acmqueue Eventually Consistent: Not What You Were Expecting?

Methods of quantifying consistency (or lack thereof) in eventually consistent storage systems

Wojciech Golab, University of Waterloo

Muntasir R. Rahman, University of Illinois at Urbana-Champaign

Alvin AuYoung, HP Labs, Palo Alto

Kimberly Keeton, HP Labs, Palo Alto

Xiaozhou (Steve) Li, Google

Storage systems continue to lay the foundation for modern Internet services such as Web search, e-commerce, and social networking. Pressures caused by rapidly growing user bases and data sets have driven system designs away from conventional centralized databases and toward more scalable distributed solutions, including simple NoSQL key-value storage systems, as well as more elaborate NewSQL databases that support transactions at scale.

Distributed key-value storage systems are among the simplest and most scalable specimens in the modern storage ecosystem. Such systems forgo many of the luxuries of conventional databases, including ACID (atomicity, consistency, isolation, durability) transactions, joins, and referential integrity constraints, but retain fundamental abstractions such as tables and indexes. As a result, application developers are sheltered from technicalities such as normalizing the relational schema, selecting the optimal transaction isolation level, and dealing with deadlocks.

Despite their simple interface and data model, distributed key-value storage systems are complex internally as they must replicate data on two or more servers to achieve higher read performance and greater availability in the face of node or network failures. Keeping these replicas synchronized requires a distributed replication protocol that can add substantial latency to storage operations, especially in geo-replicated systems. Specifically, write operations must update a subset of the replicas before the system acknowledges the completion of the write to the client, and reads may fetch data from a subset of the replicas, adopting the latest value observed (e.g., the one having the highest timestamp) as the response.

As it turns out, choosing which subset of replicas to contact for a storage operation profoundly impacts the behavior of distributed storage systems. For strong consistency a client may read and write a majority of replicas. Since majorities overlap, this ensures that each read “sees” the latest write. In contrast, eventually consistent systems may read and write non-overlapping subsets.^{26,28} Once a write is acknowledged, the new value is propagated to the remaining replicas; thus, all replicas are eventually updated unless a failure occurs. In the meantime, readers may observe stale values if they fetch data from replicas that have not yet received the update.

Although many applications benefit from strong consistency, latency-sensitive applications such as shopping carts in e-commerce Web sites choose eventual consistency to gain lower latency.¹ This can lead to consistency anomalies such as items lost from a shopping cart or oversold. However, since a user can detect and correct the problem during checkout, such anomalies are tolerable provided they are short-lived and infrequent. The critical task for application developers and system

administrators, therefore, is to understand how consistency and latency are impacted by various storage and application configuration parameters, as well as the workload that the application places on the storage system. Only with proper insight into this subtle issue can administrators and developers make sensible decisions regarding the configuration of the storage system or the choice of consistency model for a given application.

This article looks at methods of quantifying consistency (or lack thereof) in eventually consistent storage systems. These methods are necessary for meaningful comparisons among different system configurations and workloads. First, the article defines eventual consistency more precisely and relates it to other notions of weak consistency. It then drills down into metrics, focusing on staleness, and surveys different techniques for predicting and measuring staleness. Finally, the relative merits of these techniques are evaluated, and any remaining open questions are identified.

DEFINING EVENTUAL CONSISTENCY

Eventual consistency can be defined as either a property of the underlying storage system, or a behavior observed by a client application. For example, Doug Terry et al. give the following definition of eventual consistency in the context of the Bayou storage system: “All replicas eventually receive all writes (assuming sufficient network connectivity and reasonable reconciliation schedules), and any two replicas that have received the same set of writes have identical databases.”²⁶ This informal definition addresses both the propagation of writes to replicas (i.e., the *eventual*) and convergence onto a well-defined state—for example, through timestamp-based reconciliation—(i.e., the *consistency*).

