Bolt-on Causal Consistency

Peter Bailis[†], Ali Ghodsi[†,‡], Joseph M. Hellerstein[†], Ion Stoica[†]
[†] UC Berkeley [‡] KTH/Royal Institute of Technology

Shallow men believe in luck. . . Strong men believe in cause and effect.—Ralph Waldo Emerson

ABSTRACT

We consider the problem of separating consistency-related safety properties from availability and durability in distributed data stores via the application of a "bolt-on" shim layer that upgrades the safety of an underlying general-purpose data store. This shim provides the same consistency guarantees atop a wide range of widely deployed but often inflexible stores. As causal consistency is one of the strongest consistency models that remain available during system partitions, we develop a shim layer that upgrades eventually consistent stores to provide convergent causal consistency. Accordingly, we leverage widely deployed eventually consistent infrastructure as a common substrate for providing causal guarantees. We describe algorithms and shim implementations that are suitable for a large class of application-level causality relationships and evaluate our techniques using an existing, production-ready data store and with real-world explicit causality relationships.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Distributed databases*

Keywords

causal consistency, eventual consistency, separation of concerns

1 Introduction

Choosing an appropriate consistency model is difficult. Strong data consistency models are easy to reason about and simplify distributed systems programming. However, in the presence of partitions between replicas, designers face a choice between strong data consistency and availability [27]; the many services that require "always on" operation must sacrifice "strong" consistency [17, 22, 34, 47]. Without partitions, there is a fundamental trade-off between latency and consistency: intuitively, weaker consistency models require "less coordination," lowering operation latency [2, 8], which is lower-bounded by the laws of physics [21]. As systems continue to scale and are geo-replicated, designers must face these consistency trade-offs head-on.

Modern data stores typically provide some form of *eventual consistency*, which guarantees that eventually—in the absence of new writes, at some future time unknown in advance—all replicas of each data item will *converge* to the same value [49]. Eventual consistency is a *liveness* property, which ensures that something good will eventually happen (replicas agree) [5]. It does not provide *safety* guarantees: it is impossible for a system to violate eventual consistency at any discrete moment because the system may converge in the future [7]. By itself, eventual consistency is a weak consistency model. If users want stronger guarantees, stores may provide stronger models that, in addition to convergence, provide safety. However, exact guarantees differ across stores and sometimes between releases of the same stores.

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

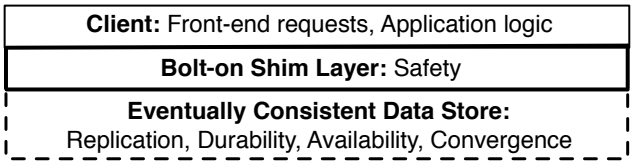


Figure 1: Separation of concerns in the bolt-on architecture. In this work, a narrow shim layer upgrades an underlying eventually consistent data store to provide causal consistency.

In this paper we adopt a general approach that separates architectural concerns of liveness, replication, and durability from the semantics of safety-related data consistency. We treat consistency-related safety as a *bolt-on* property provided by a shim layer on top of a general-purpose data store. This allows a clean, layered separation of safety and liveness concerns: the underlying store is responsible for liveness, replication, durability, and convergence, which require substantial engineering effort to make practically usable. Safety properties, which are algorithmic in nature and often hard to implement correctly, are kept separate and located in a separate layer atop the general-purpose store. This enables a shim to provide the exact same consistency model regardless of the implementation or semantics of the underlying store, even if provided as an online service (e.g., DynamoDB). This architecture can also lead to standardized implementations that bring clarity and portability to distributed consistency, a design space occupied by an array of custom implementations that often differ only slightly in the consistency guarantees they provide.

In this work, we take a first step in the direction of the general approach outlined above and develop a bolt-on architecture to provide causal consistency on top of eventually consistent stores (Figure 1). Causal consistency guarantees that reads will obey causality relationships between writes: if one event "influences" a subsequent operation, a client cannot observe the second without the first [4, 31]. Our choice of causal consistency is motivated by several recent developments. Multiple "NoSQL" stores have added stronger consistency, implying that there is a demand for consistency guarantees that are stronger than eventual [50]. However service availability is crucial for many applications, as many companies observe negative business impact in the face of unavailability or increased latency [33]. Towards these goals, recent work has shown that no model stronger than causal consistency is achievable with high availability in the presence of partitions [36]. Thus, we upgrade one of the weakest highly available models—eventual—to one of the strongest. Moreover, causal consistency allows purely

local reads, enabling better performance than stronger models that often require inter-replica communication on every operation. As a final, more pragmatic motivation, while there has recently been considerable academic interest in causal consistency [6, 34, 36], no production-ready data stores provide it, so bolt-on causal consistency fills a gap in current technology.

There are two main challenges in providing causal consistency in a bolt-on architecture: handling overwritten histories and maintaining availability. First, when there are multiple writes to a given data item, the eventually consistent store often chooses a single, winning write. Attempting to directly use existing algorithms for causal consistency [4, 9, 29, 34, 45] by using the underlying store as a communication medium will fail because the bolt-on layer is not guaranteed to see all writes. We refer to this as the problem of *overwritten histories*, or lost metadata due to data store overwrites. Second, we need to ensure that the shim maintains availability (one of the key benefits of causal consistency) in the presence of partitions. The shim should never block waiting for a causal dependency that is either delayed or may never arrive (due to overwritten histories). If we employ an existing algorithm that assumes reliable write delivery, then the shim will either become unavailable or lose convergence properties in the presence of overwrites.

The main innovation in our bolt-on shim is an algorithm that buffers writes and ensures that each write satisfies the criteria for a *causal cut* across the writes it has observed. By developing a declarative specification for causal cuts, we determine what writes to either synchronously or asynchronously fetch from the underlying data store as well as what metadata to store with each write. Only when a causal cut is satisfied are writes made visible to shim clients. This guarantees that the shim is always causally consistent, convergent, and highly available.

In this paper, we make the following contributions:

- We describe a system architecture for achieving convergent causal consistency by applying a narrow bolt-on shim layer to an eventually consistent data store.
- We present a solution to the problem of overwritten histories, which arises from most eventually consistent stores' register semantics. Solving this problem key to achieving convergent bolt-on causal consistency and requires a generalized, declarative specification for causal consistency.
- We evaluate our shim implementation on top of Cassandra [30], a production-ready eventually consistent data store, using real-world *explicit causal consistency* traces. For many traces, we achieve peak throughput within a factor of two and often within 25% of eventually consistent operation, and, for one variant of bolt-on causal consistency, outperform default eventually consistent operation.

2 Background

Service operators today are faced with a wide range of consistency models. In this section, we provide background on two major models: eventual consistency and causal consistency. Readers familiar with these consistency models may wish to proceed to Section 3.

Weak Consistency Many distributed systems designs choose to employ weak consistency, placing few restrictions on the evolution of data values. In the presence of partitions between participants in the system, services that must remain available, or "always-on," have no choice: systems cannot maintain both availability of read and write operations and strong consistency (e.g., serializability [20], linearizability [27]) in the presence of network partitions. Even without partitions (or with duration-bounded partitions), systems face a trade-off between latency and consistency: achieving stronger models like atomicity or serializability requires more coordination overhead than achieving "weaker" ones [2, 8]. In this work, we consider the relationship between two important weak consistency models: eventual consistency and causal consistency.

Eventual Consistency Among weak consistency models, eventual consistency is one of the most widely deployed [22, 34, 49]. Under eventual consistency, if writes stop, then, processes will—at some point in time and forever afterwards—all agree on the same value for each object in the system. This *convergence* guarantee provided by eventual consistency is a liveness property [36]: the property that all participants agree will *eventually* hold. However, this is *not* a safety guarantee: a system cannot "violate" eventual consistency at any fixed point in time—there is always the possibility that it becomes consistent later [7]. Thus, at any given time, clients may witness a wide range of consistency anomalies.

Despite these weak guarantees, a wide range of distributed storage systems offer a binary choice between (bare-minimum) eventual and some form of strong consistency. Real-world systems adopting the Dynamo [22] replication model (e.g., Cassandra, Riak, Voldemort), many hosted systems (e.g., DynamoDB, Amazon S3 and SimpleDB, Google App Engine Store), and master-slave replicated stores supporting reading from slaves (e.g., HBase, MongoDB, many relational stores) all offer this choice. Accordingly, some form of strong, potentially unavailable consistency is often the only option if one opts for anything stronger than eventual consistency. Unless one is willing to change the implementations of these data stores, in the presence of partitions, read availability may require accepting eventual consistency.

Causal Consistency In contrast with eventual consistency's weak guarantees, causal consistency is the strongest consistency model that is available in the presence of partitions [34, 36]. Informally, a system is causally consistent if participants can never observe data items before they can observe items that influenced their creation.

