp in each), and each reader examines those two locations. The SimpleDB data model is comprised of domains, items, attributes and values. We explore the effect of how closely the elements are related in the data model: they might be two attributes within one item, or attributes that are in different items within the same domain, or they might be in different items in different domains.

SimpleDB provides several operations to write and read values from elements; we also explore the effect of using various combinations of these to do writing and reading.

- `PutAttributes` updates attribute-value pairs in a certain item.
- `BatchPutAttributes` performs multiple `PutAttribute` operations in a single call.
- `GetAttributes` returns all attribute-value pairs in a certain item.
- `Select` returns a set of attribute-value pairs in a certain domain that match a query statment.

Table 3 shows the probability observed under different ways to write/read the values in two attribute-value pairs X

Table 3: Write and Read Two Elements in SimpleDB

		two `GetAttributes`				one `GetAttributes`				one `Select`			
		A (%)	B (%)	C (%)	D (%)	A (%)	B (%)	C (%)	D (%)	A (%)	B (%)	C (%)	D (%)
Values in same item	two `PutAttributes`	34.459	65.142	0.138	0.262	33.593	66.246	0.000	0.161	33.559	63.808	0.000	2.633
	one `PutAttributes`	12.050	21.820	22.554	43.576	33.859	0.000	0.000	66.141	33.756	0.000	0.000	66.244
	one `BatchPutAttributes`	11.789	21.964	22.507	43.740	33.649	0.000	0.000	66.351	33.610	0.000	0.000	66.390
Values in diff items in a domain	two `PutAttributes`	34.443	65.138	0.149	0.271	N/A	N/A	N/A	N/A	32.908	66.832	0.000	0.260
	one `BatchPutAttributes`	11.872	21.918	22.471	43.738	N/A	N/A	N/A	N/A	33.763	0.000	0.000	66.237
Values in diff domains	two `PutAttributes`	14.097	24.882	20.740	40.282	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	two `BatchPutAttributes`	14.187	24.924	20.740	40.149	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

and `Y`. N/A means this combination is infeasible; for example, it is impossible to read two values from different domains by one `GetAttributes` call. For each approach we show 4 probabilities: A for when the values of `X` and `Y` are both fresh, B when `X` is fresh but the value of `Y` is stale, C for the case where the value of `X` is stale but the value of `Y` is fresh, and finally D when both `X` and `Y` are stale. We run a measurement once every hour for seven days and obtained the total probability for reads that occur from 0ms to 450ms after the time of a write.

There are three distinct patterns observed. The first pattern is that the probabilities of A, B, C and D are about 12%, 22%, 22% and 44%, respectively. The second pattern is that the probabilities are about 34%, 0%, 0% and 66%. The third pattern is that the probabilities are about 34%, 66%, 0% and 0%.

The first pattern is what one would expect given independence between the items, based on the 33% probability for a single read seeing a fresh value. For example, when updating two attribute-value pairs with a single call of `PutAttributes` or `BatchPutAttributes` and then reading them with two consecutive calls of `GetAttributes`, the probability of reading fresh `X` and `Y` is about 12% (close to 33% × 33%).

The second pattern shows interaction between the items; both `X` and `Y` are stale or both are fresh. For example, it is observed when updates are done with one `PutAttributes`, and reading with a single call of `GetAttributes` or `Select`. The third pattern holds when two operations are used to do the writing; no matter how the reading is done, we see here that the first item has almost 100% chance of freshness, and the second has about 34% chance of freshness. This is the same phenomenon observed in Figure 6 with data no more than one write behind the current value.

4. TRADE-OFF ANALYSIS OF SIMPLEDB

The previous sections show the behavior of SimpleDB for different read options. For the platform provider, there are added costs for stronger consistency options (less availability, higher latency) [1]. We wish to see however what the consumer experiences, as this is what will guide the users of SimpleDB make a well-informed decision about which consistency option to ask for when reading.

We used the benchmark architecture described in Section 2. The measurement ran between 1 and 25 virtual machines in US West to write and read one attribute (which is a 14bytes string data) from an item in SimpleDB. Each virtual machine runs 100 threads, i.e., emulated clients, each of which executes one read or write request every second in a synchronous manner. Thus, if all requests' response time is below 1,000ms, the throughput of SimpleDB can be reported as 100% of the potential load. Three different read-write ratios were studied: 99% read and 1% write, 75% read and 25% write, and 50% read and 50% write cases. We run a measurement, which runs for five minutes with a set number of virtual machines, once every hour for one day.

