The Potential Dangers of Causal Consistency and an Explicit Solution

Peter Bailis[†], Alan Fekete[◇], Ali Ghodsi[†,‡], Joseph M. Hellerstein[†], Ion Stoica[†]

[†] University of California, Berkeley [◇] University of Sydney [‡] KTH/Royal Institute of Technology

{pbailis, alig, hellerstein, stoica}@cs.berkeley.edu, alan.fekete@sydney.edu.au

ABSTRACT

Causal consistency is the strongest consistency model that is available in the presence of partitions and provides useful semantics for human-facing distributed services. Here, we expose its serious and inherent scalability limitations due to write propagation requirements and traditional dependency tracking mechanisms. As an alternative to classic potential causality, we advocate the use of explicit causality, or application-defined happens-before relations. Explicit causality, a subset of potential causality, tracks only relevant dependencies and reduces several of the potential dangers of causal consistency.

CATEGORIES AND SUBJECT DESCRIPTORS

H.2.4 [**Systems**]: Distributed Databases

GENERAL TERMS

Design, Algorithms

KEYWORDS

causality, scalability, explicit causality, data dependencies, weak consistency, semantic knowledge, convergence

1 INTRODUCTION

Replicating distributed services requires making hard trade-offs between multiple competing factors, which are exacerbated in wide-area settings. Returning a "highly consistent" query response incurs high performance overheads due to expensive coordination [2] and may be unachievable in the presence of network partitions and failures [10]. A growing crowd of distribution mechanisms and designs provide a spectrum of data consistency guarantees, each making different trade-offs.

Recent work has identified causal consistency as a noteworthy consistency model [19, 20]. First, it is the "strongest achievable" model that is available in the presence of network partitions. This is an attractive property in multi-datacenter settings, as a causally consistent distributed data store can respond to queries locally without incurring long round-trip latencies to other remote datacenters. This also enables datacenters to safely serve requests in the presence of network partitions and failures. Second, causal consistency provides useful semantics, disallowing many scenarios that contradict natural human expectations of system behavior. Causal consistency guarantees that effects are observed only after their causes: participants will not see data unless its dependencies are also seen. For modern web services, this means message replies will only be seen along with their parents, commenting structures will be preserved, and users' privacy settings will work correctly in concealing scandalous Spring Break photos from their families and employers [6, 19].

SOCC'12, October 14-17, 2012, San Jose, CA USA
Copyright 2012 ACM 978-1-4503-1761-0/12/10 ...$15.00.

1.1 Lurking Dangers, Difficult Trade-offs

With useful semantics, low latency, partition tolerance, and, recently, a demonstrably efficient architecture [19], causal consistency appears an ideal model for the future of wide-area distributed data stores. However, implementations of causal consistency face serious scalability challenges.

In this paper, we identify a critical trade-off between write throughput and visibility latency, or the amount of time that each write is hidden from readers due to missing dependencies (§3.1). This trade-off is influenced by two factors: the number of datacenters and the rate at which each datacenter can check dependencies and apply new writes. First, scaling the number of datacenters does not improve throughput: causal dependencies are not cleanly partitionable and must be sent to all datacenters, limiting sustainable throughput to that of the slowest datacenter (effectively zero in the event of network partitions)—or, alternatively, requiring unbounded visibility latency. Scaling total throughput while adding more datacenters requires quadratic increases in total server capacity (§3.2). Second, each datacenter must buffer each write until it has observed all of the write's dependencies. This phenomenon is well documented [5] but is amplified by the enormous causal dependency graphs of modern services (§3.3). Combined, these effects result in either severe peak write throughput limitations or unacceptably high visibility latency.

1.2 Better Living Through Semantic Context

While these dangers are potentially prohibitive, we can decrease their severity by considering modern application contexts. In the three decades since Lamport's seminal work defining the causal `happens-before` relation [17], both practitioners and theoreticians have almost exclusively considered the problem of *potential causality*: each new write causally depends on all writes (versions) that could have influenced it. This is a useful model for closed systems and debugging (§5) but is too general for modern real-world applications. Steve's latest Facebook comment was potentially influenced by the hundreds of status updates he had recently read, but its primary dependency is Mary's wall post—to which his comment is attached—asking if he was planning to attend her party.