Consider the following hypothetical scenario on a social networking site like Facebook [6]:

1. Sally cannot find her son Billy. She posts update S to her friends: "I think Billy is missing!"
2. Momentarily after Sally posts S, Billy calls his mother to let her know that he is at a friend's house. Sally edits S, resulting in S^*: "False alarm! Billy went out to play."
3. Sally's friend James observes S^* and posts status J in response: "What a relief!"

If causality is not respected, a third user, Henry, could perceive effects before their causes; if Henry observes S and J but not S^*, he might think James is pleased to hear of Billy's would-be disappearance! If the site had respected causality, then Henry could not have observed J without S^*.

Maintaining these causal relationships is important for online services. Human perception of causality between events is well studied in the fields of in psychology and philosophy [38]; just as space and time impose causality on real-world events, maintaining causal relationships between virtual events is an important consideration when building human-facing services. Sometimes, maintaining causality is insufficient, and stronger models are required [16]. However, especially given its availability and latency benefits, causal consistency is often a reasonable choice [6, 11, 34].

Defining Causal Consistency Under causal consistency, each process's reads must obey a partial order called the `happens-before` relation (denoted $\rightarrow$). Application operations create ordering constraints between operations in the `happens-before` relation, and the system enforces the order. `happens-before` is transitive, and

all writes that precede a given write in the happens-before relation comprise its causal *history*. Operations that are causally unrelated are called concurrent (denoted ||). To provide a total ordering across versions, we can use *commutative merge functions* [34, 41]: given v and v', two concurrent writes to a single object, $merge(v,v') = merge(v',v)$. As we will see, wall-clock "last-writer-wins" is a common commutative merging policy that satisfies this requirement. If desired, we can place the merged version so that v and v' both happen-before it. Schwarz and Mattern [46] provide an in-depth overview of causal consistency.

Capturing Causality Causal relationships (happens-before) are typically defined in two ways: via potential and explicit causality.

Under *potential causality* [4, 31] all writes that could have influenced a write must be visible before the write is made visible. Potential causality represents *all possible* influences via (transitive) data dependencies (á la the "Butterfly Effect"). The potential causality relation respects program ordering along with the causality of any data items an agent reads. Future writes reflect any reads: if process P_i reads write w_i, then all other processes must also be able to read w_i before P_i's future writes.

We can also capture *explicit causality* between operations: a program can define its own causal relationships [6, 29]. Causality tracking mechanisms are often exposed through user interfaces: in our prior example, James likely clicked a "reply" button in order to input J as a response to S^*. The application can capture these *explicit* relationships between replies and define a separate causal relation for each conversation. Of course, if James did not explicitly signal that J was a response to S (say, he posted to Sally's feed without pressing "reply"), we would have to rely on potential causality. However, given its scalability benefits [6], *in this work, we consider the problem of maintaining explicit causality relationships*.

Convergence We can trivially satisfy causal consistency by storing a local copy of each key and never communicating updates [6, 36]. This strategy does not require coordination and is easy to implement but is not particularly useful. In the presence of partitions between all processes, every causally consistent system will behave like our trivial implementation, so we must admit it as valid. However, in the absence of partitions, we would prefer different behavior. Coupling convergence properties with causality yields a useful model: *convergent causal consistency* [34, 36]. All processes eventually agree on the same value for each item (liveness), and the processes observe causality throughout (safety). Convergent causal consistency is eventually consistent, but the converse does not hold: eventual consistency is not causally consistent.

3 Architecture

In this section, we propose a "bolt-on" architecture that separates implementation and design of consistency safety properties from liveness, replication, and durability concerns. We subsequently introduce a specific instance of the bolt-on architecture—the primary subject of this paper—that upgrades an eventually consistent store to provide causal consistency.

3.1 Separation of Concerns

In this paper, we consider a bolt-on architecture in which an underlying data store handles most aspects of data management, including replication, availability, and convergence. External to the data store, a *shim* layer composed of multiple shim processes upgrades the consistency guarantees that the system provides. The underlying data store allows a large space of read and write histories; it is the shim's job to restrict the space of system executions to those that match the desired consistency model.

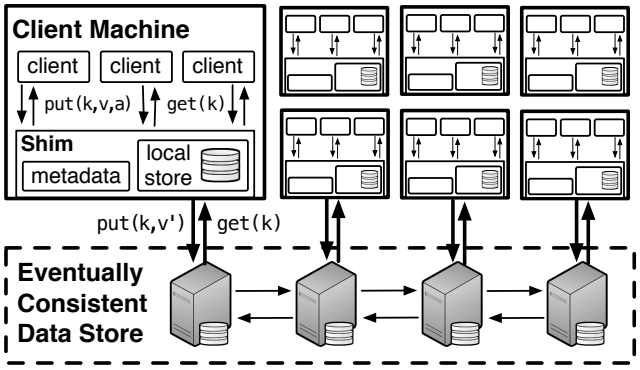


Figure 2: Bolt-on architecture: a causally consistent shim layer mediates access to an underlying eventually consistent data store.

There are several advantages to a layered, bolt-on approach to consistency. A bolt-on architecture decouples the implementation of safety properties from the implementation of liveness properties [5]. This enables a clean separation of concerns. The underlying store is responsible for liveness, handling replication, durability, and convergence. Achieving these goals typically requires substantial engineering effort: many commercial and open source projects have spent considerable effort in implementing and refining this functionality. The bolt-on shim layer provides safety, which restricts the type of consistency violations that can be exposed to the user. This safety functionality is algorithmic in nature and often difficult to implement correctly. By separating these concerns, systems can deploy thin, bolt-on layers that have been carefully proven and tested. Separately, practical issues dealing with technology, hardware, and distribution can be addressed in the lower layers. Today, systems do not separate safety and liveness properties. This makes reasoning about data consistency difficult, as systems often provide slightly different consistency guarantees. A bolt-on architecture can provide the same consistency model across a wide range of storage systems.

A correct shim implementation should not violate the properties guaranteed by the underlying data store. For example, if, as we propose, the underlying store is tasked with providing liveness guarantees, then a correct shim should always be live as well; an incorrect shim implementation can violate liveness by blocking indefinitely, but, if it does, it will not be due to the underlying store. Similarly, if the shim is not responsible for durability or fault-tolerance, which are concerns delegated to the underlying store, then information contained in the shim should be treated as "soft state" that is reconstructible from the underlying store. If the shim needs to be available in the presence of partitions, it should not depend on other shim processes: intra-shim communication can be used as an optimization but cannot be required for availability.

3.2 Bolt-on Causal Consistency

In the bolt-on causal consistency architecture (Figure 2), a shim upgrades an eventually consistent data store (hereafter, *ECDS*) to provide convergent causal consistency. The shim is tasked only with causal safety guarantees, and replication, availability, durability, and convergence are guaranteed by the *ECDS*. Clients make requests to the shim layer, which handles all communication with an underlying *ECDS* and other clients. Clients are agnostic to the *ECDS* choice. For simplicity, we consider simple key-value stores, but our results generalize to richer data models. The shim exposes two operations: get(key) and put(key, value, after), where after is any set of previously returned writes (or write identifiers)—or none—that specifies the happens-before relation for the write.

The *ECDS* in turn exposes two operations to the shim layer: `get(key)` and `put(key,value)`. The shim may exercise these methods as often as it prefers, and the keys and values it stores need not exactly mirror the keys and values that clients store. As an initial design and implementation, we treat the shim as a client-side library, but we discuss other approaches in Section 5.3.

4 Bolt-on Challenges

Prior algorithms for achieving causal consistency do not provide highly available, convergent bolt-on causal consistency because they do not address the problem of *overwritten histories*. In this section, we outline existing algorithms, introduce overwritten histories, and state our assumptions for *ECDS* behavior.

4.1 Implementing Causal Consistency

Implementations of causal consistency usually have two components: dependency tracking and local stores. *Dependency tracking* metadata attached to each write is used to record information about its causal history, determining the write's `happens-before` relation. Processes keep a local copy of keys and values—called a *local store*—that is guaranteed (by design) to be causally consistent at all times. Processes serve reads from their local stores and, using dependency tracking, determine when to update their local stores with newer writes (*apply* them).

The causal memory implementation by Ahamad et al. [4] is typical of many others, like causal delivery [45]. When performing a write, a process first updates its local store, then sends the new write via *reliable broadcast* to all other processes, who buffer it. A process applies a buffered write (originating from another process) to its local data store once it has applied all of the write's causal predecessors. If each process receives all writes, each process will apply all writes. Each process can safely update its local store with its own writes without checking dependencies because, by construction, it has already applied each of their dependencies.