This definition, while simple and elegant, does not say precisely what clients observe. Instead, it captures a range of behaviors encompassing both strongly consistent relational databases, and weakly consistent systems that may return stale or out-of-order data without bound. In contrast, Werner Vogels describes the consistency observed by clients in a concrete way: “The storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value.”²⁸ Vogels’s definition captures in an intuitive way the convergence of replicas to the last updated value, but only in the special case where updates are suspended while clients read an object—a scenario very different from the online environment in which many eventually consistent systems are deployed.

Since weak consistency is difficult to reason about, application developers often seek stronger properties such as a consistent prefix, monotonic reads, “read my writes,” or causal consistency.²⁵ Indeed, some systems support such stronger-than-eventual properties (e.g., COPS [Clusters of Order-Preserving Servers²²] and Pileus²⁷), but the commercial success of systems such as Amazon’s Dynamo¹² proves that eventual consistency can be good enough in practice. To understand why, researchers have sought to describe the range of behaviors of eventually consistent systems in more precise terms than abstract definitions in the style of Terry et al. and Vogels. Existing approaches in this endeavor fall into three categories: relaxed consistency metrics, system modeling and prediction, and empirical measurement tools. The next three sections survey representative works in each category.

RELAXED CONSISTENCY PROPERTIES

Abstract definitions of eventual consistency leave open a number of questions regarding system behavior in the presence of concurrent accesses, as well as in a failure-prone environment. For

example: How quickly do updates propagate to replicas in practice, and how often do replicas agree on the latest value of an object? When replicas do not agree, how does that affect the clients' view of the data? What exactly do clients observe during an extended failure that partitions the network, or in the moments shortly after the network is healed?

An emerging approach for describing these behaviors is to relate them to a strict form of consistency called *linearizability*.¹⁸ Informally speaking, this property states that the storage system behaves as though it executes operations one at a time, in some serial order, despite actually executing some operations in parallel. As will be explained later, this serial order is further constrained in a manner that forbids stale reads. As a result, linearizability is a suitable gold standard against which eventually consistent systems can be judged. In other words, the observed behavior can be described in terms of deviations from this standard, which in this context can be regarded as consistency violations. For example, one can think of Amazon's Dynamo as violating linearizability when reads return stale values, even though the system promises only eventual consistency.

Whereas linearizability is ubiquitous in centralized systems—for example, in multithreaded data structures—Brewer's CAP principle⁹ states that such a strong consistency property (C) is unattainable in always-available (A) and partition-tolerant (P) distributed systems. (Database experts should read *consistency* in this context as *transaction isolation*.) As a result, one cannot expect any storage system to accept updates and yet remain consistent during a network partition. Although this does not preclude strong consistency during failure-free operation, even in that case the system may be configured to sacrifice consistency for better latency.¹ For example, in the Cassandra key-value store, clients can achieve this trade-off by requesting various “consistency levels,” which control the number of replicas that respond to a read or write.²⁰

The side effect of weakening consistency is increased staleness, which so far has been discussed informally. More precisely, a value is considered *fresh* from the moment it is written into a storage system until it is overwritten, and *stale* thereafter. Thus, staleness describes the age of a value returned by a read relative to the last updated value, and hence quantifies how badly a system's behavior deviates from the gold standard. Two interpretations of this concept have been discussed in the literature,^{2,15} arising from different ways to define “age”:

- *Version-based staleness* defines age by counting versions of an object (e.g., a read returns the *k*th-latest version).
- *Time-based staleness* defines age in terms of wall-clock time (e.g., a read returns a value *t* time units older than the last updated value).

The concept of staleness, although intuitive, is fraught with technical subtleties. In the simple case where operations are executed one at a time, there is a natural order in which clients expect these operations to take effect, and so stale reads can be identified easily. When operations are executed in parallel, however, their order can be very difficult to determine from the client's perspective.

To make sense of eventual consistency, we turn to relaxed consistency properties—*k*-*atomicity* and *Δ*-*atomicity*—which give precise meaning to the notions of version-based and time-based staleness, respectively. Both of these properties are relaxed forms of linearizability,¹⁸ but for historical reasons they refer in name to Lamport's *atomicity* property,²¹ which is similar in spirit.