Modern, human-facing services already naturally express semantic dependencies in their APIs and the data they produce; in most cases we can use these application-level relationships to explicitly define relevant dependencies instead of having the system assume all potential causality patterns (§4). This application-defined *explicit causality* [5, 16, 7] is a small subset of traditional potential causality and dramatically reduces the depth and degree of the causality graph, ameliorating scalability concerns. To quantitatively illustrate these effects, we draw on a substantial body of literature studying behavior patterns in modern Internet services (§4.1). Studies show that explicit causality graphs for these services are often in the tens of events and seldom in the hundreds or thousands of events. As an example, Twitter conversation lengths average approximately 11 Tweets [23, 29]. In contrast, if we capture the potential causality for a year's worth of Tweets, the resulting causality graph is around nine orders of magnitude larger.

Prior work by Cheriton and Skeen famously denounced the semantics and scalability of causally ordered communication systems (i.e., CATOCS) [4, 5]; in light of recent interest, we revisit their concerns in the context of causally consistent *data stores*. We embrace Cheriton and Skeen's position that a communication system is ill-suited to express or capture *data-level* dependencies and believe these dependencies are best captured at or above the storage level, where many of today's systems share data. While our causally consistent data storage provides a more natural model (§5), it presents new pitfalls (§3) and requires a different approach (§4).

Explicit causality is not a perfect solution, yet it helps mitigate many of the difficulties operators will face in providing causal consistency. Explicit causality decreases the number of dependencies per write, which increases throughput, lowers metadata overhead (particularly in the presence of partitions), and improves concurrency (§4.2, §4.4). Many dependencies are already captured by applications and present in their data models; adopting explicit causality is often transparent or is a simple extension to many applications (§4.3). Explicit causality does not solve the fundamental problems of all-to-all replication or guaranteed graceful partition tolerance but instead reduces constant factor overheads by several orders of magnitude. Given these scalability improvements and ease of use in modern applications, we believe explicit causality will be a key component in achieving causal consistency at scale under real-world conditions.

2 A PRIMER ON CAUSAL CONSISTENCY

In this section, we provide definitions and an overview of the dominant architectures for achieving causal consistency. Readers familiar with causally consistent data stores may wish to proceed to Section 3.

Informally, causal consistency guarantees that effects are observable only after their causes. Consider the following scenario on a social networking site like Facebook:

1 . Lewis posts status update *L* to his friends: "Jenny's unconscious in the hospital! The doctors think it's a coma."

2 . Shortly afterwards, Jenny regains consciousness, and Lewis edits his status update L, resulting in L^*: "Jenny's okay after all! Hooray!"

3 . Lewis's friend Mary observes L^* and posts comment M in response: "What terrific news!"

If causality is not respected, another user, Stan, could perceive effects before their causes; if Stan observes L and M but not L^*, he might think that Mary is pleased to hear of Jenny's would-be coma! If the site had respected causality, Stan could not have seen M without L^*.

2.1 Definitions

More formally, under causal consistency, the sequence of versions that each agent reads obeys a partial order called the `happens-before` relation ($\rightarrow$). If a causally consistent data store contains three versions of a given object v_1, v_2, and v_3 and $v_1 \rightarrow v_2 \rightarrow v_3$, then two subsequent reads by an agent can return v_1 then v_3 but not v_3 then v_2 or v_3 then v_1. Some versions are incomparable under the `happens-before` relation and can be safely returned as long as the partial order of operations is not violated.

The `happens-before` relation is traditionally defined according to *potential causality*, which reflects three relationships: (*i*) each agent's program order (e.g., if a single agent performs operations a and b, then $a \rightarrow b$), (*ii*) reads-from (e.g., if read b returns write a, then $a \rightarrow b$), and (*iii*) transitivity (i.e., if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$) [3, 19]. The `happens-before` relationship for a given write is called its causal *history*, and the graph of causal dependencies forms a partial order over writes. Schwarz and Mattern provide an in-depth overview of causal relationships [26]. We will revisit potential causality later on in the context of real-world applications (§3.3) and will propose an important modification to the `happens-before` relation (§4).