As an example, consider the following causal graph. We denote a write to key k with unique version i as k_i, so our graph contains two writes: a write x_1 to key x and a second write y_1 to key y such that x_1 `happens-before` y_1:

$$x_1 \longrightarrow y_1$$

According to the definition of causal consistency, if a process reads y_1, then its subsequent reads from x must return x_1 or another write x_i such that $x_i \nrightarrow x_1$ (i.e., $x_1 \,||\, x_i$ or $x_1 \rightarrow x_i$). It is not causally consistent if a process reads y_1 then reads the initial value (`null`) for x because `happens-before` is not respected.

To better understand the role of the local store in a causally consistent system, consider a scenario in which x_1 and y_1 are generated by two different processes that each send their writes to a third process—say, process P—that wishes to read y. Depending on the delivery order of the messages containing x_1 and y_1, P will read different values under causal consistency:

1. P has not received any writes. P will return $y =$ `null`.
2. P has received write x_1 but not write y_1. P can apply x_1 to its local store but P will still return $y =$ `null`.
3. P has received write y_1 but not write x_1. P will buffer write y_1 because it has not yet applied all of the y_1's dependencies ($\{x_1\}$). P will again return $y =$ `null`.
4. P has received both of the x_1 and y_1 writes. It can apply x_1 then y_1 to its local store and it will return $y = y_1$.

The third case is most interesting; what would happen if we allowed P to observe $y = y_1$? If we did, then if P were to subsequently read from x, to preserve causal safety, it would have to block until it received x_1 (or a suitable substitute for x_1, as above). During network partitions, this could take an indefinite amount of time and P would lose the availability benefits of causal consistency. Without network partitions, P might have to block due to network latency.

With reliable channels, existing algorithms provide both safe and live implementations of causal consistency [4, 29, 34].

4.2 Overwritten Histories

If process do not see all writes (i.e., delivery is unreliable), then existing techniques—which assume reliable delivery—will not provide convergent causal consistency with high availability. This phenomenon occurs when we use the *ECDS* as a write delivery mechanism. Consider the following example, where one process writes x_1, y_1, and z_1, and a second process writes y_2, which is `concurrent` with the first process's updates:

$$x_1 \longrightarrow y_1 \longrightarrow z_1$$
$$y_2$$

In this example, if a third process receives z_1 and y_2 but not y_1, it may have problems: under prior approaches, if processes do not see all writes, they can *detect* missing dependencies (here, y_1) but cannot tell if the missing dependencies are needed or not (y_1 is not but x_1 is). If, as is common [29, 34], each shim stores the immediate dependencies between writes (e.g., z_1 contains a reference to y_1, then y_1 to x_1, and finally x_1 and y_2 to `null`) and a shim reads y_2 but not y_1, it will never learn of the dependency to x_1. The shim cannot be sure that there is not another dependency for z_1 that it is not aware of and will never be able to safely read z_1. This makes causality difficult to enforce. Accordingly, different shims may end up with different values for z and the system will not converge. In fact, if shims miss writes, any fixed-length chain of pointers a shim might attach (say, attach the last k dependencies) can be broken. Vector clocks [39] fall prey to a similar problem: was the missing write overwritten, or has the shim simply not seen it?

A bolt-on system may experience these missing writes due to *ECDS* behavior, or *overwritten histories*. The *ECDS* is responsible for write distribution: to read new versions generated by other processes, each shim reads from the *ECDS*. However, the *ECDS* may overwrite values before all shims have read them and therefore may not expose all writes: typical *ECDS* implementations provide register semantics, meaning that newer versions overwrite older versions of the same key. If we are lucky, shims may be able to read all versions, but we cannot guarantee that this will occur—the shim layer does not control *ECDS* convergence. If there are multiple writes to a key (i.e., overwrites), then, due to *ECDS* replication, we can only be guaranteed that the final, converged version will be visible to all processes. In the example above, if a shim process does not read y_1 by the time it is overwritten by y_2, it will—with existing algorithms—"lose history" due to overwrites, a problem we call overwritten histories. Prior algorithms do not address this problem [4, 9, 29, 34, 45]. They can detect missing updates in the history but, due to overwrites—as we demonstrated in the example above—cannot safely apply all writes, resulting in solutions that lack convergence or, alternatively, availability.

One way to think of *ECDS* write propagation is as an unreliable network that is guaranteed only to deliver a single message per key. We can build a reliable network between N shim processes by using N^2 keys, but this is inefficient. Alternatively, we can avoid overwrites entirely by storing a new data item for every write, but this requires storage linear in the number of writes ever performed. These solutions are both expensive. In this paper, we derive a different solution (Section 5) and leverage application context via explicit causality for efficiency.

4.3 Assumptions

Before addressing the problem of overwritten histories, we first state our assumptions regarding *ECDS* behavior. We opt for a pessimistic but general model that will work with a wide range of existing *ECDS*s. We discuss possible relaxations and their effect on our algorithms in Section 7.

We assume that the *ECDS* is eventually consistent (convergent) and that shim layer knows the *ECDS* merge function for overwrites along with any write-specific information that the *ECDS* uses (e.g., if the *ECDS* uses a last-writer-wins merge function, the shim processes can determine the timestamps for writes). This is useful for convergence, so that all shims eventually return the same version in the event of concurrent writes. We also assume that the *ECDS* provides single-value register semantics (in contrast with a store that might keep multiple versions—Section 7). Moreover, we assume there are no callbacks from the *ECDS*: as in existing systems, the *ECDS* does not directly notify its clients of new writes.

We assume that the converged *ECDS* will obey the partial happens-before order for each key. That is, if $x_1 \rightarrow x_2$, then the *ECDS* cannot converge to x_1. This is possibly problematic for arbitrary merge functions, but last-writer-wins, the default merge function employed by most eventually consistent stores by default (e.g., Cassandra [30], Riak, DynamoDB, and in master-slave systems supporting slave reads) satisfies this requirement. Our API, which requires writers to pass in existing dependencies in order to `put` them `after`, ensures that causality relations respect wall-clock order.

We co-locate the shim layer with (possibly many) clients. In the presence of failures, this means that clients and their shims will fail together (fate-sharing), and that clients will always contact the same shim (affinity). Furthermore, we assume that end users of the service will always contact the same set of shim processes for each session, or desired scope of causality. These "stickiness" assumptions are standard in existing implementations of causal consistency [4, 29, 34, 49].

5 Bolt-on Causal Consistency

In this section, we present a declarative specification for causal consistency and algorithms for achieving bolt-on causal consistency.

5.1 Causal Cuts

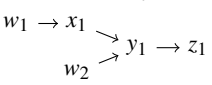
To begin, we declaratively describe the behavior of our causally consistent store. Prior definitions of causal consistency are expressed operationally, in terms of reads and writes, with implementations described in terms of carefully controlled buffering and admission to the local store. As we discussed in Section 4.2, bolt-on causality differs from prior approaches in both its write propagation mechanism (through the *ECDS*) and the resulting challenges (namely, overwritten histories). A bolt-on shim cannot wait for the *ECDS* to provide it to deliver new writes, especially as some writes may be "dropped" due to overwrites. To understand when it is safe to apply writes, we reformulate the correctness criteria for causal consistency as a more general, declarative specification that is inspired by the clarity provided by declarative translations of (operational) isolation level definitions [3].

The local store holds a "consistent" set of writes that can be read from and added to at any time without violating safety constraints. We formalize this notion of "consistency via *causal cuts*. Given a write w to key k, we denote k=w.key and the set of its transitive dependencies as w.deps:

Definition. *A causal cut is a set of writes C such that $\forall$ writes $w \in \bigcup_{c \in C} c.deps$, $\exists w' \in C$ such that $w'.key = w.key$ and $w' \nrightarrow w$ (equivalently, either $w = w'$, $w \rightarrow w'$, or $w \parallel w'$).*

This dense definition is simple to explain: the dependencies for each write in the causal cut should either *i.*) be in the cut, *ii.*) `happen-before` a write to the same key that is already in the cut, or *iii.*) be concurrent with a write to the same key that is already in the cut. This means that a cut contains the "latest" version of each key along at least one concurrent branch of the history ending along the cut's frontier. A causally consistent system maintains the invariant that each local store is a causal cut.

As an example, consider the history below:



The set of writes $\{w_1, x_1, y_1\}$ is a causal cut: the union of the dependencies for each write (equivalently, the inverse transitive closure of the graph beginning at the writes) is $\{w_1, w_2, x_1, y_1\}$, and, for each key (w, x, and y), we have the latest write for a concurrent branch in the history. Similarly, $\{x_1, w_2, y_1\}$ is a causal cut, as are $\{w_1, x_1, y_1, z_1\}$, $\{w_2, x_1, y_1, z_1\}$, $\{w_1\}$, $\{w_2\}$, $\{w_1, x_1\}$, and $\{w_2, x_1\}$. However, for example, $\{w_1, z_1\}$, $\{x_1, z_1\}$, and $\{y_1, z_1\}$ are not due to missing dependencies (in the examples: $\{x_1, y_1\}$, $\{y_1, w_1$ or $w_2\}$, and $\{x_1, w_1$ or $w_2\}$).