LINEARIZABILITY (THE “GOLD STANDARD”)

Consider the trace of operations in figure 1a, showing three operations applied to an object denoted

FIGURE 1 Example Trace of Operations Illustrating Linearizability, K-Atomicity, and Δ -Atomicity

by X . First, a write operation $W(X,1)$ assigns 1 to object X , and then this value is updated to 2 by a second write operation, $W(X,2)$. The third operation $R(X)$ is a read of X and begins after both writes have ended. In this case, linearizability dictates that $W(X,1)$ should appear to take effect before $W(X,2)$, because $W(X,1)$ ends before $W(X,2)$ begins. Thus, 2 is the last updated value of X when $R(X)$ is applied, but $R(X)$ returns 1 instead, so the trace is not linearizable. The more complex case when operations overlap in time is addressed by the formal definition of linearizability,¹⁸ a discussion that is beyond the scope of this article.

K-ATOMICITY (VERSION-BASED STALENESS)

The k -atomicity property was introduced by Amitanand Aiyer et al.² Like linearizability, it requires that operations appear to take effect in an order that is constrained by their start and finish times; within this order, however, a read may return *any* of the k last updated values. For example, the trace shown in figure 1a is k -atomic for $k = 2$ but not $k = 1$.

Δ -ATOMICITY (TIME-BASED STALENESS)

The Δ -atomicity property was proposed by Wojciech Golab et al.¹⁵ Similar to k -atomicity, it relaxes linearizability by allowing reads to return stale values. Staleness, however, is defined in terms of time: a read may return a value that is up to Δ time units stale. For example, the trace in figure 1a is Δ -atomic if Δ is defined as the width of the gap between $W(X,2)$ and $R(X)$. Linearizability permits $R(X)$ to take effect before $W(X,2)$ if the two operations overlap in time, as shown in figure 1b, whereas it requires $R(X)$ to take effect after $W(X,2)$ in figure 1a. If $R(X)$ is hypothetically “stretched” to the left by Δ time units (i.e., if $R(X)$ had started Δ time units earlier), as in figure 1b, then the trace becomes linearizable. Thus, the response of $R(X)$ is considered only Δ time units stale in figure 1a.

PREDICTION

Another method used to characterize eventual consistency is based on a combination of system modeling and prediction. Peter Bailis et al.⁶ present the PBS (Probabilistically Bounded Staleness) framework, in which a white-box system model is proposed to predict data staleness observed by client applications. The PBS model estimates the probability of $\langle k, t \rangle$ staleness—the condition that the read, which begins t time units after the end of the write, returns the value assigned by one of the last k writes. This condition is similar to k -atomicity but considers only a single read at a fixed distance t from the write. When $k = 1$, it captures the probability that the read returns the latest value (i.e., is not stale).

The PBS model makes two simplifying assumptions. First, like Vogels's definition of eventual consistency,²⁸ PBS does not consider workloads where writes overlap in time with reads, in which the width t of the gap between writes and reads is not well defined. Second, PBS does not model failures explicitly. Although storage node failures can be simulated using longer latencies, PBS does not account for network partitions. Follow-up work¹¹ extends the PBS model by considering node failures.

EMPIRICAL MEASUREMENT

Thus far, this article has addressed techniques for quantifying staleness in eventually consistent systems. This section discusses approaches that measure consistency empirically from the perspective of both the system and the client.

Measuring eventual consistency “in the wild” is as hard as defining it precisely. Concurrent operations make it difficult to identify the order in which operations take effect, and hence to classify reads as stale or not. To make matters worse, in the event of a network partition, clients on opposite sides of the divide may observe the last updated value differently, even if no new updates are made to an object after some point in time. Such an anomaly is possible because in an always-available system each partition will continue to accept writes.

Despite these challenges, a number of techniques have been devised for measuring eventual consistency, particularly data staleness. This article looks at two fundamentally different methodologies: *active measurement*, in which the storage system is exercised in an artificial way to determine the time lag from when a new value is written to the storage system until this value becomes visible to clients; and *passive analysis*, in which a trace of operations is recorded for an arbitrary workload and analyzed mathematically to obtain a measurement of staleness.