4.1 Response Time and Throughput

The benefit of eventual consistent read in SimpleDB is explained as follows [5].

> The eventually consistent read option maximizes your read performance (in terms of low latency and high throughput).
>
> Since a consistent read can potentially incur higher latency and lower read throughput it is best to use it only when an application scenario mandates that a read operation absolutely needs to read all writes that received a successful response prior to that read.

To test this advice, we investigated the difference in response time, throughput and availability of the two options, as offered load increased. Figure 7 shows the average, 95 percentile and 99.9 percentile response time of eventual consistent reads and consistent reads at various levels of load. The result is obtained from the case of 99% read ratio and all failed requests are excluded. The result shows no visible difference in average response time; however, consistent read slightly *outperforms* eventual consistent read in 95 percentile and 99.9 percentile response time.

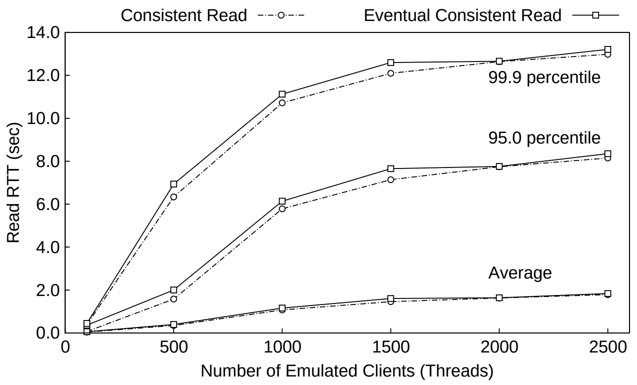

Figure 7: Response Time of Read on SimpleDB

Figure 8 and 9 show the average response time of reads and writes at various read-write ratios, plotted against the number of emulated clients. We conclude that changing the level of update-intensity does not have a marked impact.

Figure 10 shows the absolute throughput, the average number of processed requests per second. We also place whiskers surrounding each average with the corresponding

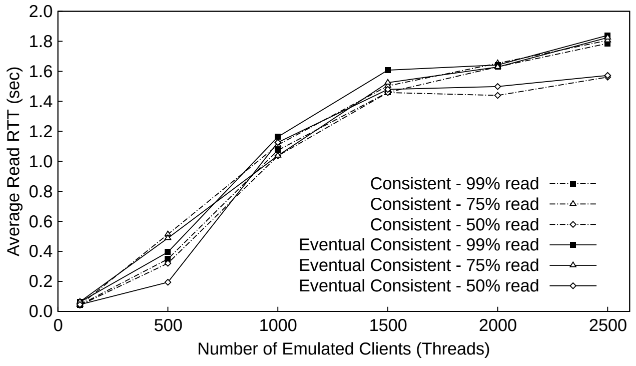

Figure 8: Response Time of Read on SimpleDB

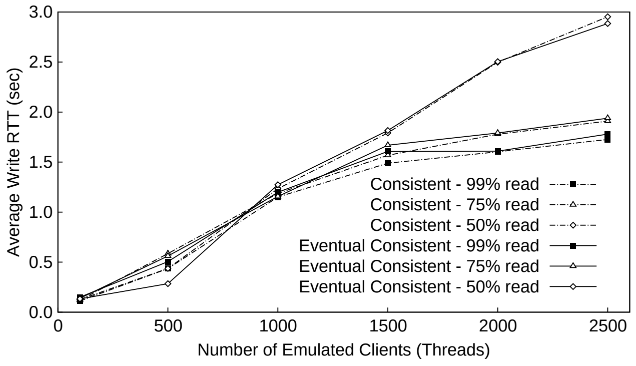

Figure 9: Response Time of Write on SimpleDB

minimum and maximum throughput. Similar to what we saw for response time, the result that consistent read slightly outperforms eventual consistent read, though the difference is not significant. Figure 11 shows the throughput as a percentage of what is possible with this number of clients. As the response time increased, each client sent less than one request every second and, therefore, the throughput percentage decreased.