In this paper, we consider the problem of achieving *convergent* causal consistency (also known as causal+ consistency): in addition to the properties above, if updates cease, all agents will eventually read the same value (or set of values) of each object [19, 20]. Convergence is an important criterion because causal consistency by itself lacks any liveness guarantees. For example, without the convergence requirement, it is possible to satisfy causal consistency by never propagating writes between agents (preserving safety but resulting in undesirable semantics).

2.2 Implementing Causal Consistency

Implementations of causally consistent data stores follow a common pattern: each agent in the system maintains a set of writes—or a local store—which can be safely (locally) modified and read from. When a new (remote) write originating from another agent in the system arrives, the agent checks the new write's metadata to ensure that its causal dependencies are satisfied by the writes in its current local store. If the dependencies are satisfied, the agent *applies* the write to the local store, and, if not (because one or more dependencies are missing), the agent waits to locally apply the new write until the dependencies have themselves been applied to the local store. A write only becomes *visible* to reads once it has been applied to the local store against which the reads are performed.

Why Wait? One way to think about causal consistency is as a temporary enforcement of data model integrity, where constraints are determined for each data item by its `happens-before` relation. Consider the status update "Mary likes Greenpeace." The "likes" relationship between users and companies—in this case, between "Mary" and "Greenpeace"— could be stored as a normalized set of `user-id` to `company-id` mappings. If the status update propagates to a remote datacenter before the `company` entry for "Greenpeace," then, without causal consistency, we might end up with a dangling reference to the (empty) entry for "Greenpeace" in the `company` table. Implicitly, there is a dependency between the `company-id` field of the "likes" relation and the `company` table (similarly for the `user-id` field and a `user` table). However, the constraint is only temporarily violated: we know that the appropriate entry in the `company` table will eventually appear. We just need to wait to (locally) reveal this particular "likes" relationship.

Applications can perform this checking for themselves on each read, but it is expensive to do so. Just because a version's immediately preceding dependencies have arrived does not mean that the predecessors' predecessors have arrived. If an application implements causal consistency outside of the data store, it must check all transitive dependencies on every read. However, when implemented at the data store level, an application reading a data item knows that its dependencies are transitively satisfied; the application need not perform any checking itself.

Implementing Waiting The causal memory implementation by Ahamad et al. [3] is standard; other designs are similar [19, 24]. When performing a write, an agent first updates its local store, then sends the new write via *reliable broadcast* to all other agents, who buffer it. An agent applies a buffered write to its local data store once it has applied all of the write's causal dependencies.

Since all messages are eventually delivered by the broadcast, all agents will eventually locally apply all writes. Agents track the causal history of writes they have applied to their local stores and attach a summary of relevant history to each write. The storage format of this metadata and write propagation are orthogonal to our concerns.

In a wide-area deployment, each datacenter may act as a single causal agent [19]. A datacenter is comprised of multiple servers, but local network latencies are often sufficiently low that each cluster of servers can provide strong consistency (acting as a single, atomic—or linearizable—local store [12]). Each server within the cluster is responsible for a subset of the key space in the local store, and, when a write arrives from a remote datacenter, its dependencies are checked against the servers responsible for each respective key. We will refer to datacenters instead of agents for the remainder of this paper.

3 POTENTIAL DANGERS

Scaling causal consistency to multiple datacenters leads to a trade-off between write throughput and latency visibility, which is the time a write is delayed before it can be read (§3.1). The severity of this trade-off is determined by two independent variables: the number of datacenters and the rate of local write application. The sustainable aggregate throughput of multiple datacenters is limited to the rate at which the slowest datacenter can locally apply new writes (zero during partitions) (§3.2), while the large size of traditional, potential causality graphs in modern applications limits the rate which each data center can locally apply new writes (§3.3).

3.1 Throughput and Visibility Latency

Causally consistent systems face a trade-off between the rate at which clients generate new writes (throughput) and *visibility latency*. If a datacenter cannot apply a new write, it must wait until its dependencies have arrived. The amount of time that the update is buffered determines its visibility latency. Visibility latency is affected by both network latency and the rate at which dependency checking can be performed (apply capacity).[1] Intuitively, visibility latency and throughput are competing goals. If the aggregate (global) throughput limit across datacenters is exceeded and new versions are generated faster than they can be applied, visibility latency increases indefinitely due to the formation of unstable queues.

