



How can applications be built on eventually consistent infrastructure given no guarantee of safety?

BY PETER BAILIS AND ALI GHODSI

Eventual Consistency Today: Limitations, Extensions, and Beyond

IN A JULY 2000 conference keynote, Eric Brewer, now VP of engineering at Google and a professor at the University of California, Berkeley, publicly postulated the CAP (consistency, availability, and partition tolerance) theorem, which would change the landscape of how distributed storage systems were architected.⁸

Brewer's conjecture—based on his experiences building infrastructure for some of the first Internet search engines at Inktomi—states that distributed systems requiring always-on, highly available operation cannot guarantee the illusion of coherent, consistent single-system operation in the presence of network partitions, which cut communication between active servers. Brewer's conjecture proved prescient: in the following decade, with the continued rise of large-scale Internet services, distributed-system architects frequently dropped “strong” guarantees in favor of weaker models—the most notable being *eventual consistency*.

Eventual consistency provides few guarantees. Informally, it guarantees

that, if no *additional* updates are made to a given data item, all reads to that item will eventually return the same value. This is a particularly weak model. At no given time can the user rule out the possibility of inconsistent behavior: the system can return *any* data and still be eventually consistent—as it might “converge” at some later point. The only guarantee is that, at some point in the future, something good will happen. Yet, given this apparent lack of useful guarantees, scores of usable applications and profitable businesses are built on top of eventually consistent infrastructure. How?

This article begins to answer this question by describing several notable developments in the theory and practice of eventual consistency, with

a focus on immediately applicable takeaways for practitioners running distributed systems in the wild. As production deployments have increasingly adopted weak consistency models such as eventual consistency, we have learned several lessons about how to reason about, program, and strengthen these weak models.

In summary, we will primarily focus on three questions and some preliminary answers:

How eventual is eventual consistency? If the scores of system architects advocating eventual consistency are any indication, eventual consistency seems to work “well enough” in practice. How is this possible when it provides such weak guarantees? *New prediction and measurement techniques allow system architects to quantify the behavior of real-world eventually consistent systems. When verified via measurement, these systems appear strongly consistent most of the time.*

How should one program under eventual consistency? How can system architects cope with the lack of guarantees provided by eventual consistency? How do they program without strong ordering guarantees? *New research enables system architects to deal with inconsistencies, either via external compensation outside of the system or by limiting themselves to data structures that avoid inconsistencies altogether.*

Is it possible to provide stronger guarantees than eventual consistency without losing its benefits? In addition to guaranteeing eventual consistency and high availability, what other guar-

antees can be provided? *Recent results show that it is possible to achieve the benefits of eventual consistency while providing substantially stronger guarantees, including causality and several ACID (atomicity, consistency, isolation, durability) properties from traditional database systems while still remaining highly available.*

This article is *not* intended as a formal survey of the literature surrounding eventual consistency. Rather, it is a pragmatic introduction to several developments on the cutting edge of our understanding of eventually consistent systems. The goal is to provide the necessary background for understanding both *how* and *why* eventually consistent systems are programmed, deployed, and have evolved, as well as where the systems of tomorrow are heading.

History and Concepts of Eventual Consistency

Brewer’s CAP theorem dictates it is impossible simultaneously to achieve always-on experience (*availability*) and to ensure users read the latest written version of a distributed database (*consistency*—as formally proven, a property known as “linearizability”¹¹) in the presence of partial failure (*partitions*).⁸ CAP pithily summarizes trade-offs inherent in decades of distributed-system designs (for example, RFC 677¹⁴ from 1975) and shows that maintaining an SSI (single-system image) in a distributed system has a cost.¹⁰ If two processes (or groups of processes) within a distributed system cannot communicate (are *parti-*

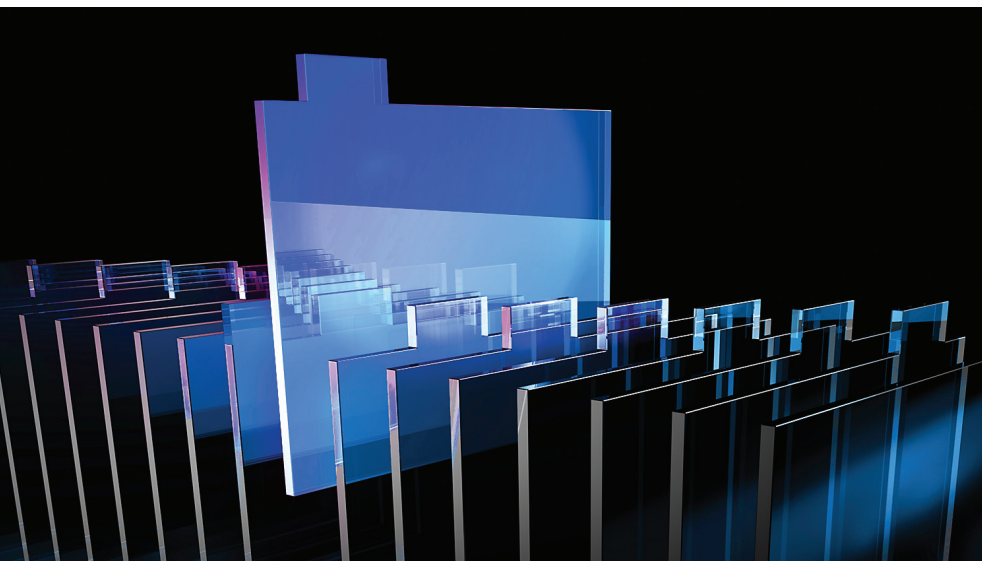
tioned)—either because of a network failure or the failure of one of the components—then updates cannot be synchronously propagated to all processes without blocking. Under partitions, an SSI system cannot safely complete updates and hence presents unavailability to some or all of its users. Moreover, even without partitions, a system that chooses availability over consistency enjoys benefits of low latency: if a server can safely respond to a user’s request when it is partitioned from all other servers, then it can *also* respond to a user’s request without contacting other servers even when it is able to do so.¹ (Note that you cannot “sacrifice” partition tolerance!¹² The choice is between consistency and availability.)