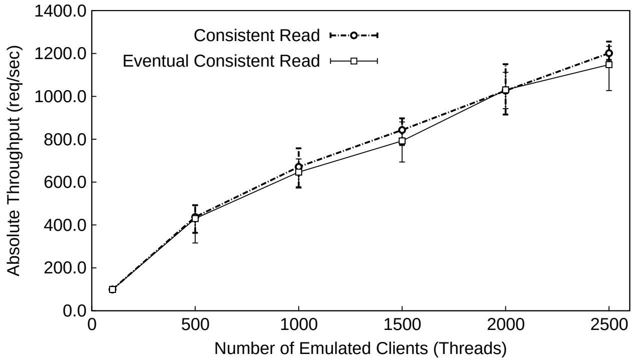

Figure 10: Processed Requests of SimpleDB

We observed that SimpleDB often returns exceptions (with status code 503: "Service is currently unavailable") under heavy load. Figure 12 shows the average failure rates of eventual consistent reads and consistent reads; each data point has whiskers to the corresponding maximum and minimum failure rates. Clearly the failure rate increased as offered load increased, but again we find that eventual consistent read does less well than consistent read, though the difference is not significant.

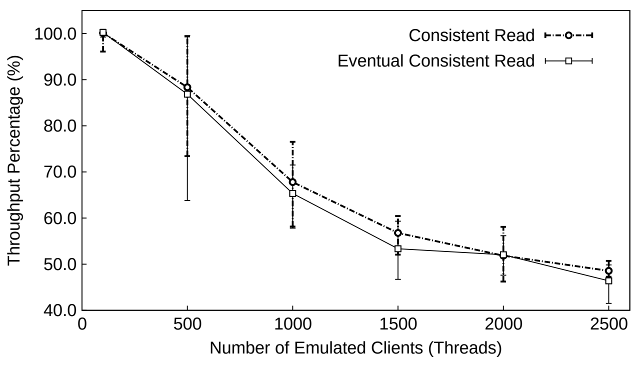

Figure 11: Throughput Percentage of SimpleDB

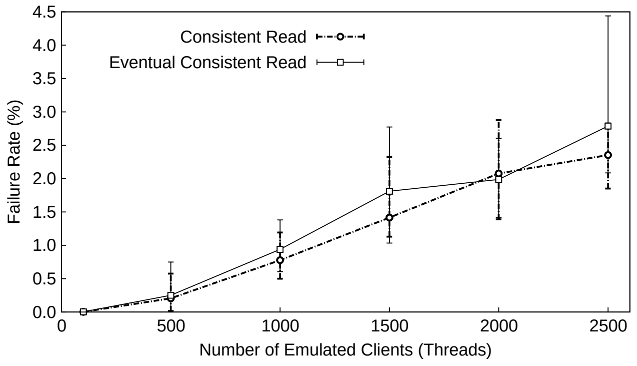

Figure 12: Request Failure Rate of SimpleDB

4.2 Financial Cost

Another way people sometimes view the consistency choice in cloud platforms is as a trade-off against financial cost [22]. In US West region, SimpleDB charges \$0.154 per SimpleDB machine hour, which is the amount of SimpleDB's server capacity used to complete requests, and which can vary depending on factors such as operation types and the amount of data to access. We compared the financial cost of two read consistency options for the runs described above; Amazon reports the SimpleDB machine hour usage, so one can calculate the financial charge incurred for each request. The cost of read operations is constant, at \$1.436 per 10^6 requests, regardless of the consistency options or workload. Also, the cost of write operations is constant at \$3.387 per 10^6 requests as well.

4.3 Implementation Ideas

While our study takes a consumer view of the storage, we have ideas about the implementation based on our experiments. It seems feasible that SimpleDB maintains each item stored in 3 replicas, one primary and two secondaries. We suspect that an eventually consistent read chooses one replica at random, and returns the value found there, while a consistent read will return the value from the primary. This aligns with our experiments showing the same latency and computational effort for the two kinds of read. We are told (James Hamilton, personal communication) that an update is sent *synchronously* to all replicas. We conjecture that the update is applied to the data immediately at the primary replica, but that it remains buffered at the secondaries for a while, explaining the 66% probability of seeing a stale value in an eventual consistent read. Perhaps a write that has been buffered at a replica is applied immediately when any subsequent write operation arrives (even one that is for a different data element), or when a timeout expires (usually 500ms after the write itself). This would explain the experiments of section 3.1 and 3.2, if we assume that all items within one domain are replicated at the same physical nodes, and items in different domains are replicated elsewhere. Further, maybe the system has an optimization that detects redundant write operations, and suppresses them. This must be sophisticated enough to detect not only repeated put requests, but also cases where one put is merely a subset of the previous update. We also suggest, from Table 3, that when multiple attributes are modified within a single operation such as `PutAttributes` or `BatchPutAttributes`, the activity happens with a single message, since in these cases we do not see any immediate application of the writes, but we do see that a following read always observes the same freshness status for each attribute.