Our definition of causal cuts is inspired by *consistent cuts*, often used to capture a snapshot of global state in a distributed environment [37]. The primary difference between causal cuts and consistent cuts is that causal cuts specify relationships between writes to keys, whereas consistent cuts specify relationships between agents. One might be tempted to find a homomorphism from causal cuts to consistent cuts (e.g., keys as agents), but the challenge is that a single key can appear on multiple concurrent branches, whereas a causal agent, by (traditional) definition, cannot.

5.2 Achieving Bolt-on Causal Consistency

Given our definition of causal cuts, we can formulate an algorithm for achieving bolt-on causal consistency in the presence of *ECDS* overwrites. Our assumptions about the underlying data store (Section 4.3) prove challenging but match those we encounter in real-world *ECDS*s in Sections 6 and 7.

The basic premise of bolt-on causal consistency is straightforward: before revealing a write w to a client, a shim needs to ensure that w, along with the set of writes in its local store L (i.e., $\{w\} \cup L$) is a causal cut (w is *covered* by L). If so, it can apply w to L. The write propagation of the *ECDS* means shims need to take care in performing this check.

Write path When a client performs a write, the shim updates its local store and sends the write (along with any relevant metadata—described below) to the *ECDS* for distribution and durability. The write, along with the local store, forms a causal cut because the agent has only read updates from its local store and its dependencies must therefore also be present in the local store. Algorithm 1 shows the write path in the PUT_{shim} method.

Read path: Causal When a client performs a read, the shim can safely return the value present in its local store. Existing algorithms for causal consistency, pioneered by Ahamad et al. [4], use this model, in which background processes are responsible for updating the local store. Accordingly, porting traditional causal consistency algorithms to a bolt-on model, all reads can complete locally, and shims fetch new writes *asynchronously* from the *ECDS*.

Algorithm 1 also shows an implementation of both the local read operation (GET_{shim}) and the asynchronous *ECDS* write resolution (RESOLVE_{async}). GET_{shim} reads the local store and notifies a background resolver process that the process should check the *ECDS* for a newer version of the key (by adding the key to a *to_check*,

set of writes shared between readers and the resolver). Because the *ECDS* does not notify the shim about new writes, clients inform the resolver about which keys it should poll on.

The resolver (RESOLVE$_{async}$) is an asynchronous process tasked with updating the local store. Suppose the resolver reads w from the *ECDS*. If we wish to maintain the invariant that our local store L is always a causal cut of writes, then the resolver needs to check w's dependencies. The resolver process iterates through each of w's dependencies and ensures that the local store contains either the dependency itself, a later (`happens-after`) write to the dependency's key, or a concurrent write to the dependency's key. If a dependency is missing, the resolver can either attempt to read the dependency from the *ECDS* (as in the implementation) or defer further checking and move on to the next key. To prevent infinite recursion when "chasing" dependency chains in the *ECDS*, we pass the set of writes that will (tentatively) be applied to the local store as T (e.g., attempt Algorithm 1 without using T given writes x_0, x_1, y_0, and y_1 such that $y_0 \rightarrow x_0$ and $x_1 \rightarrow y_1$ with an *ECDS* containing only x_0 and y_1; for brevity, we omit a full walkthrough, but the algorithm—without T— will infinitely recurse). If the resolver finds a set of writes that form a cover along with the existing local store then the set is applied in causal order (or, if possible, atomically) to the local store, taking care to only writes that will overwrite existing versions ($merge(old, new) = new$).

The above implementation is a faithful adaptation of prior causally consistent implementations. Shims perform dependency checking asynchronously, resulting in fast read operations that do not violate safety. However, readers may experience poor visibility, or time until writes are applied. For example, the first read from a key will always return `null` unless the resolver knows which keys to check in advance. For workloads with good temporal locality or a restricted key set, this is potentially acceptable. If we are willing to synchronously wait for dependency checking—a departure from prior architectures—we can instead make sure we read the most up-to-date copy possible given the underlying *ECDS* state.

Read path: Pessimistic Algorithm 2 is an implementation of *pessimistic* causal dependency checking. Writes proceed as before, but, for reads, shims attempt to read the latest value from the *ECDS* and cover it. This requires synchronously testing the value's dependencies against the local store and, if the dependency checks fail, attempting to also read the dependencies themselves from the *ECDS*. Whereas in the traditional Algorithm 1, reads proceeded without any checks, here we move the checking to the critical path: in a traditional architecture, this is equivalent to having readers check for unapplied but resolvable writes. The trade-off here—which we have not observed in prior literature—is between staleness of values (liveness) and read operation speed. There are a variety of policies one can use to tune this trade-off (e.g., limit the number of synchronous *ECDS* reads to some constant number), but, for the purposes of this paper, we focus on the two extremes: bolt-on causal consistency, with all local reads, and pessimistic bolt-on causal consistency, in which, when reading from the *ECDS*, shims synchronously chase the entire dependency chain.

Metadata To compute causal covers, we need to know the dependencies for each write and to be able to compare them. In the event of overwritten histories, we must store the dependency set with every write. We do not need to store the entire history but the expected size of the dependency set is the number unique of keys in the causal history (*not* the number of unique versions or writes). Each write in the dependency set requires metadata about the writing processes, which can be represented by a vector clock (requiring space proportional to the number of writers to the write's key in the history). For example, if shim P_1 writes x_1 and shim P_2 writes y_2 with $x_1 \rightarrow y_2$, x_1 will be have an empty dependency set and a vector of $\langle P_1 : 1 \rangle$, while y_1 will have a dependency set of $\{x : \langle P_1 : 1 \rangle\}$ and a vector of $\langle P_2 : 2 \rangle$ (where 1 and 2 are a logical clocks representing the first action of P_1 and second action of P_2). A third process P_3 writing y_3 such that $y_2 \rightarrow y_3$ would write y_3 with a dependency set of $\{x : \langle P_1 : 1 \rangle\}$ and a vector of $\langle P_2 : 2, P_3 : 3 \rangle$. On a `put`, the shim writes each value along with the metadata to the *ECDS* in a serialized blob.

The metadata required for bolt-on causal consistency is a significant cost of our bolt-on solution. Our assumptions about underly-

Algorithm 1 Bolt-on causal consistency.

a local store L is a set of writes, where each write is a triple $[key, value, deps = \{\text{dependent writes}\}]$

```
procedure PUT_shim(key k, val v, after, local store L, ECDS E)
    v_serialized := serialize(value, deps=after)
    E.put(k, v_serialized); L.put(k, v_serialized)
    return

procedure GET_shim(key k, local store L, set to_check)
    to_check.add(k)
    return L.get(k)

procedure RESOLVE_async(set to_check, local store L, ECDS E)
    loop forever
        for key k ∈ to_check do
            // T holds writes that will complete the cut
            e_val := E.get(k); T := {e_val}
            if is_covered(e_val, T, L, E) then
                // L should only apply newer writes
                L.put_all(T)
                to_check.remove({w.key | w ∈ T})

procedure IS_COVERED(write w, set T, local store L, ECDS E)
    for dep ∈ w.deps do
        // already applied a suitable write to L
        // or already covered a suitable write to dep.key?
        if (L.get(dep.key) != null and L.get(dep.key) ↛ dep) or
          (∃ T_dep ∈ T s.t. dep.key = T_dep.key, T_dep ↛ dep) then
            continue
        // read from the ECDS and try to cover response
        E_dep := E.get(dep.key)
        if E_dep != null and E_dep ↛ dep then
            T.add(E_dep)
            if IS_COVERED(E_dep, T, L, E) then
                continue
        return False
    return True
```

Algorithm 2 Pessimistic bolt-on causal consistency.

```
procedure PUT_shim(key k, val v, after, local store L, ECDS E)
    [same as Algorithm 1]
procedure GET_shim(key k, local store L, ECDS E)
    e_val := E.get(k); T := {e_val}
    if IS_COVERED(e_val, T, L, E) then
        L.put_all(T)
    return L.get(k)
procedure IS_COVERED(write w, set T, local store L, ECDS E)
    [same as Algorithm 1]
```

ing *ECDS* behavior require shims to write a possibly large amount of metadata with every write—proportional to the number of keys in the causal history, along with vector clocks (typically of size 1) for each. A shim could compress this metadata by, say, storing a single vector—at the expense of knowing which keys to "poll" on for new writes, which is likely unrealistic. Devising a more efficient metadata representation that does not suffer these penalties is an open problem. While these metadata requirements are prohibitive for full potential causality, they are modest for explicit causality, which has typical histories on the order of single-digit to tens and, maximally, several hundreds of updates [6]. As we demonstrate in Section 6, the overheads for explicit causality metadata may be acceptable for many modern services.