ACTIVE MEASUREMENT

Active measurement underlies early studies of consistency in cloud storage systems. In this category of techniques, one client writes a new value to a key, and a different client then reads the same key repeatedly until the new value is returned. As in Vogels's definition of eventual consistency,²⁸ writes do not overlap in time with reads in this scenario. The time from the write to the last read that returns the old value—or alternatively, the first read of the new value—answers the question, “How eventual?” and can be regarded as an estimate of the *convergence time* of the replication protocol—the time needed to propagate a new value to all the replicas of an object.

At a technical level, the main challenge in active measurement is to determine the difference in time between operations executed at different client nodes—namely, a write and a read. Figure 2

FIGURE 2 Infrastructure for Consistency Measurement

illustrates how this difference can be computed using a collection of clients. To capture precisely the moment when the last replica receives the updated value, Hiroshi Wada et al. apply reads 50 times per second using one or more clients.²⁹ Bermbach et al. go one step further and use a collection of geographically distributed readers to ensure that the reads hit all possible replicas.⁸ Staleness measurement in YCSB++ (Yahoo! Cloud-serving Benchmark)²³ follows a similar approach but uses a ZooKeeper producer-consumer queue¹⁹ to synchronize writers and readers. This approach circumvents issues related to clock skew but introduces additional latencies caused by queue operations, which may limit precision.

Active measurement can be used to discover the range of update propagation times in an eventually consistent system. For example, in experiments involving Amazon's SimpleDB,⁴ Wada et al. report that convergence occurred in at most one second in more than 90 percent of the runs, but took more than four seconds in a few (less than 1 percent) cases. On the other hand, active measurement does not indicate what proportion of reads in a real workload will return stale values, as this quantity depends on how the data objects are accessed. For example, the proportion of stale reads would be expected to vary with the rate at which operations are applied on a given object—

close to zero when operations are applied infrequently and the gaps between them exceed the convergence time, and larger when reads follow writes more closely. Active measurement does not separate these two cases, as it is based upon a controlled workload designed to measure convergence time.

PASSIVE ANALYSIS

In the earlier section on relaxed consistency properties, the discussion focused on ways of defining staleness in a precise and meaningful way and provided a hint of how a consistency metric can be derived from such definitions. It begins with a “black-or-white” consistency property, such as linearizability, which is either satisfied or not satisfied by a system or an execution trace; next, this property is relaxed by introducing a parameter that bounds the staleness of reads (e.g., Δ -atomicity); the last step is to determine the parameter value that most accurately describes the system’s behavior (e.g., Δ -atomic for $\Delta \geq 10$ ms). Passive analysis refers to this final step and entails examining the operations recorded in an execution trace to determine the order in which they appear to take effect.

The most immediate technical challenge in passive analysis is the collection of the trace, which records for each operation its type, start and finish times, as well as its arguments and response (e.g., a read of object X , starting at time t_1 and ending at t_2 , returning the value 5). This trace can be obtained at clients, as shown in figure 2.

Because the trace is obtained by merging data from multiple clients or storage nodes over a finite period of time, it is prone to two types of anomalies: *dangling reads*, which return values that lack a corresponding write (i.e., a write that assigns the value returned by the read); and *reversed operations*, whereby a read appears in the trace before its corresponding write. These anomalies must be removed if they occur; otherwise, the k -atomicity and Δ -atomicity properties are undefined and cannot be used for computing staleness.

Dangling reads indicate missing information, such as when the first access to an object in a trace is a read and the corresponding write occurs before the start of the trace. In this case the dangling read is identified easily and can be dropped from the trace with no ill effects. Reversed operations are equally easy to detect, but harder to remedy. Clock synchronization techniques such as atomic clocks and GPS¹⁰ can eliminate reversed operations altogether. Even ordinary NTP (Network Time Protocol) is sufficient if the synchronization of clocks is tight enough. Alternatively, one can also estimate clock skew directly from reversed operations and adjust the trace accordingly.