5. CAN CONSUMERS RELY ON OUR RESULTS?

Our paper reports on the properties and performance of various cloud-based NoSQL storage platforms, as we observed them during some experiments. A natural concern is whether our results can be extrapolated to predict what the consumer will experience when using one of the platforms. We really can't say!

All the usual caveats of benchmarks measurements apply to us. For example, the workload may be unrepresentative for the consumer's needs, perhaps because in our tests the size of the writes is so small, and the number of data elements is small. Similarly, the metrics quoted may not be what matters to the consumer, the consumer's staff may be more or less skilled in operating the system than we were, perhaps the experiments were not run for long enough and the figures might reflect chance rather than system fundamentals, etc.

As well, there is a particular issue when measuring cloud systems: the vendor might change any aspect of hardware or software without notice to the consumer. For example, even if the algorithm used by a platform currently provides read-your-writes, the vendor could shift to a different implementation that lacked this guarantee. As another example, a vendor that currently places all replicas within a single data center might implement geographical distribution, with replicas stored across data centers for better reliability. Such a change could happen without notice to the consumers, but it might lead to a situation where eventual consistent reads have observably better performance than consistent reads. Similarly, the background load on the vendor's systems might have a large impact, on latency or availability or consistency, but the consumer cannot control or even measure what that load is at any time [29]. For all these reasons, our observations that eventual consistent reads are no better for the consumer, might not hold in the future.

The observations reported in this paper were mainly obtained in October and November, 2011. We had conducted similar experiments in May, 2011. Most aspects were similar between the two sets of experiments, in particular the 500ms latency till SimpleDB reached 99% chance for a fresh response to a read, the high chance of fresh data in eventual consistent reads in S3, Azure and GAE, and the lack of performance difference between SimpleDB for reads with different consistency. Other aspects had changed, for example in the earlier measurements there was less variation in the response time seen by reads on SimpleDB.

6. RELATED WORK

A broad survey of database replication techniques is given in [21].

Many papers have described particular architectures and algorithms for storage in the cloud. These owe much to earlier designs for distributed and especially mobile systems. The concept of eventual consistency arose in work on disconnected operation [13]. Saito and Shapiro offer a valuable survey of techniques that keep replicas loosely synchronized, such as those that provide eventual consistency [28]. Specifically dealing with the cloud, we note several papers from the past five years that describe particular systems: Yahoo!'s PNUTS [10], Amazon's Dynamo [12], Google's Bigtable [9]. The algorithms may be similar to those used in some of the consumer-accessible storage services.

Much research has investigated the performance and cost effectiveness of cloud computation platforms [20,22,30], using benchmark applications simulating typical web applications. For example Kossmann et al use the TPC-W workload with platforms that provide both storage and computation service, and report on throughput (accepted requests per second), financial cost per throughput achieved, and also the variability of the cost. In contrast, our paper focuses directly on NoSQL storage systems, and especially on their consistency properties.

Some previous papers have measured consistency aspects of storage platforms. For a single SQL-interface database engine, Fekete et al [14] define a benchmark that reports how often a consistency condition is violated. They observe rates that depend on the amount of contention between items, and the spacing of read and write operations within a transaction. Considering cloud platforms, Florescu and Kossman [16] argued for the importance of including consistency among the features that are measured, and they suggested that system evaluation should identify the tradeoff between consistency and other properties such as financial cost.

The CloudCmp [25] project benchmarks many features of cloud computing. It includes a measure of "time to consistency" of the storage layer. CloudCmp shows a very different pattern to what we found, and they indicate that the median time to consistency is only about 80 milliseconds. This seems to be because they report the delay from the write until the *first* time that a read returns the up-to-date value, whereas we note that such a read may be followed by others that show stale values; thus we measure the period till *almost all* reads see the recent write. Another difference, though probably not the reason for the different outcomes, is that CloudCmp does an insertion of a new key as the write operation, while we update the value in an existing element. Our work also goes further than CloudCmp by considering more aspects than just reading the recent write; we measure for example properties like monotonicity of reads and inter-element consistency.

A blog [26] reported results, like ours in Section 4, that eventual consistent and consistent reads have similar latency and throughput in SimpleDB. They did not explore the financial costs.