As services are increasingly replicated to provide fault tolerance (ensuring services remain online despite individual server failures) and capacity (to allow systems to scale with variable request rates), architects must face these consistency-availability and -latency trade-offs head on. In a dynamic, partitionable Internet, services requiring guaranteed low latency must often relax their expectations of data consistency.

Eventual consistency as an available alternative. Given the CAP impossibility result, distributed-database designers sought weaker consistency models that would enable both availability and high performance. While weak consistency has been studied and deployed in various forms since the 1970s,¹⁹ the eventual consistency model has become prominent, particularly among emerging, highly scalable NoSQL stores.

One of the earliest definitions of eventual consistency comes from a 1988 paper describing a group communication system¹⁵ not unlike a shared text editor such as Google Docs today: “...changes made to one copy eventually migrate to all. If all update activity stops, after a period of time all replicas of the database will converge to be logically equivalent: each copy of the database will contain, in a predictable order, the same documents; replicas of each document will contain the same fields.”

Under eventual consistency, all servers eventually “converge” to the same state; at some point in the future, servers are indistinguishable from one an-




other. This eventual convergence, however, does not provide SSI semantics. First, the “predictable order” will not necessarily correspond to an execution that could have arisen under SSI; eventual consistency does not specify which value is eventually chosen. Second, there is an unspecified window before convergence is reached, during which the system will not provide SSI semantics, but rather arbitrary values. As we will illustrate, this promise of eventual convergence is a rather weak property. Finally, a system with SSI provides eventual consistency—the “eventuality” is immediate—but not vice versa.


Why is eventual consistency useful? Pretend you are in charge of the data infrastructure at a social network where users post new status updates that are sent to their followers’ timelines, represented by separate lists—one per user. Because of large scale and frequent server failures, the database of timelines is stored across multiple physical servers. In the event of a partition between two servers, however, you cannot deliver each update to all timelines. What should you do? Should you tell the user he or she cannot post an update, or should you wait until the partition heals before providing a response? Both of these strategies choose consistency over availability, at the cost of user experience.

Instead, what if you propagate the update to the reachable set of followers’ timelines, return to the user, and delay delivering the update to the other followers until the partition heals? In choosing this option, you give up the guarantee that all users see the same set of updates at every point in time (and admit the possibility of timeline reordering as partitions heal), but you gain high availability and (arguably) a better user experience. Moreover, because updates are eventually delivered, all users eventually see the same timeline with all of the updates that users posted.

Implementing eventual consistency. A key benefit of eventual consistency is that it is fairly straightforward to implement. To ensure convergence, replicas must exchange information with one another about which writes they have seen. This information exchange is often called *anti-entropy*, a homage to the process of reversing entropy, or thermodynamic randomness, in a



Under eventual consistency, all servers eventually “converge” to the same state; at some point in the future, servers are indistinguishable from one another.



physical system.¹⁹ Protocols for achieving anti-entropy take a variety of forms; one simple solution is to use an asynchronous all-to-all broadcast: when a replica receives a write to a data item, it immediately responds to the user, then, in the background, sends the write to all other replicas, which in turn update their locally stored data items. In the event of concurrent writes to a given data item, replicas deterministically choose a “winning” value, often using a simple rule such as “last writer wins” (for example, via a clock value embedded in each write).²²

Suppose you want to make a single-node database into an eventually consistent distributed database. When you get a request, you route it to any server you can contact. When a server performs a write to its local key-value store, it can send the write to all other servers in the cluster. This write-forwarding becomes the anti-entropy process. Be careful, however, when sending the write to the other servers. If you wait for other servers to respond before acknowledging the local write, then, if another server is down or partitioned from you, the write request will hang indefinitely. Instead, you should send the request in the background; anti-entropy should be an asynchronous process. Implicitly, the model for eventual consistency assumes system partitions are eventually healed and updates are eventually propagated, or that partitioned nodes eventually die and the system ends up operating in a single partition.

