Published in proceedings of the sixth Workshop on Hot Topics in System Dependability (HotDep 2010), October 3, 2010, Vancouver, BC, Canada.

What consistency does your key-value store *actually* provide?

Eric Anderson, Xiaozhou Li, Mehul A. Shah, Joseph Tucek, Jay J. Wylie
Hewlett-Packard Laboratories

Abstract

Many key-value stores have recently been proposed as platforms for always-on, globally-distributed, Internet-scale applications. To meet their needs, these stores often sacrifice consistency for availability. Yet, few tools exist that can verify the consistency actually provided by a key-value store, and quantify the violations if any. How can a user check if a storage system meets its promise of consistency? If a system only promises eventual consistency, how bad is it really? In this paper, we present efficient algorithms that help answer these questions. By analyzing the trace of interactions between the client machines and a key-value store, the algorithms can report whether the trace is safe, regular, or atomic, and if not, how many violations there are in the trace. We run these algorithms on traces of our eventually consistent key-value store called Pahoehoe and find few or no violations, thus showing that it often behaves like a strongly consistent system during our tests.

1 Introduction

Internet applications often rely on globally distributed, highly available storage systems to meet the promise of ubiquitous 24x7 operation. The main challenge in building a globally distributed system is dealing with network partitions. Brewer's CAP principle [6] states that any shared data system can provide only two of the following three properties: consistency, availability, and partition tolerance. Since partitions are inevitable in wide-area networks, storage system designers are only left with the option of trading-off consistency for availability. Traditional systems such as databases and file-systems choose to sacrifice availability and only offer strict consistency. However, many Internet services such as social networks or media-sharing sites favor availability over consistency.

To fill this gap, several new key-value stores have been built recently (e.g., Dynamo [11], BigTable [8], PNUTS [9], Cassandra [7], S3 [17]) that offer weaker consistency guarantees in order to provide higher availability. Although many of these stores were initially designed with consistency semantics specific to certain applications, their use has spread widely under the assumption that these weaker semantics are sufficient for many web services. These systems offer a smorgasbord of semantics ranging from row-level atomicity to various forms of eventual consistency [19]. Some systems do not even provide a precise specification of their consistency.

In this paper, we consider how to assess and measure the consistency actually *observed* by the clients when using these key-value stores rather than assuming the worst-case consistency that is promised. Although some applications may tolerate weak consistency, users can still become dissatisfied if inconsistency happens frequently or severely. Knowing the actual consistency delivered has two benefits. First, users can verify whether a key-value store provides its promised level of consistency. Second, users can make an informed decision in choosing among key-value stores, or a service level from a single key-value store (e.g., least expensive but still provides adequate consistency for their workloads).

This paper makes the following contributions: Section 2 presents algorithms that, given a trace of operations obtained from the clients, can verify whether the trace is safe, regular, or atomic [14]. Unlike previous work, our methods incorporate these three successively stronger consistency levels within one simple framework. Moreover, our algorithms quantify the severity of violations. Section 4 then presents experiments with our highly available key-value store, Pahoehoe, using micro-benchmarks similar to those in the YCSB cloud benchmark [10]. These experiments show that even though Pahoehoe only promises eventual consistency, there are relatively few violations of atomicity in failure-free executions. Section 5 discusses future work.

Published in proceedings of the sixth Workshop on Hot Topics in System Dependability (HotDep 2010), October 3, 2010, Vancouver, BC, Canada.

2 Consistency checking

In this section, we present algorithms that check whether a certain consistency level has been achieved in an execution. A key-value store can be viewed as a collection of registers, each identified by a key. Lamport [14] defines three kinds of consistency semantics on registers: safe, regular, and atomic. Lamport's original definition does not allow multiple writers. Pierce and Alvisi [16] have extended the definition to allow multiple writers and we summarize this definition as follows.

An *operation* on the register is either a read (i.e., get) or a write (i.e., put). Using a global clock, we assign each operation a *start* time and a *finish* time. We say operation A *time-precedes* operation B, written as $A < B$, if A finishes before B starts. If neither $A < B$ nor $B < A$, then A and B are said to be *concurrent* with each other, written as $A||B$. It is easy to see that this time precedence relation imposes a partial order on the operations. A *valid* total order is one that conforms to this partial order. Given a total order, each read has a unique most recent write, if any. A register has the following semantics if there exists a valid total order that satisfy the corresponding conditions.