A different approach to measuring consistency of cloud storage platforms is taken by Anderson et al [6], where they record lengthy traces with interleaved operations, and after the fact they check for cycles in various conflict graphs to determine whether various properties hold. The properties they analyse are those that are important in parallel hardware design, such as regular or safe registers, rather than the properties usual in cloud storage platforms such as eventual consistency with monotonic reads.

There is also work on formally defining weak consistency properties. Usually eventual consistency is defined in terms of internal properties such as the state of the replicas, but Fekete and Ramamritham [15] have proposed a definition based only on the results that are returned to the consumer. Extra properties such as session properties, that can strengthen the programmability of eventual consistency, were identified by Terry et al [31]. Awareness of these was spread by the important advocacy of Vogels [32]. Aiyer et al [3] define "consistability" based on the percentage of the operating period in which different consistency models are present.

Kraska et al. [23] consider having a layer above the storage, where different consistency models are supported with different performance properties, and then the client can choose dynamically what is appropriate. They build a theoretical model to analyze the impact of the choice of consistency model in terms of performance and cost, and propose a framework that allows for specifying different consistency guarantees on data. The results discussed in our paper could be used as inputs, i.e., actual behavior, performance and cost of different consistency models, to complement Kraska's work.

In contrast to the NoSQL databases we have studied, Microsoft Azure SQL[5] aims to support the traditional relational model and transactional guarantees in the cloud. It provides strong consistency in reads. However, it has a restriction on the size of the data (a database can grow up to 50GB) and does not support distributed transactions.

7. CONCLUSION

To achieve high availability and low latency, many cloud data storage platforms (or particular operations within a platform) use techniques that avoid two-phase commit and/or synchronous access to a quorum of sites. Thus they can't guarantee strong consistency. It is commonly said that developers should program around this by designing applications that can work with eventual consistency or similar weak models. We have examined the experience of the consumer of cloud storage, in regard to weak consistency and possible performance tradeoffs to justify it. This information should help a developer who is seeking to understand the properties of the new NoSQL storage platforms for the cloud, and who needs to make sensible choices about which storage platform to use.

We found that platforms differed widely in how much weak consistency is seen by consumers. On some platforms, we found that the consumer did not observe any inconsistency or stale data, over several million reads through a week. While inconsistency is presumably possible, it seems very rare; perhaps only happening if there is a failure of one of the nodes or communication links actually used in the computation. Since replication of storage is typically done on 3 or at most 4 nodes, such a failure is unlikely during the computation. Here the risks from inconsistency seem less important compared to other sources of data corruption, such as bad data entry, operator error, customers repeating input, fraud by insiders, etc. Any system design needs to have recourse to manual processes to fix the mistakes and errors from these other sources, and the same processes should be able to cover rare inconsistency-induced difficulties. On these platforms, we wonder whether the developer might sensibly choose to treat eventual consistent reads as if they are consistent, and accept the rare errors as part of doing business.

On Amazon SimpleDB, the consumer who requests eventual consistent reads experiences frequent stale reads and inter-item inconsistency. Also, this choice does not provide other desirable properties like read-your-writes and monotonic reads. Thus the programmer who uses eventual consistent reads must take great care in application design, to code around the dangers of this. However, we found no compensating benefit to the programmer: no reduction in latency, increase in observed availability or lower financial cost, for eventual consistent reads compared to using consistent reads (which are also offered as an option in SimpleDB). There may be benefits to the platform provider when eventual consistent reads are done, but at present these gains seem not to be passed on to the consumer. Thus on this platform in its current implementation, we see no reason for a developer to code with eventual consistent reads.

This work highlights the importance of a research agenda to expand the scope of the service level agreement between cloud provider and customer, to describe more carefully and quantitatively the consistency properties that a platform offers. Tool support for monitoring service levels should also include reporting on consistency aspects. We plan to pursue these ideas.

Acknowledgement

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. We thanks Sherif Sakr and Shirley Goldrei for proofreading.

8. REFERENCES

[1] D. Abadi. Problems with CAP, and Yahoo's little known NoSQL System. `dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html`. [Accessed: Oct 1, 2010].

[2] R. Agrawal et al. The Claremont Report on Database Research. *SIGMOD Record*, 37(3):9–19, 2008.

[3] A. S. Aiyer, E. Anderson, X. Li, M. A. Shah, and J. J. Wylie. Consistability: Describing usually consistent systems. In *Usenix Workshop on Hot Topics in Systems Dependability (HotDep'08)*, 2008.