5.3 Additional Concerns

In this section, we briefly describe the properties of our solution, how to perform transactions, maintain the local store, configure large-scale deployments, and support mixed consistency models.

Availability, Durability, and Replication Bolt-on and pessimistic bolt-on causal consistency are always available. Each shim client can always safely read from its respective local store, which always contains a causal cut, preserving safety. As long as the *ECDS* is available (and it should be, by contract), all writes will be made durable by passing them to the underlying storage system. If a shim crashes, its client will crash as well (Section 3.2), and so losing shim state is safe. The *ECDS* handles all propagation, and shims need not (and do not) communicate to ensure safety or liveness.

In the event that a shim is partitioned from the *ECDS*, it can serve reads and writes (albeit without durability) and can connect to any non-failing *ECDS* replica while remaining available. This is a consequence of the fact that a causal cut (and, by construction, the local store) contains all writes that are necessary to maintain availability. However, as in standard causally consistent systems, writes that did not propagate between *ECDS* replicas before a partition occurred will not become visible until the partition heals.

Transactions Recent work introduces the notion of causal read-only and write-only transactions [34, 35]. A bolt-on shim can provide these transactions with high availability with minor modifications. To perform a read-only "get transaction," and return a causally consistent set of write, the shim needs to read a causal cut across the specified keys from the local store (note that the causal cut definition again simplifies the semantics of the operation). This is easily accomplished by employing a local store supporting "snapshot" reads at an equivalent of snapshot isolation, repeatable read isolation, or stronger [3]. Similarly, causal write-only transactions can be accomplished via transactionally applying groups of related updates to the local store [53]. While we do not further consider it, we believe a bolt-on architecture is well-suited to general-purpose causally consistent, transactionally atomic read-write transactions as in Adya's *PL-2L* [3] and, more recently, in the Swift system [53] (Section 8).

Store Pruning In both algorithms, a write can be removed from the local store if desired once the readers accessing it no longer require the write and its removal does not impact the cut invariant (i.e., many writes may need to be removed at once). This can be done using sessionization [34] or via language-centric dependency checking. However, keeping writes in the local store for as long as possible can reduce latency in pessimistic bolt-on causality while improving the freshness of results returned by bolt-on causality.

Clustered Architecture We have considered a model where shims and shim clients are co-located. However, the local store can act as a cache, and there is significant benefit to consolidation. As an alternative architecture, we can consider using a fault-tolerant shared local store (e.g., Memcached) as the local store with a secondary set of resolver processes, ferrying writes from the *ECDS*. In a multi-datacenter deployment, this allows writes to occur locally, without cross-site synchronization, while guaranteeing causal consistency.

Mixed Models Our algorithms allow mixed consistency guarantees. Consider the case where one *ECDS* user, *C*, wants causal consistency and another, *E*, wants eventual consistency. If *C* wants to observe causality generated by *E*'s actions, *E* must use the shim `put after` parameter for writes, attaching her dependencies. *E* can read directly from the *ECDS* and, aside from placing the relevant dependencies into the `after` call, will not use any of the metadata she reads. The main cost to *E* is increased *ECDS* write sizes. If *C* does not need to observe *E*'s causal relationships, *E* can use the *ECDS* directly, without attaching metadata.

6 Evaluation

In this section, we evaluate the performance of bolt-on causal consistency using a shim implementation and a production-ready eventually consistent data store. We find that providing explicit causality for several online service traces easily scales to tens of thousands of operations per second. The local read property of causal consistency allows bolt-on causal consistency to outperform eventual consistency, while pessimistic bolt-on causal consistency is often within 25% to 50% of eventual consistency's peak throughput. Using public cloud infrastructure and production-ready *ECDS* (Section 6.1), we perform single and multi-node benchmarking (Sections 6.3, 6.4) as well a full wide-area system evaluation (Section 6.5), all using real-world causal traces (Section 6.2).

6.1 Implementation and Setup

We implemented a causal and pessimistic causal consistency shims that expose `get` and `put` operations as described in Section 3. Our implementation is in Java and we use Google's Protocol Buffers (v.2.4.1) and snappy-java (v.1.0.5) for serialization and compression. From a software engineering perspective, the bolt-on architecture proved lightweight: our shim library is comprised of fewer than 2,000 lines of Java, and the core algorithms and version resolution require only 322 lines of code.

We evaluate the system with the Yahoo! Cloud Serving Benchmark [18]. Our shim interposes between YCSB and an *ECDS*, and we modified YCSB to use the shim's `after` API parameter but otherwise did not make modifications to the core YCSB code. By default, we use `workloada`, a mix of 50% read and 50% write operations and increase the number of records from the default $1,000$ to $100,000$. YCSB uses 20-byte keys, which adversely affects metadata overheads. All threads on a server share a shim and each thread writes explicit causality chains of varying lengths (Section 6.2).

For an *ECDS*, we use Cassandra 1.1.6, a widely deployed NoSQL "column family" store with replication patterned on Amazon's Dynamo [22]. From a bolt-on perspective, Cassandra represents the most general *ECDS* we have come across (Section 7): it does not support concurrent update detection, have conditional operations, or provide any stability guarantees. For concurrent writes, Cassandra uses a last-writer-wins update policy based on client-generated timestamps. We run Cassandra on Amazon EC2 `m1.xlarge` instances and use ephemeral RAID0 storage as recommended by commercial support documentation [30]. We do not modify Cassandra and only change standard parameters like the cluster topology, location of the data directories, and JVM heap settings. In each experiment, we vary the number of YCSB client threads and report

the average throughput and latency for five sixty second trials. For each trial, we destroy the Cassandra database and repopulate it with new data and causal chains via an independent shim process.

6.2 Causality Datasets

To drive our evaluation, we use publicly available datasets profiling explicit causality relationships on four different online services:

Twitter Twitter is a microblogging service with over 500 million users. We use a corpus of 936,236 conversations on Twitter comprising 4,937,001 Tweets collected between Feb. and July 2011 [43]. Tweets that are not part of a conversation—72% in one study [51]—are omitted. This results in a substantially higher proportion of causal chains than we would experience in practice.

Flickr Flickr is an image and video sharing website of over 51 million members. We use a corpus of 11,207,456 posts and 51,786,515 comments collected from February to May 2007 [15].

Metafilter Metafilter is an active community weblog with a range of topics and over 59,000 users. We use a corpus of Metafilter posts from July 1999 to October 2012 containing 362,584 posts and 8,749,130 comments [1].

TUAW The Unofficial Apple Weblog (TUAW) is a blog devoted to Apple products. We use a corpus of 10,485 blog posts and 100,192 comments collected between January 2004 and February 2007 [52].

We primarily focus on the comment lengths for these datasets (for Twitter, conversation lengths) and use the commenting reply relationship as the basis of our explicit causality chains. This is for two reasons, both of which are pessimistic for bolt-on causality. First, linear chains perform worse under bolt-on causality—a bushy causality graph can be checked in parallel, while sequentially traversing a linear graph limits concurrency. To obtain an approximate upper bound on bushy graph behavior, one can condense the graph into a linear chain. Second, we do not reproduce the comment sizes and intentionally limit value sizes in our experiments to 1 byte to fully expose the effects of our metadata overheads.

We show the distribution of comment lengths in Table 1. The median length across chains is short (single digit for three of our four distributions), but the tail is long; the 99th percentile is between 6 and 10 times the mean length and 10 and 13 times the median. This means that for many conversations, metadata sizes will be limited, but long conversations may be problematic.

Write Throughput In our evaluation, we generate chains as quickly as possible. In practice, write throughput requirements are limited. For example, as of June 2012, Twitter reports receiving an average of 4,630 Tweets per second [24] and reached a one-second peak of 15,107 Tweets per second during the 2012 US presidential election [42]. Facebook reported an *average* of 31,250 Likes and Comments per second during October to December 2011 [23]—almost an order of magnitude difference. In contrast, in our Metafilter dataset, we observe a peak of 27 comments per second, averaging 0.021 comments per second over a period of more than 13 years. These data points do not account for read traffic, which overwhelms write traffic for many modern services. However, we aim for higher throughput in our evaluation.

6.3 Single-node Microbenchmarks

We begin by micro-benchmarking shim performance on a single Cassandra instance and single shim server. Figure 3 shows pareto frontier [28] latency-throughput curves for bolt-on and pessimistic bolt-on shim configurations.