A register is said to be *safe* if a read not concurrent with any writes returns the value of the most recent write, and a read concurrent with some writes returns any value.

A register is said to be *regular* if a read not concurrent with any writes returns the value of the most recent write, and a read concurrent with some writes returns either the value of the most recent write, or the value of one of the concurrent writes.

A register is said to be *atomic* if every read (regardless of whether they are concurrent with any writes or not) returns the value of the most recent write. It is easy to see that safety is weaker than regularity, which is in turn weaker than atomicity.

2.1 High level approach

Our checker is an offline algorithm that processes a global trace of client interactions with a key-value store. We first describe the requirements of traces and then outline our technique for checking consistency and estimating the number of violations.

A global trace consists of a list of read/write (i.e. put/get) requests from all clients to a key-value store. Along with each request, we obtain the value retrieved or stored, and the operation's start and finish times at the client. We obtain a global trace by merging individual traces from multiple clients. As long as the relative orderings and overlapping relations among all operations are preserved, our checker does not need precisely synchronized timestamps across clients. Moreover, since the above semantics are independent of operations across keys, we split the global trace by key and process each resulting trace separately. To distinguish different writes to the same key, we assume that each write writes a unique value, and that before any operations start, each register contains a default value.

Given a trace, our approach to check whether a trace satisfies safe, regular, or atomic semantics is as follows. We first construct a directed graph called the *precedence graph* where a vertex represents an operation and an edge indicates that the source operation should happen before the target operation. We then add three kinds of edges: time, data, and hybrid. As presented below, there are different rules for adding edges in the graph, depending on the desired consistency semantics. If the resulting graph is acyclic (i.e. a DAG), then we conclude that the trace satisfies the consistency semantics in question, otherwise the semantics are violated. To determine if the graph is acyclic, we perform a depth-first search (DFS) on the graph by first treating all added edges as the same and removing duplicate edges between the same two vertices. For proofs of this approach, see the technical report [5].

To quantify the *severity* of consistency violations, we count the number of cycles during the DFS. The counter is incremented whenever a cycle is detected. We realize that this counter is coarse because an edge can be part of multiple cycles, but we believe it is still informative. We plan to devise a more refined metric in the future. For example, one possible refinement is to count the minimum number of edges that need to be removed to make the graph acyclic.

2.2 Checking safety

The following algorithm checks whether a trace is safe:

Step 1. Remove all the reads that are concurrent with some writes from the trace. These reads can be removed without consequence because by the definition of safety, the values returned are allowed to be arbitrary values.

Step 2. (Time rule) For all operations A and B where $A < B$, add an edge $A \rightarrow B$. We call these edges *time edges*.

Step 3. (Data rule for safety and atomicity) For every read R and write W such that R reads W's value, add an edge $W \rightarrow R$. We call these edges *data edges*. We call W the *dictating write* of R, and R one of W's *dictated reads*. Since we assume that every write writes a different value, every read has only one dictating write, but a write can have zero or more dictated reads. If there exists a read that does not obtain any write's value or the default value, we abort the checking by declaring the trace being unsafe. We exit early because now unsafety is caused by the lack of edges, not the excess of them. We note, however, that such violations are unlikely to occur

Published in proceedings of the sixth Workshop on Hot Topics in System
Dependability (HotDep 2010), October 3, 2010, Vancouver, BC, Canada.

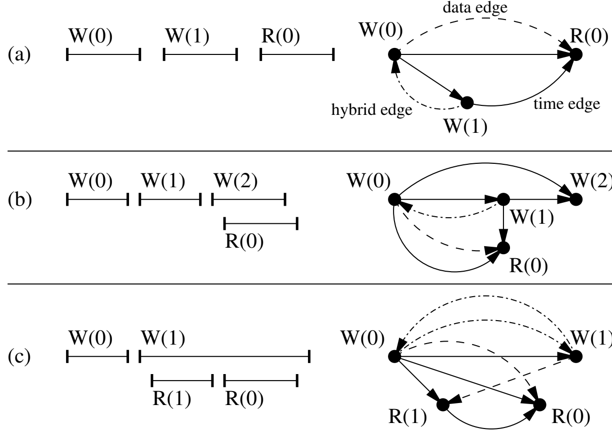

Figure 1: Examples of precedence graphs: (a) An unsafe trace. (b) A safe but not regular trace. (c) A regular but not atomic trace.

in real key-value stores because key-value stores tend to return values that have been previously written.