The effects of this trade-off are magnified by the fact that convergent causal consistency effectively requires that all datacenters locally apply all writes: the global throughput limit is limited to the minimum apply-capacity across datacenters. Potential causality for most writes forms a complex graph structure that is not cleanly partitionable, meaning replication to all datacenters is required to ensure that writes can be applied remotely. While weaker consistency models are often amenable to partial replication (i.e., replicating to a subset of participants) [25], allowing "flexibility in the number of datacenters required" in causally consistent replication currently "remains an interesting aspect of future work" [19].[2] At a minimum, to ensure convergence, all versions—or, a suitable "final" version of every key—must eventually be broadcast.

3.2 (Not) Scaling Throughput and Datacenters

Convergent causal consistency requires all-to-all replication that limits global write throughput. Assume we have two datacenters, each of which has an apply-capacity of A. To prevent unstable queuing, whereby writes are generated at a rate that is faster than they are applied remotely, the aggregate new write throughput must be limited to A; if we allocate new throughput equally between datacenters, each can locally generate new writes at a rate of $\frac{A}{2}$. Adding a third, equally powerful datacenter does not improve the situation: the aggregate global write throughput is still A, and each datacenter can now generate writes at a rate of $\frac{A}{3}$. With N datacenters, each can write at a rate $\frac{A}{N}$. With heterogeneously powerful datacenters, the sustainable aggregate write rate is limited to the local apply-capacity of the weakest datacenter.

Maintaining per-datacenter write throughput requires total apply-capacity quadratic in the growth of datacenters (or, equivalently, per-datacenter apply-capacity linear in the number of datacenters). For example, if we have 2 datacenters generating $1K$ new writes/s, then each datacenter must have apply capacity of $2K$ new writes/s. If we want to add a third datacenter that can also locally generate writes at $1K$ new writes/s, then each datacenter must now locally apply $3K$ new writes/s. This means that the two

[1]Visibility latency is correlated with the amount of "buffering" of incoming writes (due to missing dependencies) [5]. However, visibility latency measures the *observable effects* of buffering, not the required buffer sizes. In modern systems, operation and visibility latencies are arguably more important than the storage required for the buffered writes.

[2]All-to-all replication (at least of "final" values) appears a necessary cost of convergence. The relevant difference here between causal consistency and a weaker form of non-convergent consistency is that causal graphs are not easily partitionable and intermediate values need to be replicated. While we can attempt to arbitrarily limit the scope of causality to disjoint subsets of data and replicate them to different sets of datacenter [5], this requires high WAN latencies to access remote items that also become unavailable in the presence of partitions, sacrificing the primary (latency and availability) benefits of causal consistency.

existing datacenters must increase apply-capacity by 50% and the new datacenter requires an additional 75% apply-capacity over the existing datacenters: a 125% overall increase in total servers for a 33% increase in throughput. More generally, moving from N datacenters to M datacenters requires $O(\frac{M^2}{N^2})$ more capacity. This is problematic because datacenters are often added to deal with limited capacity of existing datacenters. With convergent causality, the addition of another datacenter requires upgrading all existing datacenters' capacity.[3]

Potential Danger: Sustainable write throughput is limited to the slowest datacenter, so adding datacenters does not increase throughput. Simultaneously scaling writes with datacenters requires quadratic server capacity, and violating this limit leads to arbitrarily high visibility latency.

There are at least two common scenarios in which required write throughput will exceed minimum apply-capacity. First, network partitions and (equivalently) datacenter failures bring the minimum local apply-capacity to zero. A failed datacenter can build up a potentially infinite queue of writes and, after a long enough period of time, will likely instead have to perform a possibly expensive bootstrap operation by "catching up" from another datacenter. Healing multi-datacenter partitions is likely to be even more expensive. Second, "future-proofing" each datacenter becomes a key concern: once write traffic exceeds a datacenter's maximum capacity, the datacenter will either become a bottleneck or have to be decommissioned. Avoiding this scenario requires careful planning and a potentially large amount of overprovisioning to achieve sustainable scale-out (requiring worst-case server capacity from each datacenter measured with respect to the entire lifetime of the service or datacenter).