The eventually consistent system has some great properties. It does not require writing difficult “corner-case” code to deal with complicated scenarios such as downed replicas or network partitions—anti-entropy will simply stall—or writing complex code for coordination such as master election. All operations complete locally, meaning latency will be bounded. In a geo-replicated scenario, with replicas located in different data centers, you do not have to endure long-haul wide-area network latencies on the order of hundreds of milliseconds on the request fast path. The mechanism just described, returning immediately on the local write, can put data durability at risk. An intermediate point in trading between durability and availability is to return after W


replicas have acknowledged the write, thus allowing the write to survive W-1 replica failures. Anti-entropy can be run as often or as rarely as desired without violating any guarantees. What's not to like?

Safety and liveness. While eventual consistency is easy to achieve, the current definition leaves some unfortunate holes. First, what is the eventual state of the database? A database always returning the value 42 is eventually consistent, even if 42 were never written. Amazon CTO Werner Vogels' preferred definition specifies that "eventually all accesses return the last updated value;" accordingly, the database cannot converge to an arbitrary value.²³ Even this new definition has another problem: what values can be returned before the eventual state of the database is reached? If replicas have not yet converged, what guarantees can be made on the data returned?


These questions stem from two kinds of properties possessed by all distributed systems: safety and liveness.² A *safety* property guarantees that "nothing bad happens;" for example, every value that is read was, at some point in time, written to the database. A *liveness* property guarantees that "something good eventually happens;" for example, all requests eventually receive a response.

The difficulty with eventual consistency is that it makes no safety guarantees—eventual consistency is purely a liveness property. Something good eventually happens—the replicas agree—but there are no guarantees with respect to what happens, and no behavior is ruled out in the meantime! For meaningful guarantees, safety and liveness properties need to be taken together: without one or the other, you can have trivial implementations that provide less-than-satisfactory results.

Virtually every other model that is stronger than eventual provides some form of safety guarantees. For almost all production systems, however, eventual consistency should be considered a bare-minimum requirement for data consistency. A system that does not guarantee replica convergence is remarkably difficult to reason about.



The difficulty with eventual consistency is that it makes no safety guarantees—eventual consistency is purely a liveness property.



How Eventual is Eventual Consistency?

Despite the lack of safety guarantees, eventually consistent data stores are widely deployed. Why? While eventually consistent stores do not promise safety, there is evidence that eventual consistency works well in practice. Eventual consistency is "good enough," given its latency and availability benefits. For the many stores that offer a choice between eventual consistency and stronger consistency models, scores of practitioners advocate eventual consistency.

The behavior of eventually consistent stores can be quantified. Just because eventual consistency does not promise safety does not mean safety is not often provided—and you can both measure and predict these properties of eventually consistent systems using a range of techniques that have been recently developed and are making their way to production stores. These techniques—which we discuss next—have surprisingly shown that eventual consistency often behaves like strong consistency in production stores.

Metrics and mechanisms. One common metric for eventual consistency is *time*: how long will it take for writes to become visible to readers? This captures the "window of consistency" measured according to the wall clock. Another metric is *versions*: how many versions old will a given read be? This information can be used to ensure readers never go back in time and observe progressively newer versions of the database. While time and versions are perhaps the most intuitive metrics, there are a range of others, such as numerical drift from the "true" value of each data item and combinations of each of these metrics.²⁵

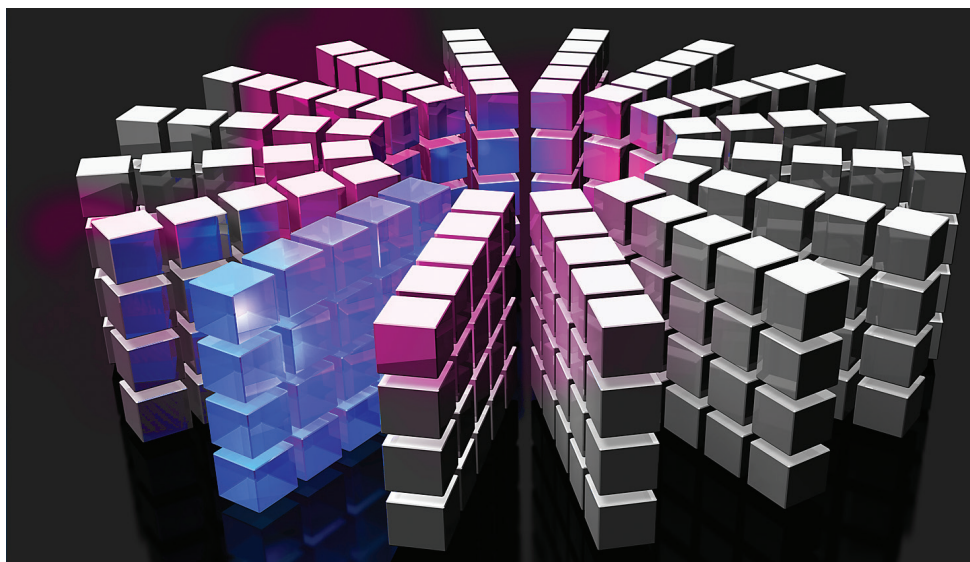
The two main kinds of mechanisms for quantifying for eventual consistency are measurement and prediction. *Measurement* answers the question, "How consistent is my store under my given workload right now?"¹⁸ while *prediction* answers the question, "How consistent will my store be under a given configuration and workload?"⁴ Measurement is useful for runtime monitoring and alerts or verifying compliance with service-level objectives (SLOs). Prediction is useful for probabilistic what-if analyses such as the effect of configuration

and workload changes and for dynamically tuning system behavior. Taken together, measurement and prediction form a useful toolkit.