Step 4. (Hybrid rule for safety and regularity) For every write W' and read R, where $W' < R$, add an edge $W' \to W$, where W is R's dictating write. We call these edges *hybrid edges* (because their existence is due to a combined consideration of time and data precedence). This rule enforces that all writes time-preceding a read should happen before the read's dictating write: otherwise the read would have obtained a different value.

We remark that the hybrid rule is necessary. For example, Figure 1(a) shows an unsafe trace, but only after we apply the hybrid rule do we introduce a cycle. The technical report [5] includes a proof that, after applying all the three rules for adding edges, we have a graph that is a DAG iff the trace is safe.

2.3 Checking regularity

To check regularity, take the safety checking algorithm, numbering as the safety checking algorithm, skip Step 1, and revise Step 3 as follows:

Step 3. (Data rule for regularity) For every read R and its dictating write W, if $W||R$, then do nothing, otherwise add a data edge $W \to R$. Similar to safety checking, if there exists a read that does not obtain any write's value or the default value, we abort the checking.

The technical report [5] contains a proof that this algorithm generates a DAG iff the trace is regular. Figure 1(b) shows a safe but not regular trace. The reader can verify that, if we had included Step 1 (i.e., disregarded $R(0)$), as in the safety checker in the previous section, we would not have been able to detect a cycle.

$\mathbf{A}$:= all intervals in increasing order of start time;
$\mathbf{B}$:= all intervals in decreasing order of finish time;
foreach ($A \in \mathbf{A}$)
  $t := -\infty$;
  foreach ($B \in \mathbf{B}$ such that $B < A$)
    if ($t <$ B's finish time)
      add edge $B \to A$;
      $t := \max(t,$ B's start time);
    else break;

Figure 2: Adding the minimum number of time edges.

2.4 Checking atomicity

To check atomicity, take the safety checking algorithm, skip Step 1, and revise Step 4 as follows:

Step 4. (Hybrid rule for atomicity) For every write W' and read R, if there exists a path from W' to R consisting of either time-edges or data-edges, then add an edge $W' \to W$, where W is R's dictating write.

In other words, instead of checking whether there is an edge from W' to R, we check whether there is a path from W' to R. Figure 1(c) shows a regular but not atomic trace. The reader can verify that, had we used the hybrid rule for safety and regularity, this hybrid edge would not have been added (because there is no time edge from $W(1)$ to $R(0)$).

The technical report [5] includes a proof that the resulting graph is a DAG iff the trace is atomic. A naiv̈e algorithm for adding hybrid edges is based on all-pairs reachability [18], which takes $\Theta(mn) \leq \Theta(n^3)$ time. In fact, an algorithm which instead calculates a DFS starting from every vertex achieves the same bound. For a moderate size graph with $n = 1000$, n^3 is a billion, which is likely to be too large. The n^3 bound follows from the graph having n^2 edges. If we can reduce the number of time edges from n^2 to just n, then the running time of the all-pairs reachability algorithm will be n^2. Figure 2 presents an algorithm that adds the minimum number of edges such that, for all operations A, B such that $A < B$, there is a path from A to B. This algorithm runs in time $O(n \log n)$ plus the number of edges to be added. The reader can find the correctness and optimality proofs of this algorithm in the technical report [5]. To see how this algorithm helps, suppose the operations can be divided into n/k groups, each of which contains k concurrent operations yet each group follows another in sequence. Then this algorithm adds about $k^2 \cdot n/k = nk$ edges. When k is relatively small, nk is close to $O(n)$. Hence, the overall atomicity checking algorithm will run in $O(n^2)$ time. For extremely concurrent traces (e.g., $k = n/2$), however, the atomicity checking algorithm still runs in $O(n^3)$ time.

Published in proceedings of the sixth Workshop on Hot Topics in System Dependability (HotDep 2010), October 3, 2010, Vancouver, BC, Canada.

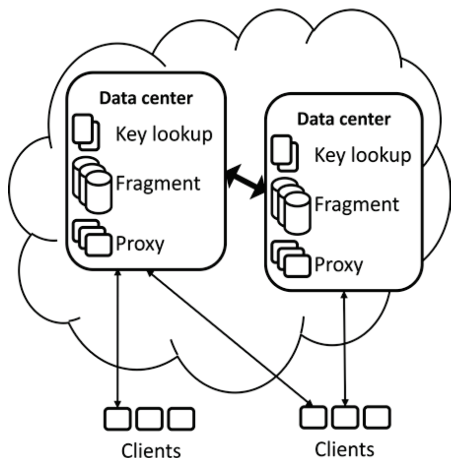

Figure 3: Architecture of Pahoehoe.