[4] Amazon Web Services. S3 FAQs. `aws.amazon.com/s3/faqs/`. [Accessed: July 6, 2010].

[5] Amazon Web Services. SimpleDB FAQs. `aws.amazon.com/simpledb/faqs/`. [Accessed: July 6, 2010].

[6] E. Anderson, X. Li, M. A. Shah, J. Tucek, and J. J. Wylie. What consistency does your key-value store

[5] www.microsoft.com/windowsazure/sqlazure/

actually provide? In *Usenix Workshop on Hot Topics in Systems Dependability (HotDep'10)*, 2010.

[7] M. Armbrust et al. Above the Clouds: A Berkeley View of Cloud Computing. Technical report, University of California, Berkeley, Feb 2009.

[8] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.

[9] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. In *USENIX Sym. on Operating Systems Design and Implementation*, 2006.

[10] B. F. Cooper et al. Pnuts: Yahoo!'s hosted data serving platform. In *Proc Very Large Databases (VLDB'08)*, pages 1277–1288, 2008.

[11] M. Creeger. Cloud Computing: An Overview. *ACM Queue*, 7(5):3–4, 2009.

[12] G. DeCandia et al. Dynamo: Amazon's Highly Available Key-Value Store. *Operating Systems Review*, 41(6):205–220, 2007.

[13] A. J. Demers et al. Epidemic algorithms for replicated database maintenance. In *Proc ACM Conference on Principles of Distributed Computing (PODC'87)*, pages 1–12, 1987.

[14] A. Fekete, S. Goldrei, and J. P. Asenjo. Quantifying isolation anomalies. In *Proc Very Large Databases (VLDB'09)*, pages 467–478, 2009.

[15] A. Fekete and K. Ramamritham. Consistency models for replicated data. In *Replication (LNCS 5959)*, pages 1–17. Springer Verlag, 2010.

[16] D. Florescu and D. Kossmann. Rethinking cost and performance of database systems. *SIGMOD Record*, 38(1):43–48, 2009.

[17] S. Gilbert and N. Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News*, 33(2):51–59, 2002.

[18] Google. Datastore Python API Overview. `code.google.com/appengine/docs/python/datastore/overview.html`. [Accessed: Jul 22, 2010].

[19] J. M. Hellerstein. Datalog redux: experience and conjecture. In *Proc. ACM Principles of Database Systems (PODS'10)*, pages 1–2, 2010.

[20] Z. Hill, M. Mao, J. Li, A. Ruiz-Alvarez, and M. Humphrey. Early Observations on the Performance of Windows Azure. In *AMC Workshop on Scientific Cloud Computing*, June 2010.

[21] B. Kemme, R. Jiménez-Peris, and M. Patiño Martínez. *Database Replication (Synthesis Lectures on Data Management 7)*. Morgan and Claypool, 2010.

[22] D. Kossmann, T. Kraska, and S. Loesing. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. In *ACM International Conference on Management of Data*, pages 579–590. ACM, 2010.

[23] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency Rationing in the Cloud: Pay only when it matters. *International Conference on Very Large Data Bases*, August 2009.

[24] S. Krishnan. *Programming Windows Azure: Programming the Microsoft Cloud*. O'Reilly, 2010.

[25] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing Public Cloud Providers. In *Internet Measurement Conference*, page to appear, 2010.

[26] H. Liu. The cost of eventual consistency. `http://huanliu.wordpress.com/2010/03/03/%EF%BB%BFthe-cost-of-eventual-consistency/`. [Accessed Oct 12, 2010].

[27] S. Malkowski et al. Empirical Analysis of Database Server Scalability using an N-tier Benchmark with Read-Intensive Workload. In *ACM Symposium on Applied Computing*, pages 1680–1687. ACM, 2010.

[28] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[29] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. In *Proc Very Large Databases (VLDB'10)*, pages 460–471, 2010.

[30] W. Sobel et al. Cloudstone: Multi-Platform, Multi-Language Benchmark and Measurement Tools for Web 2.0. In *Workshop on Cloud Computing and its Applications*, October 2008.

[31] D. B. Terry et al. Session guarantees for weakly consistent replicated data. In *Proc of International Conference on Parallel and Distributed Information Systems (PDIS'94).*, pages 140–149. IEEE Computer Society, 1994.

[32] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[33] A. Weiss. Computing in the Clouds. *netWorker*, 11(4):16–25, 2007.