Given a trace free from dangling reads and reversed operations, the next challenge is to compute staleness metrics. Efficient algorithms are critical in this context because the trace may be very long, which means that both the space and computation required are high. In fact, the problem of testing whether a trace is linearizable was shown by Gibbons and Korach to be intractable.¹³ Testing whether a trace is k -atomic or Δ -atomic for fixed k and Δ is at least as hard since these properties are equivalent to linearizability when $k = 1$ and $\Delta = 0$. Computing k and Δ from a trace is harder still, and hence also intractable.

Fortunately, the intractability result breaks in the special case when every write on a given object assigns a unique value. This condition is straightforward to enforce (e.g., by embedding a unique token in each value written, such as a node ID and timestamp), which opens the door to efficient algorithms for computing staleness metrics from traces in practice. For Δ -atomicity, such an algorithm is known,¹⁵ and has been used to analyze time-based staleness in traces obtained using

Cassandra.²⁴ For k -atomicity, an efficient algorithm is known only for deciding whether a trace is 2-atomic.¹⁴

Passive analysis in general is not limited to staleness metrics, as illustrated by the technique of *cycle analysis*,^{5,30} which detects linearizability violations by analyzing a *conflict graph* representing the execution trace. The absence of cycles in such a graph indicates that the trace is linearizable, and so the number and length of such cycles can be defined as metrics for inconsistency.^{5,30} The relationship of these metrics to staleness is not known precisely.

COMPARISON AND DISCUSSION

Prediction, active measurement, and passive analysis are all useful ways to study eventual consistency. Although each method has specific strengths and weaknesses, it is difficult to compare them directly, as they meet different goals. Passive analysis is *client-centric* in that it reflects the manner in which the client application interacts with the system. As a result, passive analysis can be used to compare staleness observed in two workloads applied to the same storage system. Active measurement, on the other hand, is *system-centric* in that it measures the convergence time of a system's replication protocol for a controlled workload. Thus, active measurement is best suited for comparing different systems in terms of staleness, or the same system under different software or hardware configurations. The PBS framework⁶ is also designed around a controlled workload, and for that reason it is classified as system-centric.

The techniques discussed in this article also vary in terms of conceptual models of eventual consistency. Active measurement and PBS are based upon a *simplified model*, similar to Vogels's definition,²⁸ in which writes occur before reads. In contrast, passive analysis considers a *general model* in which writes may overlap in time with reads and with other writes. As explained in another article by Golab et al.,¹⁶ storage systems may behave differently in these two models when clients follow the " $R + W > N$ " rule,²⁸ which ensures that read and write operations access overlapping subsets of replicas. The latter condition has long been considered sufficient for "strong consistency,"^{6,28} and in fact guarantees linearizability in the simplified model but permits linearizability violations in the general model.

Despite the somewhat high cost of analyzing a detailed trace of operations, passive analysis plays an important role in understanding eventual consistency. Whereas any perspective on consistency is ultimately affected by the state of data in replicas in a storage system, which can be thought of as the "ground truth," passive analysis is most closely tied to the actual consistency observed by client applications. Specifically, it reflects data-level anomalies only when they manifest themselves to clients—for example, as stale reads.

FUTURE WORK

For modern online service providers, small decreases in latency are known to have a measurable effect on user experience and, as a consequence, revenue.¹⁷ As weak consistency continues to gain traction as the means for reducing latency, the ability to reason clearly about this trade-off will be an important competitive advantage for service providers.

Consistency-aware pricing schemes and guarantees are already being adopted by industry. Amazon's DynamoDB supports differentiated pricing for consistent reads and eventually consistent reads, with the former guarantee costing the application developer twice as much.³ More recently,

Terry et al. describe a system that allows client applications to select different grades of weak consistency, such as “at most 200-ms latency and at most 5-minute staleness,” through novel SLAs (service-level agreements).²⁷ Other read guarantees, such as monotonic reads or causal consistency, can be requested, and a “wish list” of different combinations can be specified with different utilities. This opens the door to even more fine-grained pricing schemes, making it more important than ever for users to understand the utility of different points in consistency-related trade-offs.