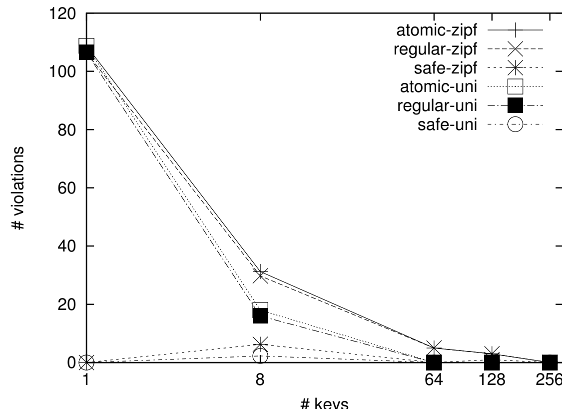

Figure 4: Violations vs. number of keys, 128 concurrent processes.

3 Pahoehoe

Pahoehoe, shown in Figure 3 is a cloud storage system designed to offer extreme availability ($>$ 4 nines), to span multiple data centers, and to scale to hundreds of petabytes using low-end commodity components. It offers a key-value-based get-put interface.

Pahoehoe is composed of three main entities: proxies, key lookup servers (KLS), and fragment servers (FS). On a put, the proxy splits the value into multiple erasure-coded fragments. The FSs are responsible for storing the fragments, which form the bulk of the data. The KLSs maintain a mapping of the user-provided keys to the locations of corresponding fragments. In a typical setup, each data center has a few KLSs for availability and many FSs for reliability and scaling capacity.

Currently, Pahoehoe only guarantees eventual consistency because availability is paramount for our initial applications, which can tolerate temporary inconsistency. Its protocols are eager in that they provide a useful result as soon as possible, thus providing the highest availability. For example, a put returns success as soon as it has updated any one of the KLSs and a minimum number of FSs to ensure that the value is durable. The remainder of the put completes in the background. A get will try the list of values referenced by the first KLS that responds, from newest to oldest, and will return as soon as it succeeds. If none of the referenced values are available, it tries contacting other KLSs. Thus, puts can return success before they are complete, and repeated gets may sometimes return earlier versions after newer ones. More details of these protocols can be found in [4].

4 Consistency provided by Pahoehoe

In this section, we present experiments that quantify the actual consistency provided by Pahoehoe. Our setup consists of 8 machines, 4 KLSs and 4 FSs, connected via gigabit Ethernet. We mimic a two data-center environment with half the KLSs and FSs on each side of a wide-area link. The link, emulated using NetEm, is configured to 50ms average latency. Each machine has 2 dual-core AMD Opterons 2216, 8GB RAM, and runs 64-bit Debian Linux 2.6.26. The FSs have 4x1TB 7.2K RPM SCSI disks, and the KLSs have 4x15K RPM disks in RAID-10.

We use a workload similar to the YCSB [10] micro-benchmarks. We have a single client machine (with 4 dual-core AMD 8220) that runs many (up to 128) concurrent processes issuing operations in a closed loop. The operations are a mix of gets and updates, where an update is a get followed by a put. For all these experiments, we run 1000 operations with a 40%-60% get-update mix (i.e. 70%-30% get-puts as the 60% updates are in fact 30% gets and 30% puts) and each value is 128KB. We mirror values across data centers (one replica per data center); puts complete as soon as one FS receives a value and one KLS has the locations. We vary the number of keys, the distribution for choosing keys (uniform and Zipfian), and concurrency level. In our experiments, we co-locate the client machine with the proxy and the traces are collected at the proxy. We use Chirp [3] to merge the proxy traces into a global, well-synchronized trace. Chirp is a toolset that can calibrate machine clocks in a data-center, post-collection to within micro-seconds. As in Section 2, a *violation* means a cycle in the precedence graph.