Under standard bolt-on causal consistency (hereafter, `bolt-on`), peak throughput is between 16,329 and 29,214 operations per second. Read latency is negligible (several microseconds), and the system ultimately bottlenecks on Cassandra. The Flickr, Twitter, and TUAW datasets all behave similarly, but Metafilter is an outlier, due in large part to its increased metadata requirements. With stock YCSB on Cassandra, increasing write sizes from 1*B* to 1*KB* decreases throughput by 12.2% (2035 ops/s), but increasing write sizes from 1*KB* to 4*KB* incurs a 50% (8148 ops/s) peak throughput penalty. Accordingly, Metafilter's long-tailed metadata sizes (Table 2) are more expensive than those of the other traces, which are often smaller than 1*KB*. Additionally, serialization and write resolution overheads grow with chain length, incurring higher shim CPU load. As expected, we find small median write latencies but exaggerated latencies at high percentiles. Nonetheless, `bolt-on` outperforms stock Cassandra ("eventual") for all data sets except Metafilter (albeit at an increased penalty to visibility, as below). This is because causal consistency does not require *ECDS* reads, reaffirming the original motivation of picking causal consistency for its increased availability and throughput.

Dataset	Mean	Median	75th	95th	99th	Max
Twitter [43]	5.3	3	5	15	40	968
Flickr [15]	4.6	2	5	19	44	3948
Metafilter [1]	24.1	13	27	78	170	5817
TUAW [52]	9.6	6	12	31	62	866

Table 1: Comment length distribution for datasets. Most chains are short, but each distribution is long-tailed.

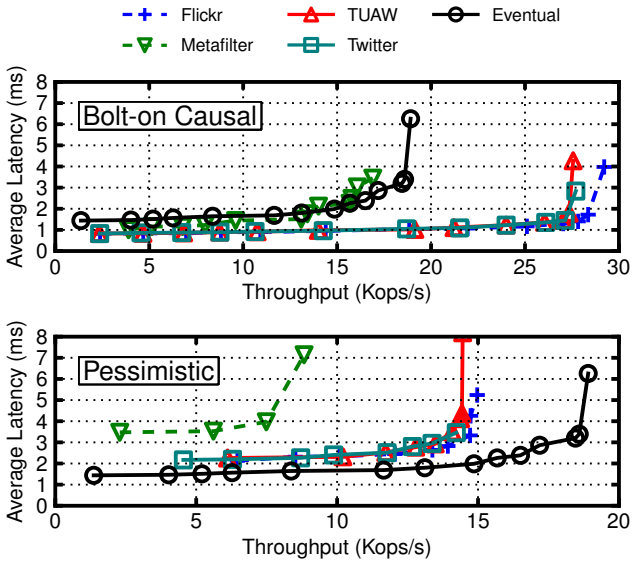


Figure 3: Latency-throughput for single node microbenchmarks.

With pessimistic bolt-on causality (hereafter, `pessimistic`), we observe peak throughput between 8827 and 14970 operations per second. For our three shorter datasets, peak throughput is within 22% of eventual consistency while Metafilter falls short by 51%. The difference between standard `bolt-on` and `pessimistic` shim performance is due to the `pessimistic` shim following the transitive dependency chains on the read fast path. For example, for Metafilter's peak throughput, pessimistic performs an average of 1.32 *ECDS* reads for every one front-end read—as opposed to zero in bolt-on causal consistency. However, the `pessimistic` strategy results in better visibility: 18% fewer `null` responses. By volume, the mean write depth is much higher than the mean chain depth (Table 2); this is because most writes are part of long chains. By cutting this tail, we could improve performance; we can consider a model in which causality is only guaranteed for the majority of

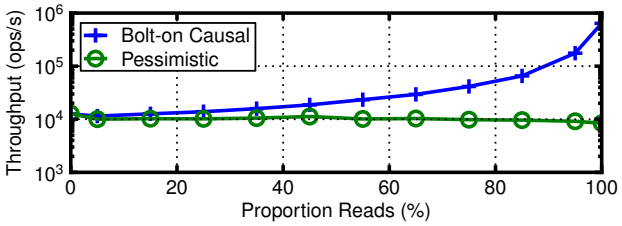


Figure 4: Peak throughput versus read proportion for Twitter.

Dataset	Mean	Median	95th	99th
Flickr	386.5 (13.0)	201 (5)	1153 (44)	2447 (100)
Metafilter	1481.6 (60.0)	525 (18)	4272 (180)	19735 (870)
Tuaw	423.8 (14.4)	275 (8)	1259 (49)	2438 (100)
Twitter	429.2 (15.0)	169 (4)	1468 (58)	5407 (230)

Table 2: Serialized data size in bytes and chain length in messages (in parentheses) for bolt-on shim and workloads.

posts—which are short—subsequently freeing resources consumed by resolving long chains.

The proportion of reads and writes affects peak throughput (Figure 4). Performing all reads under the Twitter dataset, `bolt-on` achieves over 640,000 gets/s because it is effectively reading values from local memory, yet the `pessimistic` strategy achieves fewer than 8,500 ops/s. With entirely writes, `bolt-on` and `pessimistic` are equivalent, achieving approximately 12,500 writes per second. With a 95% mix of reads, `bolt-on` achieves 175,000 operations per second (roughly 8.8K writes/s). This result is unsurprising—`bolt-on` allows purely local reads at the expense of visibility; by treating the local store as a fast-path cache, reads can proceed without being affected by *ECDS* latencies.

We observe that, in general, the efficiency of the `bolt-on` strategy depends on metadata overheads. For many datasets, overhead is less than 500B but, for datasets like Metafilter, can exceed 19KB. Thus, typical metadata overheads are inexpensive, but, for long histories, the cost of causality must be weighed against end-user benefits (e.g., is it worth storing 200 comments' worth of metadata, or just the last 10?) Equally importantly, YCSB has good spatial and temporal locality, meaning that each shim's local store is likely to contain many recent writes and that dependency resolution is inexpensive. While we have pessimistically adjusted the YCSB defaults, in practice, and for different workloads, we expect that hybrid strategies (e.g., bounding checking times) will be necessary to mitigate the effect of worsened locality (at the expense of increased data staleness).

6.4 Scale-out

We next consider the scalability of bolt-on causality. Both primary architectural components of our implementation—our shim and Cassandra—are designed as shared-nothing systems. As expected, the shim implementation scales linearly with the number of replicas (Figure 5). For `bolt-on` causality, throughput scales from 35K ops/s to 50K ops/s moving from 3 to 5 replicas and doubles moving from 5 to 10 replicas. For `pessimistic` bolt-on causality, throughput also scales linearly. With 10 replicas, we achieve 140K operations per second. Although we do not present a full evaluation, `bolt-on` achieves slightly over 1M operations per second under bolt-on causal consistency with 100 Cassandra instances and 100 shim processes.

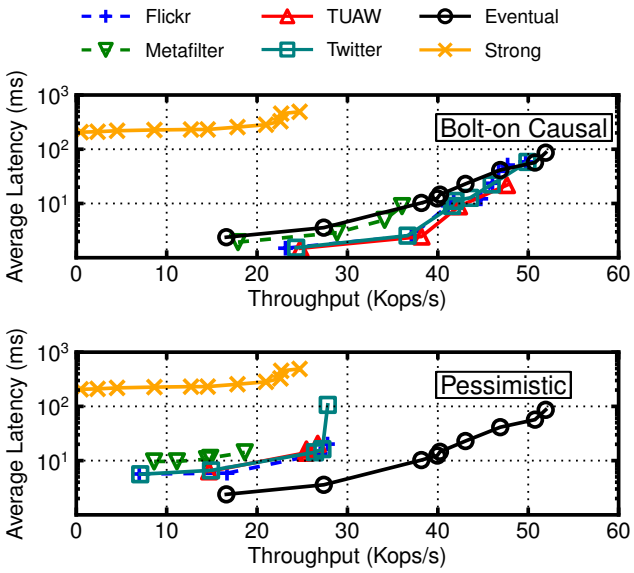


Figure 6: Latency-throughput for WAN deployment.

6.5 WAN Deployment

As an end-to-end demonstration of bolt-on causality, we deployed our shim across two datacenters. We place five shim and five Cassandra servers in each datacenter (`us-east` and `us-west-1`) for a total of ten shims and ten Cassandra instances with a replication factor of 2 per datacenter, ensuring 4 replicas per key (3-server-fault tolerance with 1-server-fault tolerance per datacenter). Eventually consistent and shim *ECDS* requests complete after contacting a single replica, while Cassandra's strong "read your writes" quorum configuration (which is weaker than causal consistency, providing regular register semantics) requires contacting a minimum of 3 replicas per request. As we show in Figure 6, `bolt-on` causal consistency achieves around 45,000 ops/s for all datasets but Metafilter, which achieves 35,000 ops/s, all without incurring WAN latencies. Eventual consistency achieves over 50,000 ops/s at an average latency of 139 ms/op, while "strong" (regular register) consistency achieves a 24,703 operations at an average latency of 490 ms/op. Employing strong consistency results in both increased latency and decreased throughput, while bolt-on causal consistency do not incur substantial overhead in either category.