Whereas the understanding of eventual consistency is expected to improve with additional empirical studies involving measurement, fundamental technical problems also remain to be solved. In particular, the problem of consistency *verification* remains an open and important challenge. Storage system designers need verification techniques to test whether their implementations correctly fulfill application-specified SLAs, and, likewise, application developers could use such tools to verify the service quality actually received.

One of the open problems related to verification algorithms is to determine the computational complexity of deciding k -atomicity. Existing algorithms handle only the special case $k < 3$,¹⁴ which limits their use in practice. Similar technical ideas can be applied when $k \geq 3$, but only for a restricted class of traces, and it is not known whether an efficient algorithm exists for deciding k -atomicity in the general case.

CONCLUSION

Eventual consistency is increasingly viewed as a spectrum of behaviors that can be quantified along various dimensions, rather than a binary property that a storage system either satisfies or fails to satisfy. Advances in characterizing and verifying these behaviors will enable service providers to offer an increasingly rich set of service levels of differentiated performance, ultimately improving the end user’s experience.

ACKNOWLEDGMENTS

We are grateful to Jay J. Wylie, Indranil Gupta, and Doug Terry for their helpful feedback.

REFERENCES

1. Abadi, D. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer* 45(2): 37–42.
2. Aiyer, A., Alvisi, L., Bazzi, R. A. 2005. On the availability of non-strict quorum systems. In *Proceedings of the 19th International Symposium on Distributed Computing*: pp. 48–62.
3. Amazon Web Services. DynamoDB; <http://aws.amazon.com/dynamodb/>.
4. Amazon Web Services. SimpleDB; <http://aws.amazon.com/simplydb/>.
5. Anderson, E., Li, X., Shah, M. A., Tucek, J., Wylie, J. J. 2010. What consistency does your key-value store actually provide? In *Proceedings of the 6th Usenix Workshop on Hot Topics in System Dependability*.
6. Bailis, P., Venkataraman, S., Franklin, M. J., Hellerstein, J. M., Stoica, I. 2012. Probabilistically bounded staleness for practical partial quorums. In *Proceedings of the VLDB (Very Large Data Base) Endowment* 5(8): 776–787.
7. Bermbach, D., Zhao, L., Sakr, S. 2013. Towards comprehensive measurement of consistency guarantees for cloud-hosted data storage services. In *Proceedings of the 5th TPC (Transaction*

- Processing Performance Council) Technology Conference on Performance Evaluation and Benchmarking.
8. Bermbach, D., Tai, S. 2011. Eventual consistency: how soon is eventual? An evaluation of Amazon S3's consistency behavior. In *Proceedings of the 6th Workshop on Middleware for Service-oriented Computing*.
 9. Brewer, E. A. 2000. Towards robust distributed systems (invited talk). In *Proceedings of the 19th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*.
 10. Corbett, J. C., et al. 2012. Spanner: Google's globally distributed database. In *Proceedings of the 10th Usenix Conference on Operating Systems Design and Implementation*: 251–264.
 11. Davidson, A., Rubinstein, A., Todt, A., Bailis, P., Venkataraman, S. 2012. Adaptive hybrid quorums in practical settings; http://www.cs.berkeley.edu/~kubitron/courses/cs262a-F12/projects/reports/project12_report_ver2.pdf.
 12. Decandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Voshall, P., Vogels, W. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*.
 13. Gibbons, P. B., Korach, E. 1997. Testing shared memories. *SIAM (Society for Industrial and Applied Mathematics) Journal on Computing* 26(4): 1208–1244.
 14. Golab, W., Hurwitz, J., Li, X. 2013. On the k -atomicity-verification problem. In *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems*.
 15. Golab, W., Li, X., Shah, M. A. 2011. Analyzing consistency properties for fun and profit. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*: 197–206.
 16. Golab, W., Rahman, M. R., AuYoung, A., Keeton, K., Gupta, I. 2013. Client-centric benchmarking of eventual consistency for cloud storage systems. Manuscript under submission.
 17. Hamilton, J. 2009. The cost of latency; <http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx>.
 18. Herlihy, M., Wing, J. M. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12(3): 463–492.
 19. Hunt, P., Konar, M., Junqueira, F. P., Reed, B. 2010. ZooKeeper: wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 Usenix Annual Technical Conference*: 11–25.
 20. Lakshman, A., Malik, P. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44(2): 35–40.
 21. Lamport, L. 1986. On interprocess communication. Part I: basic formalism; and Part II: algorithms. *Distributed Computing* 1(2): 77–101.
 22. Lloyd, W., Freedman, M. J., Kaminsky, M., Andersen, D. G. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*: 401–416.
 23. Patil, S., Polte, M., Ren, K., Tantisiroj, W., Xiao, L., López, J., Gibson, G., Fuchs, A., Rinaldi, B. 2011. YCSB++: benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*: 9:1–9:14.
 24. Rahman, M. R., Golab, W., AuYoung, A., Keeton, K., Wylie, J. J. 2012. Toward a principled framework for benchmarking consistency. In *Proceedings of the 8th Usenix Workshop on Hot Topics in System Dependability*.
 25. Terry, D. 2013. Replicated data consistency explained through baseball. *Communications of the ACM* 56(12): 82–89.