Figure 4 shows the number of violations of each consistency level as we vary the number of keys and distribution. We find $<$ 10% violations even when there is only one key and all 128 processes continually read and

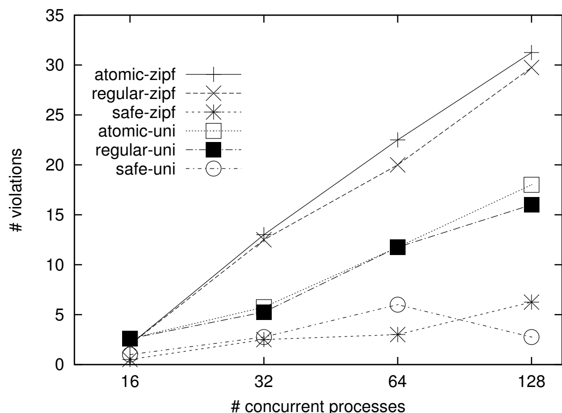

Figure 5: Violations vs. concurrency, 8 keys.

write that key. Also, violations drop to zero when there are more keys than concurrent requests. Even though operations in the Zipfian workload focus on a few items, violations drop quickly. Figure 5 shows the number of violations as we vary the number of processes. As expected, violations increase with number of concurrent operations, and there are more violations with the Zipfian workload.

We note that there are only slightly more atomicity violations than regularity violations because of our system implementation. For a trace to have an atomicity but not a regularity violation, there must be a read that returns the value of a concurrent write. A put tends to finish relatively quickly in our system because puts return success optimistically. Thus, a get only has a short window in which to retrieve a value written in a concurrent put and a put is concurrent with only a few reads, resulting in few atomic but not regular violations. Moreover, we see much fewer safety violations because in our system, it is unlikely for a read to obtain a future write's value, or a past write's value that has been replaced. Nonetheless, these results confirm what many web applications rely on today. In failure-free conditions, an eager, eventually consistent protocol often achieves strong consistency.

5 Discussion and Future Work

Obtaining a global trace. Our checker requires a global trace merged from traces on different machines, with potential clock skew. We used the Chirp toolset [3] to calibrate the timestamps obtained from different machines, but we could also have just used NTP to achieve sufficient synchronization.

Measuring consistability. There are many types of failures in a data-center: disks failure, node failure, network partitions, link flapping, etc. How these failure conditions affect the observed consistency in weakly consistent systems is still unexplored. Going further, we previously proposed the notion of consistability [2]; the ability of a system to offer weaker consistency rather than becoming unavailable under failure. Consistability allows us to simultaneously compare the consistency and performance provided by different systems. This work has shown how to measure the consistency provided solely from the user's perspective.

Other forms of consistency. In this paper, we evaluated atomic, regular, and safe consistency. Many other forms of consistency have been proposed in the literature. We plan to investigate what other semantics can be checked efficiently and design algorithms to check those semantics.

Monitoring consistency. Currently, our method for checking consistency is an offline technique, which has certain limitations. For example, consistency violations often occur during unfavorable operating conditions such as network partitioning, which does not happen very often. Consequently, in order to catch violations, one needs to process a trace that spans a long period. Such a trace may overwhelm the offline checker because the checker analyzes the trace as a whole.

We believe our techniques can be extended to provide online checking of consistency. Namely, we can equip users with a program that monitors the operations and issues an alarm whenever a violation is detected. Such an extension will prove useful in two ways. In the cloud, all services are external, and users often want to monitor the services to ensure that they meet their SLAs. Today, these SLAs are expressed primarily by performance metrics, e.g., latency. With an online checker, these SLAs could also include consistency to see if users are getting what was promised. An online checker could also change the way we architect our applications. For a system that offers multiple consistency levels, instead of always choosing the most conservative semantics, applications could monitor the consistency and change the level as needed. They could react to the needs of users, or react to changes in the underlying system. For example, in failure-free situations, applications could be optimistic and as violations increase use more conservative levels.

6 Related work

Brewer's CAP principle [6], later formalized and proven by Gilbert and Lynch [12], states that among consistency, availability, and partition-tolerance, only two of these three properties can be attained simultaneously. Since partition-tolerance is a must for global systems, most key-value stores favor availability over consistency, and many only promise eventual consistency [19]. We have implemented Pahoehoe in a similar spirit, but we plan to extend the system to provide get/put operations

Published in proceedings of the sixth Workshop on Hot Topics in System Dependability (HotDep 2010), October 3, 2010, Vancouver, BC, Canada.

with stronger consistency semantics. Different key-value stores have chosen to provide different levels of consistency. For example, Amazon's S3 [17] and Dynamo [11] only promise eventual consistency. Google Storage [13] provides read-your-writes consistency. Cassandra [7] provides quorum-based consistency and eventual consistency. Voldemort [20] uses vector clocks to version data items and resolve conflicts at read time (which may require user intervention). Finally, many cloud systems are starting to provide atomic primitives that applications can use to implement strong consistency.