Probabilistically bounded staleness. As a brief deep dive into how to quantify eventually consistent behavior, we will discuss our experiences developing, deploying, and integrating state-of-the-art prediction techniques into Cassandra, a popular NoSQL. Probabilistically Bounded Staleness, or PBS, provides an *expectation* of recency for reads of data items.⁴ This allows us to measure how far an eventually consistent store's behavior deviates from that of a strongly consistent, linearizable (or regular) store. PBS enables metrics of the form: "100 milliseconds after a write completes, 99.9% of reads will return the most recent version," and "85% of reads will return a version that is within two of the most recent."

Building PBS. How does PBS work? Intuitively, the degree of inconsistency is determined by the rate of anti-entropy. If replicas constantly exchange their last written writes, then the window of inconsistency should be bounded by the network delay and local processing delay at each node. If replicas delay anti-entropy (possibly to save bandwidth or processing time), then this delay is added to the window of inconsistency; many systems (Amazon's Dynamo, for example) offer settings in the replication protocol to control these delays. Given the anti-entropy protocol, then—given the configured anti-entropy rate, the network delay, and local processing delay—you can calculate the expected consistency. In Cassandra, we piggyback timing information on top of the write distribution protocol (the primary source of anti-entropy) and maintain a running sample. When a user wants to know the effect of a given replication configuration, we use the collected sample in a Monte Carlo simulation of the protocol to return an expected value for the consistency of the data store, which closely matches consistency measurements on our Cassandra clusters at Berkeley.

PBS in the wild. Using our PBS consistency prediction tool, and with the help of several friends at LinkedIn and Yammer, we quantified the consistency of three eventually consistent stores running in production. PBS models



predicted that LinkedIn's data stores returned consistent data 99.9% of the time within 13.6ms; and on SSDs, within 1.63ms. These eventually consistent configurations were 16.5% and 59.5% faster than their strongly consistent counterparts at the 99.9th percentile. Yammer's data stores experienced a 99.9% inconsistency window of 202ms at 81.1% latency reduction. The results confirmed the anecdotal evidence: eventually consistent stores are often faster than their strongly consistent counterparts, and they are frequently consistent within tens or hundreds of milliseconds.

In order to make consistency prediction more accessible, with the help of the Cassandra community, we recently released support for PBS predictions in Cassandra 1.2.0. Cassandra users can now run predictions on their own production clusters to tune their consistency parameters and perform what-if analyses for normal-case, failure-free operation. For example, to explore the effect of adding solid-state drives (SSDs) to a set of servers, users can adjust the expected distribution of read and write speeds on the local node. These predictions are inexpensive; a JavaScript-based demonstration we have created⁴ completes tens of thousands of trials in less than a second.

Of course, prediction is not without faults: predictions are only as good as the underlying model and input data. As statistician George E.P. Box famously stated, "All models are wrong, but some are useful." Failure to account for an important aspect of the system or anti-entropy protocol may lead to

inaccurate predictions. Similarly, prediction works by assuming that past behavior is correlated with future behavior. If environmental conditions change, predictions may be of limited accuracy. These issues are fundamental to the problem at hand, and they are a reminder that prediction is best paired with measurement to ensure accuracy.

Eventual consistency is often strongly consistent. In addition to PBS, several recent projects have verified the consistency of real-world eventually consistent stores. One study found that Amazon SimpleDB's inconsistency window for eventually consistent reads was almost always less than 500ms,²⁴ while another study found that Amazon S3's inconsistency window lasted up to 12 seconds.⁷ Other recent work shows results similar to those presented for PBS, with Cassandra closing its inconsistency window within around 200ms.¹⁸

These results confirm the anecdotal evidence that eventual consistency is often "good enough" by providing quantitative metrics for system behavior. As techniques such as PBS and consistency measurement continue to make their way into more production infrastructure, reasoning about the behavior of eventual consistency across deployments, failures, and system configurations will be increasingly straightforward.

Programming Eventual Consistency

While users can verify and predict the consistency behavior of eventually consistent systems, these techniques do

not provide absolute guarantees against safety violations. What if an application requires that safety is always respected? There is a growing body of knowledge about how to program and reason about eventually consistent stores.