7 Design Space

Our approach to bolt-on causal consistency has been pragmatically motivated: we considered a restrictive *ECDS* model that approximates the "bare minimum" of guarantees provided by eventually consistent stores. This allows us to use a variety of *ECDS* implementations. However, eventually consistent stores vary in their guarantees, and understanding the implications of API design on layered approaches to consistency is instructive for future systems architectures. In this section, we discuss *ECDS* API modifications that affect the efficiency of bolt-on causality. We found that most *ECDS*s except for Cassandra and Amazon's S3 supported at least one of these optimizations (Table 3).

7.1 No Overwrites

Our initial *ECDS* model chooses a single value for each key, so a shim may encounter overwritten histories and lose metadata. If however, we disallow overwrites, then implementing bolt-on causal

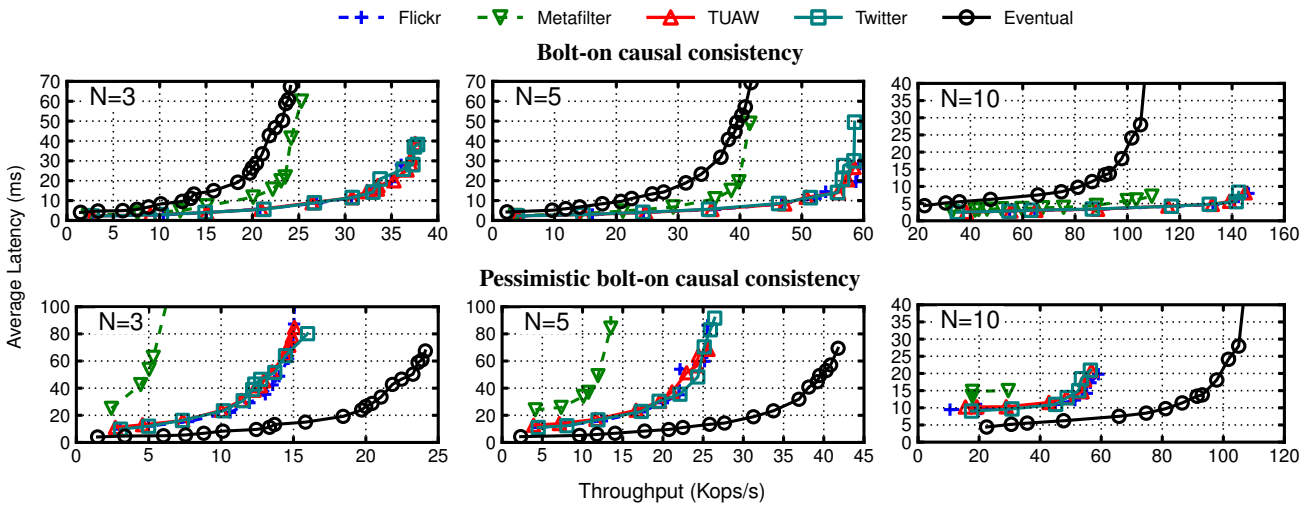


Figure 5: Bolt-on and pessimistic bolt-on latency-throughput for varying cluster sizes. The shared-nothing architectures scales linearly.

Data Store	Concurrent Versions	Cond. Update	Stable Callback
Hosted			
Amazon DynamoDB	N	Y	N
Amazon S3	N	N	N
Amazon SimpleDB	N	Y	N
Amazon Dynamo [22]	Y	N	N
Cloudant Data Layer	Y	N	N
Google App Engine	N	Y	N
Self-Hosted			
Apache Cassandra [30]	N	N	N
Apache CouchDB	Y	N	N
Basho Riak	Y	N	N
LinkedIn Voldemort	Y	N	N
MongoDB	N	Y	N
Yahoo! PNUTS [17]	N	Y	N
Effect of *ECDS* API Change			
Decreases metadata	Y	Y	Y
Decreases *ECDS* reads	N	N	Y

Table 3: Additional primitives provided by real-world *ECDS* systems (both purely eventually consistent and allowing eventually consistent reads) as of November 2012.

consistency becomes considerably easier. If there is only one version of each key, overwrites are not a problem: a shim would only need to store the *nearest dependencies* in each write, reducing the storage overhead [34, 35]. In addition, without overwrites, there cannot be concurrent or later updates to a given key, so the definition of a causal cut becomes much simpler: we simply have to check that each key in the transitive closure is in the local store. If the *ECDS* provides monotonic reads session guarantees (i.e., time never goes backwards with respect to a single key) [48], then we need not store the value of the key in the local store. However, for the many modern applications that require mutation, this solution may have limited practical applicability.

7.2 Controlling Concurrent Overwrites

Given that we cannot disallow overwrites for all applications, we can attempt to reduce their severity. There are at least two mechanisms that allow us to control overwrites in an eventually consistent system. First, eventually consistent systems can track concurrent operations across replicas and present all concurrent updates to the client, who then decides which to keep [22, 25]; a bolt-on system could avoid overwritten histories by appropriately merging concurrent updates. Alternatively, if the *ECDS* provides a test-and-set operation, shims can trivially control overwrites without the need for merging as above. Linearizable test-and-set requires a strongly consistent operation, but we can consider an *ECDS* allowing both linearizable writes and eventually consistent reads (e.g., PNUTS [17], DynamoDB). The cost of these mechanisms is that shims may have to read more data if the *ECDS* returns concurrent writes or retry writes in the event of failed test-and-set operations.

7.3 Callback on Convergence

Once a version is visible to all processes, it is *stable* and can be omitted from the metadata history [16]; this helps enable the best-known causally consistent architectures to perform well in the absence of partitions [34, 35]. A bolt-on solution can leverage a hypothetical callback on convergence from the *ECDS*: shims can instead buffer new writes until their dependencies have reached all *ECDS* replicas. This way, the *ECDS* will always contain a causal cut and the shim can treat the *ECDS* as its local store, eliminating all metadata and dependency checks. However, shims may need to buffer their own writes indefinitely; to ensure durability, each shim can queue its pending-but-unstable updates in a private *ECDS* data item. Moreover, to ensure availability in the event of *ECDS* disconnections, the shim may need to cache some of its reads. While we are not aware of these callbacks in any production ready stores, we believe bolt-on layers would greatly benefit from their inclusion at only a small cost to *ECDS* complexity. Alternatively, shims can leverage consistency prediction or monitoring systems to provide estimates for write stability (e.g., Cassandra 1.2 recently integrated support for PBS, which provides expected bounds on write visibility to readers [8]). This allows probabilistic causal consistency guarantees, which may be sufficient for some applications.

8 Related Work

Our work is motivated by and related to a wide range of work on weak consistency and architectural separation in database systems.

Weak Consistency The large number of distributed data stores and consistency models reflects the complex trade-offs between both consistency and availability [20, 27] and consistency and performance [2, 8]. In the presence of partitions, causal consistency is

one of the strongest available models [36], but there are many others such as session guarantees [48]. In this paper, we focus specifically on causally and eventually consistent stores.

Causal Consistency Lamport [31] introduced the concept of causal ordering between events in distributed systems, which Ahamad et al. [4] generalized to causal memory—a consistency model for reads and writes that respects causality. There are many causally consistent systems in the literature, including Bayou [41], COPS [34], PRACTI [9], Swift [53], and lazy replication [29]. Many of these systems provide convergent causal consistency [36], also termed causal+ consistency [34]. These systems often adopt variants of the causal memory algorithm [4] outlined in Section 4, using log shipping [9, 41], explicit messaging [45], or dependency tracking [29] but assume reliable delivery and therefore do not handle the problem of overwritten histories.

Recent work has renewed interest in causally consistent systems. Mahajan et al. have provided a proof that no model stronger than real-time causal consistency is achievable in a one-way convergent, always available system [36], highlighting it as a particularly noteworthy model. Concurrently, Lloyd et al. showed that causal consistency with read-only and write-only transactions can be achieved are achievable with minimal latency across a wide-area network and with high availability across datacenters [34, 35] Their COPS and Eiger systems adopt a variant of the log-shipping approach from Section 4.1 where the local store is sharded across a set of fault-tolerant, linearizable servers in each datacenter. Finally, Swift [53] is the first system we are aware of that provides highly available causal read/write transactions similar to those we discussed in Section 5.3. Swift employs client-side caching and "scout" processes to update client caches and maintain per-datacenter vector clock metadata. In this work, we attempt to bridge the gap between the literature and current practice by identifying how to achieve these semantics in existing systems.