In a seminal paper, Misra presented an elegant algorithm [15] for checking atomicity. His algorithm works by reasoning about the *values* of the register. The observation is that, at some point during the span of an operation, the register assumes the value of the operation (either read or write). Atomicity stipulates that if a value is replaced by another value, then the old value is not allowed to re-appear in the future. Therefore, if a trace violates this condition, then it is not atomic. Somewhat surprisingly, if a trace does not violate this condition, then it is atomic. In contrast, our algorithms reason about the operations. We choose to reason about operations but not values because we aim to provide a common framework to check a variety of semantics, many of which (e.g., safety and regularity) were introduced after Misra's paper. It is not immediately clear to us how to extend Misra's algorithm to check, say, regularity, because for regularity, a replaced value is allowed to re-appear.

Lamport [14] defines three consistency semantics: safety, regularity, and atomicity. Besides these three, additional semantics have been introduced in the literature. For example, k-atomicity, proposed by Aiyer et al. [1], bounds the number of distinct recently completed writes that may be returned by a read operation, and k-regularity can be similarly defined. We could devise algorithms to check for these semantics.

7 Concluding remarks

We have presented algorithms that help users check whether a key-value store provides safe, regular, or atomic consistency semantics. We have found that traces from Pahoehoe, an eventually consistent store, often exhibit strong consistency during failure-free runs of our cloud-based micro-benchmarks. We plan to extend this work in at least two ways. First, we plan on devising efficient online algorithms for checking consistency. Second, we plan to extend the checker for other kinds of consistency proposed in the literature.

References

[1] A. Aiyer, L. Alvisi, and R. A. Bazzi. On the availability of non-strict quorum systems. In *Proc. 19th DISC*, pages 48–62, September 2005.

[2] A. S. Aiyer, E. Anderson, X. Li, M. A. Shah, and J. J. Wylie. Consistability: Describing usually consistent systems. In *HotDep*. USENIX Association, 2008.

[3] E. Anderson, C. Hoover, X. Li, and J. Tucek. Efficient Tracing and Performance Analysis for Large Distributed Systems. In *MASCOTS '09: Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation*, September 2009.

[4] E. Anderson, X. Li, A. Merchant, M. A. Shah, K. Smathers, J. Tucek, M. Uysal, and J. J. Wylie. Efficient eventual consistency in Pahoehoe, an erasure-coded key-blob archive. In *DSN'10*, pages 181–190, June 2010.

[5] E. Anderson, X. Li, M. Shah, J. Tucek, and J. J. Wylie. What consistency does your key-value store actually provide? Technical Report HPL-2010-98, Hewlett-Packard Laboratories, 2010.

[6] E. Brewer. Towards robust distributed systems, 2000. Available at `http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf`. Accessed May 2010.

[7] Cassandra. Available at `http://incubator.apache.org/cassandra/`. Accessed December 2009.

[8] F. Chang, J. Dean, S. Ghemawhat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI'06*, pages 205–218, November 2006.

[9] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB'08)*, pages 1277–1288, August 2008.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, 2010.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP'07*, pages 205–220, October 2007.

[12] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services. *ACM SIGACT News*, 33(2):51–59, June 2002.

[13] Google Storage for Developers. Available at `http://code.google.com/apis/storage`. Accessed May 2010.

[14] L. Lamport. On interprocess communication, Part I: Basic formalism and Part II: Algorithms. *Distributed Computing*, 1(2):77–101, June 1986.

[15] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142–153, January 1986.

[16] E. Pierce and L. Alvisi. A recipe for atomic semantics for Byzantine quorum systems. Technical report, University of Texas at Austin, 2000.

[17] Amazon's Simple Storage Service. Available at `http://aws.amazon.com/s3`. Accessed May 2010.

[18] K. Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theoretical Computer Science*, 58(1–3):325–346, 1988.

[19] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[20] Voldemort. Available at `http://project-voldemort.com/`. Accessed March 2010.