Compensation, costs, and benefits. Programming around consistency anomalies is similar to speculation: you do not know what the latest value of a given data item is, but you can proceed as if the value presented is the latest. When you have guessed wrong, you have to compensate for any incorrect actions taken in the interim. In effect, compensation is a way to achieve safety retroactively—to restore guarantees to users.¹³ Compensation ensures mistakes are eventually corrected but does not guarantee mistakes are not made.

As an example of speculation and compensation, consider running an ATM machine.^{8,13} Without strong con-

tency just as well as other errors such as data-entry mistakes.

An application designer deciding whether to use eventual consistency faces a choice. In effect, the designer needs to weigh the benefit of weak consistency B (in terms of high availability or low latency) against the cost C of each inconsistency anomaly multiplied by the rate of anomalies R :

maximize $B - RC$

This decision is, by necessity, application and deployment specific. The cost of anomalies is determined by the cost of compensation: too many overdrafts might cause customers to leave a bank, while propagation of status updates that is too slow might cause users to leave a social network. The rate of anomalies—as seen before—depends on the system architecture, configura-

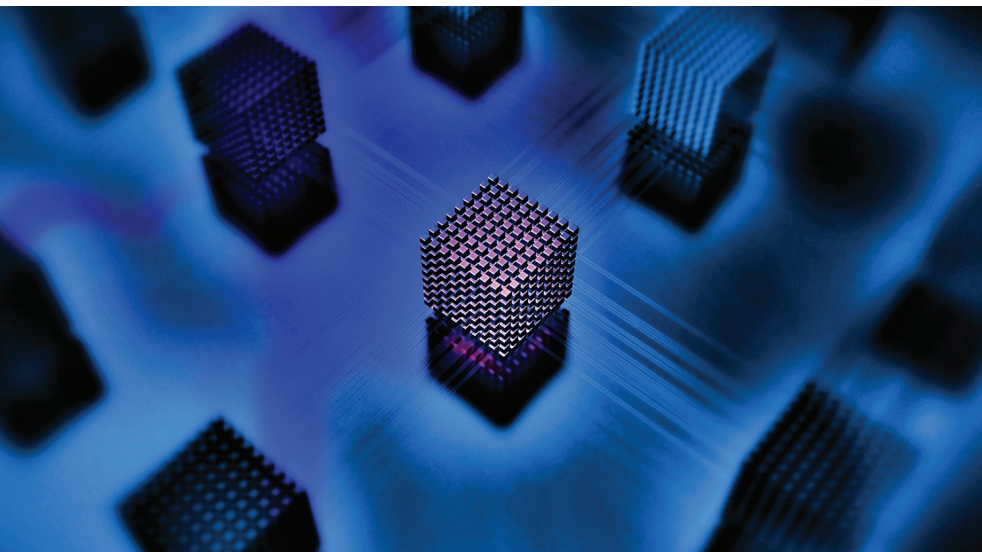
when the cost of inconsistency is high, with tangible monetary consequences (for example, ATMs), compensation is more likely to be well thought out.

For some applications, however, the rate of anomalies may be low enough or the cost of inconsistency sufficiently small so that the application designer may choose to forgo including compensation entirely. If the chance of inconsistency is sufficiently low, users may experience anomalies in only a small number of cases. Anecdotally, many online services such as social networking largely operate with weakly consistent configurations: if a user's status update takes seconds or even minutes to propagate to followers, they are unlikely to notice or even care. The complexities of operating a strongly consistent service at scale may outweigh the benefit of, say, preventing an off-by-one error in Justin Bieber's follower count on Twitter.

Compensation by design. Compensation is error prone and laborious, and it exposes the programmer (and sometimes the application) to the effects of replication. What if you could program without it? Recent research has provided “compensation-free” programming for many eventually consistent applications.

The formal underpinnings of eventually consistent programs that are consistent by design are captured by the CALM theorem, indicating which programs are safe under eventual consistency and also (conservatively) what is not.³ Formally, CALM means *consistency as logical monotonicity*; informally, it means programs that are monotonic, or compute an ever-growing set of facts (by for example, receiving new messages or performing operations on behalf of a client) and do not ever “retract” facts that they emit (that is, the basis for decisions it has already made do not change), can always be safely run on an eventually consistent store. (Full disclosure: CALM was developed by colleagues at UC Berkeley). Accordingly, CALM tells programmers which operations and programs can guarantee safety when used in an eventually consistent system. Any code that fails CALM tests is a candidate for stronger coordination mechanisms.

As a concrete example of this logical monotonicity, consider build-



sistency, two users might simultaneously withdraw money from an account and end up with more money than the account ever held. Would a bank ever want this behavior? In practice, yes. An ATM's ability to dispense money (availability) outweighs the cost of temporary inconsistency in the event that an ATM is partitioned from the master bank branch's servers. In the event of overdrawing an account, banks have a well-defined system of external compensating actions: for example, overdraft fees charged to the user. Banking software is often used to illustrate the need for strong consistency, but in practice the socio-technical system of the bank can deal with data inconsis-

tion, and deployment. Similarly, the benefit of weak consistency is itself possibly a compound term composed of factors such as the incidence of communication failures and communication latency.