Layered Database Architectures Our bolt-on architecture is influenced by the end-to-end principle for systems design [44], which dictates that application functionality should be located at system endpoints—in our case, we separate the consistency safety properties from the mechanics of replication. Several systems such as Deuteronomy [32], G-Store [19], and Microsoft SQL Azure [14] and related research in database middleware [40] and "component databases" [26] similarly separate transaction management from distributed storage. In providing strongly consistent semantics, these solutions often rely on distributed atomic operations [14, 32, 40] and/or stronger storage-level guarantees (e.g., fine-grained partitioning [19]). This body of work is a useful, *strongly consistent* parallel to our approach here.

Two recent projects build weak semantics above eventually consistent infrastructure. Brantner et al. provide a range of data consistency properties (like atomic transactions and read-your-writes guarantees) with varying availability properties by leveraging infrastructure like Amazon S3 and atomic queues [12]. Bermbach et al. have concurrently studied middleware for cache-based monotonic reads and read-your-writes guarantees above eventually consistent stores [10]. Their focus on single-key session guarantees forms an interesting contrast with our multi-key causal consistency: overwritten histories are not problematic for these single-key guarantees, which are achieved with standard dependency tracking metadata. Our focus in this paper is on providing convergent causal consistency with high availability.

Our shim design leverages many of the guarantees provided by the *ECDS* such as convergence and write propagation. The idea of assembling a stronger consistency model out of weaker components is partially due to Brzezinski et al. [13], who prove that the composition of many session guarantees yields causal consistency. Assembling causal consistency guarantees from eventually consistent infrastructure yields a new set of challenges—here, primarily the problem of overwritten histories.

9 Conclusions

In this paper, we upgraded production-ready eventually consistent data stores to provide convergent causal consistency, one of the strongest consistency models achievable in the presence of partitions. We separated concerns of replication, fault-tolerance, and availability from the safety properties of consistency by applying a thin shim layer on top of unmodified, existing stores. The problem of overwritten histories made providing convergent causality through a shim difficult: the underlying store's register semantics allowed the loss of causal dependency information, resulting in unavailability for existing algorithms. To solve this problem, we introduced causal cuts, an extension of consistent cuts to causal memories. While maintaining causal cuts for complete potential causality is prohibitively expensive, a large class of application-level causal dependencies—explicit causality—is well suited for this model. Our bolt-on shim implementation outperforms eventual consistency in the case where local reads are acceptable and otherwise achieves throughput and latency often within 20% and typically within 50% for representative explicit causality workloads.

In light of our experiences, we believe that bolt-on and layered approaches to data storage can provide a useful and modular alternative to often monolithic existing data storage infrastructure. In this work, we have taken a fairly restrictive approach, with an eye towards interoperability and broad applicability; the key challenge of overwritten histories and the metadata overheads required by our solution are artifacts of this decision. However, a less restrictive set of assumptions (as is reasonable in future data storage systems) offers the possibility of more harmonious cross-layer coordination. In particular, we believe that the opportunity to revisit the data storage interface and its interaction with higher layers of data management functionality is especially ripe.

Acknowledgments The authors would like to thank Ganesh Ananthanarayanan, Neil Conway, Joseph Gonzalez, Adam Oliner, Aurojit Panda, Josh Rosen, Colin Scott, Evan Sparks, Shivaram Venkataraman, Patrick Wendell, Matei Zaharia, and especially Alan Fekete for their constructive feedback on earlier revisions of this work.

This research is supported in part by National Science Foundation grants CCF-1139158, CNS-0722077, IIS-0713661, IIS-0803690, and IIS-0917349, DARPA awards FA8650-11-C-7136 and FA8750-12-2-0331, Air Force OSR Grant FA9550-08-1-0352, the NSF GRFP under Grant DGE-1106400. and gifts from Amazon Web Services, Google, SAP, Blue Goji, Cisco, Clearstory Data, Cloudera, EMC, Ericsson, Facebook, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, NTT Multimedia Communications Laboratories, Oracle, Quanta, Samsung, Splunk, VMware and Yahoo!.

10 References

[1] Metafilter Infodump. http://stuff.metafilter.com/infodump/. Combination of all available comment datasets: `mefi`, `askme`, `meta`, `music`. User count from `usernames`.

[2] D. J. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2):37–42, 2012.

[3] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[4] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. Hutto.

Causal memory: Definitions, implementation and programming. *Distributed Computing*, 9(1), 1995.

[5] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[6] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution. In *ACM SOCC 2012*.

[7] P. Bailis and A. Ghodsi. Eventual Consistency today: Limitations, extensions, and beyond. *ACM Queue*, 11(3), 2013.

[8] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically Bounded Staleness for practical partial quorums. In *VLDB 2012*.

[9] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *NSDI 2006*.

[10] D. Bermbach, J. Kuhlenkamp, B. Derre, M. Klems, and S. Tai. A middleware guaranteeing client-centric consistency on top of eventually consistent datastores. In *IEEE IC2E 2013*.

[11] K. Birman. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. *SIGOPS Oper. Syst. Rev.*, 28(1):11–21, Jan. 1994.

[12] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *SIGMOD 2008*.

[13] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. From session causality to causal consistency. In *PDP 2004*.

[14] D. G. Campbell, G. Kakivaya, and N. Ellis. Extreme scale with full SQL language support in Microsoft SQL Azure. In *SIGMOD 2010*.

[15] M. Cha, A. Mislove, and K. P. Gummadi. A Measurement-driven Analysis of Information Propagation in the Flickr Social Network. In *WWW 2009*.

[16] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *SOSP 1993*.

[17] 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 *VLDB 2008*.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *ACM SOCC 2010*.

[19] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *ACM SOCC 2010*.

[20] S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, 1985.

[21] J. Dean. Designs, lessons, and advice from building large distributed systems. Keynote from LADIS 2009.

[22] 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 2007*.

[23] Facebook, Inc. SEC Form S-1, February 2012.

[24] D. Farber. Twitter hits 400 million tweets per day, mostly mobile. CNET, 2012. http://cnet.co/KHlg8q.

[25] A. Feinberg. Project Voldemort: Reliable distributed storage. In *ICDE 2011*.

[26] A. Geppert and K. Dittrich. Component database systems: Introduction, foundations, and overview, in *Component Database Systems*, 2001.

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

[28] M. Klems. cassandra-user mailing list: Benchmarking Cassandra with YCSB, February 2011. http://bit.ly/15iQfBD.

[29] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, Nov. 1992.

[30] A. Lakshman and P. Malik. Cassandra - a decentralized structured storage system. In *LADIS 2008*. Configuration: http://cassandra.apache.org (2013).

[31] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[32] J. Levandoski, D. Lomet, M. Mokbel, and K. Zhao. Deuteronomy: Transaction support for cloud data. In *CIDR 2011*.

[33] G. Linden. Make data useful. https://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-29.ppt, 2006.

[34] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP 2011*.

[35] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI 2013*.

[36] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, convergence. Technical Report TR-11-22, Computer Science Department, UT Austin, May 2011.

[37] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

[38] J. S. Mill. *A System of Logic, Ratiocinative and Inductive, Being a Connected View of the Principles of Evidence, and the Methods of Scientific Investigation*, volume 1. John W. Parker, West Strand, London, 1843.

[39] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng.*, 9(3):240–247, May 1983.

[40] M. Patiño-Martinez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Consistent database replication at the middleware level. *ACM TOCS*, 23(4):375–423, 2005.

[41] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP 1997*.

[42] M. Rawashdeh. Bolstering our infrastructure. http://engineering.twitter.com/2012/11/bolstering-our-infrastructure.html (2013).

[43] A. Ritter, C. Cherry, and B. Dolan. Unsupervised modeling of Twitter conversations. In *HLT 2010*.

[44] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, Nov. 1984.

[45] A. Schiper, K. Birman, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, Aug. 1991.

[46] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7:149–174, 1994.

[47] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP 2011*.

[48] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *PDIS 1994*.

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

[50] W. Vogels. Choosing consistency. http://www.allthingsdistributed.com/2010/02/strong_consistency_simpledb.html, 2010.

[51] S. Ye and S. F. Wu. Measuring message propagation and social influence on Twitter.com. In *SocInfo 2010*.

[52] R. Zafarani and H. Liu. Social computing data repository at ASU, 2009. TUAW dataset.

[53] M. Zawirski, A. Bieniusa, V. Balegas, N. Preguica, S. Duarte, M. Shapiro, and C. Baquero. Geo-replication all the way to the edge. 2013. Personal communication and draft under submission. http://asc.di.fct.unl.pt/~nmp/swiftcomp/swiftcloud.html.