26. Terry, D. B., Petersen, K., Spreitzer, M. J., Theimer, M. M. 1998. The case for non-transparent replication: examples from Bayou. *IEEE Data Engineering Bulletin* 21(4): 12–20.
27. Terry, D. B., Prabhakaran, V., Kotla, R., Balakrishnan, M., Aguilera, M. K., Abu-Libdeh, H. 2013. Consistency-based service-level agreements for cloud storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*.
28. Vogels, W. 2008. Eventually consistent. *ACM Queue* 6(6): 14–19.
29. Wada, H., Fekete, A., Zhao, L., Lee, K., Liu, A. 2011. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*: 134–143.
30. Zellag, K., Kemme, B. 2012. How consistent is your cloud application? In *Proceedings of the 3rd ACM Symposium on Cloud Computing*.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

WOJCIECH GOLAB is an assistant professor in electrical and computer engineering at the University of Waterloo. His current research focuses on algorithmic problems in distributed computing with applications to storage, transaction processing, and big-data analytics. Prior to joining Waterloo, he worked as a researcher in storage systems at Hewlett-Packard Labs in Palo Alto and as a postdoctoral fellow in theory at the University of Calgary. He received a Ph.D in computer science from the University of Toronto in 2010.

MUNTASIR RAIHAN RAHMAN is a computer science Ph.D student in the Distributed Protocols Research Group at the University of Illinois at Urbana-Champaign. His current research is focused on consistency in distributed systems. He received a B.Sc. degree in computer science and engineering from Bangladesh University of Engineering and Technology in 2006, and the M. Math degree in computer science from the University of Waterloo in 2010.

ALVIN AU YOUNG is a researcher in the systems group at Hewlett-Packard Labs. His current work focuses on designing in-memory distributed systems for storage and large-scale data analysis. In the past he has also investigated problems at the intersection of economics and computer science, specifically applying ideas from markets and auctions to distributed systems. He received a Ph.D. in computer science from the University of California at San Diego.

KIMBERLY KEETON is a principal researcher at Hewlett-Packard Laboratories. Her recent research is in the areas of NoSQL databases, consistency models, and information management. She has also worked in the areas of storage management, storage dependability, workload characterization, and intelligent storage. She earned a Ph.D. in computer science from the University of California, Berkeley, and is an ACM Distinguished Scientist and a Senior Member of the IEEE.

XIAOZHOU (STEVE) LI is a software engineer at Google. In the past, he worked at Hewlett-Packard Labs as a researcher and at Microsoft as a software design engineer. His main interests are in the theory and practice of distributed computing. He received a Ph.D. in computer science from the University of Texas at Austin.

© 2014 ACM 1542-7730/14/0100 \$10.00