Second, application designers actually must design for compensation. Writing corner-case compensation code is nontrivial. Determining the correct business application logic to handle each type of consistency anomaly is a difficult task. Carefully reasoning about each possible sequence of anomalies and the correct “apologies” to make to the user for each can become more onerous than designing a solution for strong consistency. In general,

ing a database for queries on stock trades. Once completed, trades cannot change, so any answers given that are based solely on the immutable *historical* data will remain true. However, if your database keeps track of the value of the *latest* trade, then new information—such as new stock prices—might retract old information, as new stock prices overwrite the latest ones in the database. Without coordination between replica copies, the second database may return inconsistent data.


By analyzing programs for monotonicity, you can “bless” monotonic programs as “safe” under eventual consistency and encourage the use of coordination protocols (such as strong consistency) in the presence of non-monotonicity. As a general rule, operations such as initializing variables, accumulating set members, and testing a threshold condition are monotonic. In contrast, operations such as variable overwrites, set deletion, counter resets, and negation (such as, “there does not exist a trade such that...”) are generally not logically monotonic.

CALM captures a wide space of design patterns sometimes referred to as ACID 2.0 (associativity, commutativity, idempotence, and distributed).¹³ *Associativity* means you can apply a function in any order: $f(a, f(b, c)) = f(f(a, b), c)$. *Commutativity* means a function’s arguments are order-insensitive: $f(a, b) = f(b, a)$. Commutative and associative programs are order-insensitive and can tolerate message reordering, as in eventual consistency. *Idempotence* means you can call a function on the same input any number of times and get the same result: $f(f(x)) = f(x)$ (for example, $\max(42, \max(42, 42)) = 42$). Idempotence allows the use of at-least-once message delivery, instead of at-most-once delivery (which is more expensive to guarantee). *Distributed* is primarily a placeholder for *D* in the acronym (!) but symbolizes the fact that ACID 2.0 is all about distributed systems. Carefully applying these design patterns can achieve logical monotonicity.

Recent work on CRDTs (commutative, replicated data types) embodies CALM and ACID 2.0 principles within a variety of standard data types, providing provably eventually consistent data structures including sets, graphs, and sequences.²⁰ Any program that correct-



The complexities of operating a strongly consistent service at scale may outweigh the benefit of, say, preventing an off-by-one error in Justin Bieber’s follower count on Twitter.



ly uses these predefined, well-specified data structures is guaranteed to never see any safety violations.

To understand CRDTs, consider building an increment-only counter that is replicated on two servers. We might implement the increment operation by first reading the counter’s value on one replica, incrementing the value by one, and writing the new value back every replica. If the counter is initially 0 and two different users simultaneously initiate increment operations at two separate, then both users may then read 0 and then distribute the value 1 to the replicas; the counter ends up with a value of 1 instead of the correct value of 2. Instead, we can use a G-counter CRDT, which relies on the fact that *increment* is a commutative operation—it does not matter in what order the two *increment* operations are applied, as long as they are both eventually applied at all sites. With a G-counter, the current counter status is represented as the count of distinct *increment* invocations, similar to how counting is introduced at the grade-school level: by making a tally mark for every increment then summing the total. In our example, instead of reading and writing counter *values*, each invocation distributes an increment *operation*. All replicas end up with two increment operations, which sum to the correct value of 2. This works because replicas understand the semantics of increment operations instead of providing general-purpose read/write operations, which are not commutative.

A key property of these advances is that they separate data store and application-level consistency concerns. While the underlying store may return inconsistent data at the level of reads and writes, CALM, ACID 2.0, and CRDT appeal to *higher-level* consistency criteria, typically in the form of application-level invariants that the application maintains. Instead of requiring that every read and write to and from the data store is strongly consistent, the application simply has to ensure a semantic guarantee (say, “the counter is strictly increasing”)—granting considerable leeway in how reads and writes are processed. This distinction between application-level and read/write consistency is often ambiguous

and poorly defined (for example, what does database ACID “consistency” have to do with “strong consistency”?). Fortunately, by identifying a large class of programs and data types that are tolerant of weak consistency, programmers can enjoy “strong” application consistency, while reaping the benefits of “weak” distributed read/write consistency.


Taken together, the CALM theorem and CRDTs make a powerful toolkit for achieving “consistency without concurrency control,” which is making its way into real-world systems. Our team’s work on the Bloom language³ embodies CALM principles. Bloom encourages the use of order-insensitive disorderly programming, which is key to architecting eventually consistent systems. Some of our recent work focuses on building custom eventually consistent data types whose correctness is grounded in formal mathematical lattice theory. Concurrently, several open source projects such as Statebox²¹ provide CRDT-like primitives as client-side extensions to eventually consistent stores, while one eventually consistent store—Riak—recently announced alpha support for CRDTs as a first-class server-side primitive.⁹

Stronger Than Eventual


While compensating actions and CALM/CRDTs provide a way around eventual consistency, they have shortcomings of their own. The former requires dealing with inconsistencies outside the system and the latter limits the operations that an application writer can employ. However, it turns out that it is possible to provide even stronger guarantees than eventual consistency—albeit weaker than SSI—for general-purpose operations while still providing availability.

The CAP theorem dictates that strong consistency (SSI) and availability are unachievable in the presence of partitions. But how weak does the consistency model have to be in order for it to be available? Clearly, eventual consistency, which simply provides a liveness guarantee, is available. Is it possible to strengthen eventual consistency by adding safety guarantees to it without losing its benefits?

Pushing the limits. A recent techni-



By analyzing programs for monotonicity, you can “bless” monotonic programs as “safe” under eventual consistency and encourage the use of coordination protocols in the presence of non-monotonicity.



cal report from the University of Texas at Austin claims no consistency model stronger than causal consistency is available in the presence of partitions.¹⁷ Causal consistency guarantees each process’s writes are seen in order writes follow reads (if a user reads a value $A=5$ and then writes $B=10$, then another user cannot read $B=10$ and subsequently read an older value of A than 5), and transitive data dependencies hold. This causal consistency is useful in ensuring, for example, comment threads are seen in the correct order, without dangling replies, and users’ privacy settings are applied to the appropriate data. The UT Austin report demonstrates that it is not possible to have a stronger model than causal consistency (that accepts fewer outcomes) without violating either high availability or ensuring that, if two servers communicate, they will agree on the same set of values for their data items. While many other available models are neither stronger nor weaker than causal consistency, this impossibility result is useful because it places an upper bound on a very familiar consistency model.

In light of the UT Austin result, several new data storage designs provide causal consistency. The COPS and Eiger systems¹⁶ developed by a team from Princeton, CMU, and Intel Research provide causal consistency without incurring high latencies across geographically distant datacenters or the loss of availability in the event of data-center failures. These systems perform particularly well, at a near-negligible cost to performance when compared to eventual consistency; Eiger, which was prototyped within the Cassandra system, incurs less than 7% overhead for one of Facebook’s workloads. In our recent work, we demonstrated how existing data stores that are already deployed in production but provide eventual consistency can be augmented with causality as an added safety guarantee.⁶ Causality can be *bolted-on* without compromising high availability, enabling system designs in which safety and liveness are cleanly decomposed into separate architectural layers.

In addition to causality, we can consider the relationship between ACID transactions and the CAP theorem. While it is impossible to provide the

gold standard of ACID isolation—serializability, or SSI—it turns out many ACID databases provide a weaker form of isolation, such as read committed, often by default and, in some cases, as the maximum offered. Some of our recent results show many of these weaker models *can* be implemented in a distributed environment while providing high availability.⁵ Current databases providing these weak isolation models are unavailable, but this is only because they have been implemented with unavailable algorithms.


We—and several others—are developing transactional algorithms that show this need not be the case. By rethinking the concurrency-control mechanisms and re-architecting distributed databases from the ground up, we can provide safety guarantees in the form of transactional atomicity, ANSI SQL Read Committed and Repeatable Read, and causality between transactions—matching many existing ACID databases—without violating high availability. This is somewhat surprising, as many in the past have assumed that, in a highly available system, arbitrary multi-object transactions are out of the question.

Recognizing the limits. While these results push the limits of what is achievable with high availability, there are several properties that a weakly consistent system will never be able to provide; there is a fundamental cost to remaining highly available (and providing guaranteed low latency). The CAP theorem states that making staleness guarantees is impossible in a highly available system. Reads that specify a constraint on data recency (for example, “give me the latest value,” “give me the latest value as of 10 minutes ago”) are not generally available in the presence of long-lasting network partitions. Similarly, we cannot maintain arbitrary global correctness constraints over sets of data items such as uniqueness requirements (for example, “create bank account with ID 50 if the account does not exist”) and, in certain cases (for example, arbitrary reads and writes), even correctness constraints on individual data items are not achievable (for example, “the bank account balance should be non-negative”). These challenges are an inherent cost of choosing weak consistency—whether eventual or a stronger but still “weak” model.

Conclusion

By simplifying the design and operation of distributed services, eventual consistency improves availability and performance at the cost of semantic guarantees to applications. While eventual consistency is a particularly weak property, eventually consistent stores often deliver consistent data, and new techniques for measurement and prediction grant us insight into the behavior of eventually consistent stores. Concurrently, new research and prototypes for building eventually consistent data types and programs are easing the burden of reasoning about disorder in distributed systems. These techniques, coupled with new results pushing the boundaries of highly available systems—including causality and transactions—make a strong case for the continued adoption of weakly consistent systems. While eventual consistency and its weakly consistent cousins are not perfect for every task, their performance and availability implications will likely continue to accrue admirers and advocates in the future.

Acknowledgments

The authors would like to thank Peter Alvaro, Carlos Baquero, Neil Conway, Alan Fekete, Joe Hellerstein, Marc Shapiro, and Ion Stoica for feedback on earlier drafts of this article. 

Related articles on queue.acm.org

Eventually Consistent

Werner Vogels

<http://queue.acm.org/detail.cfm?id=1466448>

BASE: An Acid Alternative

Dan Pritchett

<http://queue.acm.org/detail.cfm?id=1394128>

Scalable SQL

Michael Rys

<http://queue.acm.org/detail.cfm?id=1971597>

References

1. Abadi, D. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer* (Feb. 2012).
2. Alpern, B. and Schneider, F.B. Defining liveness. *Information Processing Letters* 21 (Oct. 1985).
3. Alvaro, P., Conway, N., Hellerstein, J. and Marczak, W. 2011. Consistency analysis in Bloom: A CALM and collected approach. *Proceedings of the Conference on Innovative Data Systems Research* (2011).
4. Bailis, P., Venkataraman, S., Franklin, M., Hellerstein, J. and Stoica, I. Probabilistically bounded staleness

for practical partial quorums. In *Proceedings of Very Large Databases* (2012). (Demo from text: <http://pbs.cs.berkeley.edu/#demo>)

5. Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J. and Stoica, I. HAT, not CAP: Highly available transactions. *arXiv:1302.0309* (Feb. 2013).
6. Bailis, P., Ghodsi, A., Hellerstein, J. and Stoica, I. Bolt-on Causal Consistency. In *Proceedings of ACM SIGMOD* (2013).
7. Bermbach, D. and Tai, S. Eventual consistency: how soon is eventual? An evaluation of Amazon S3's consistency behavior. In *Proceedings of Workshop on Middleware for Service-Oriented Computing* (2011).
8. Brewer, E. CAP twelve years later: How the “rules” have changed. *IEEE Computer* (Feb. 2012).
9. Brown, R. and Cribbs, S. Data structures in Riak (2012); <https://speakerdeck.com/basho/data-structures-in-riak>. RICON Conference.
10. Davidson, S., Garcia-Molina, H. and Skeen, D. Consistency in a partitioned network: A survey. *ACM Computing Surveys* 17, 3 (1985).
11. Gilbert, S. and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News* 33, 2 (June 2002).
12. Hale, C. You can't sacrifice partition tolerance (2010); <http://codahale.com/you-cant-sacrifice-partition-tolerance/>
13. Helland, P. and Campbell, D. Building on quicksand. In *Proceedings of the Conference on Innovative Data Systems Research* (2009).
14. Johnson, P. R., Thomas, R. H. Maintenance of duplicate databases; RFC 677 (1975); <http://www.faqs.org/rfcs/rfc677.html>.
15. Kawell Jr., L., Beckhardt, S., Halvorsen, T., Ozzie, R. and Greif, I. Replicated document management in a group communication system. In *Proceedings of the 1988 ACM Conference on Computer-supported Cooperative Work*: 395; <http://dl.acm.org/citation.cfm?id=1024798>.
16. Lloyd, W., Freedman, M., Kaminsky, M. and Andersen, D. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of Networked Systems Design and Implementation* (2011).
17. Mahajan, P., Alvisi, L. and Dahlin, M. Consistency, availability, convergence. University of Texas at Austin TR-11-22 (May 2011).
18. Rahman, M., Golab, W., AuYoung, A., Keeton, K. and Wylie, J. Toward a principled framework for benchmarking consistency. *Workshop on Hot Topics in System Dependability* (2012).
19. Saito, Y. and Shapiro, M. Optimistic Replication. *ACM Computing Surveys* 37, 1 (Mar. 2005). <http://dl.acm.org/citation.cfm?id=1057980>
20. Shapiro, M., Preguiça, N., Baquero, C. and Zawirski, M. A comprehensive study of convergent and commutative replicated data types. INRIA Technical Report RR-7506 (Jan. 2011).
21. Statebox; <https://github.com/mochi/statebox>.
22. Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M. and Hauser, C. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings on Symposium on Operating Systems Principles* (1995).
23. Vogels, W. Eventually consistent. *ACM Queue*, (2008).
24. Wada, H., Fekete, A., Zhao, L., Lee, K., A. and Liu, A. Data consistency and the tradeoffs in commercial cloud storage: the consumers' perspective. *Proceedings of the Conference on Innovative Data Systems Research* (2011).
25. Yu, H. and Vahdat, A. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. on Computer Systems* (2002).

Peter Bailis is a graduate student of computer science in the AMPLab and BOOM projects at UC Berkeley, where he works closely with Ali Ghodsi, Joe Hellerstein, and Ion Stoica. He currently studies distributed systems and databases, with a particular focus on distributed consistency models. He blogs at <http://bailis.org/blog> and tweets as @pbailis.

Ali Ghodsi (alig@cs.berkeley.edu) is an assistant professor at KTH/Royal Institute of Technology in Sweden and a Visiting Researcher at UC Berkeley since 2009. His general interests are in the broader areas of distributed systems, and networking. He received his Ph.D. in 2006 from KTH/Royal Institute of Technology in the area of distributed computing.

Copyright of Communications of the ACM is the property of Association for Computing Machinery and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.