3.3 Potential Histories and Cluster Capacity

Sustainable write throughput is determined by the rate at which the slowest datacenter can locally apply new writes, which in turn depends on the causality graph. The causality graph fanout (degree) determines the number of dependency checks required, while its depth and connectivity determines the degree of concurrency in performing checks. In practical settings, potential causality graphs are on the order of hundreds of millions of writes. As discussed in Section 2.1, classic causal consistency tracks each data item's *potential* dependencies. On a social network, if a user posts a status update after viewing 10 other updates, the new update potentially depends on all of them. This leads to a graph vertex degree (or fanout) of 10. At the level of a storage system, given that many modern applications are read-dominated, it is likely that this number is much higher—in the hundreds or thousands of dependencies. For many modern web services, it is rare—or impossible—to post an update without viewing tens or hundreds of data items upon which the update will potentially depend. Each update's dependencies' dependencies are included in the graph, along with those updates' dependencies' dependencies' dependencies—ad infinitum—until a (set of) terminal source events is reached.

As an example, consider a user's session on Twitter. Full potential causality graphs are enormous. 20 tweets are displayed on the Twitter homepage upon page load. Currently, scrolling down the home page automatically fetches more tweets at a rate of at least 600 tweets/minute. If a user authors a new tweet after viewing just 20 others, the new tweet will have a potential causal history fanout of at least 20. Each of the tweets in the history will in turn have causal histories, so the size of the new history is exponentially increased. There is redundancy between tweet causality relationships, but an upper bound on the history size is equal to the number of tweets ever authored (currently at 340 million tweets/day [1]). This requires a large number of dependency checks.

Potential Danger: Potential causality graphs have large fanout and depth, limiting local apply-capacities and, accordingly, maximum global throughput.

If all datacenters have applied a version—sometimes called a "stable version" [5]—datacenters do not need to attach it in dependency metadata for new writes or check for it remotely. There is no utility in informing a remote datacenter that it needs to check a given dependency for a given write if we know that the dependency has already been applied there. This greatly reduces the dependencies attached to each item [19]. However, this approach leads to two key challenges. First, it requires global coordination, so, in the presence of any partitions, stability detection will stall and dependency metadata sizes will balloon. Second, (even without partitions) in the presence of the previously described throughput violations, datacenters may slow in applying new versions, in turn decreasing the rate of stabilization.

4 AN EXPLICIT SOLUTION

Instead of tracking all potential dependencies, why not track only those that matter?

[3]Unlike in a causal communication system, which would create additional causality links during all-to-all broadcast, this storage-level limitation is largely independent of the causal graph structure. Structural properties of the causal graph such as its "diameter" and its "number of arcs" [5] affect capacity and influence this trade-off (§3.3) but ultimately act as another independent variable.

Judea Pearl's selection for the 2011 Turing Award was potentially influenced by what he ate for lunch on July 3, 1980, but it was mostly due to his pioneering research on Bayesian networks. We cannot discount the possibility that Pearl's lunch in 1980 played a role in his Turing Award (or, for that matter, the lunch of his neighbor, or even the lunch—or potential lack of lunch—of the reader on that date—or any other meal leading up to the committee's decision), but, for all practical purposes, the majority of these potential causes are irrelevant.[4] Full potential causality covers all possibilities, but it leads to a deluge of relationships, many of which are meaningless.

To address these concerns, we consider *explicit causality*, or application-specified causal dependencies. Instead of including the entire set of possible influences in the `happens-before` relation, we defer to the application to tell the data store which dependencies matter. Each new write is accompanied by a set of dependencies that determine its `happens-before` relation. The causally consistent data store still enforces transitivity of the provided `happens-before` relation but does not enforce program order or reads-from relationships unless explicitly instructed to do so by the application.

This explicit causality does not solve the problem of all-to-all replication, but it has important implications for the trade-off between throughput and latency visibility (§4.2). For example, in a threaded comment feature on a website, instead of including all content a user has viewed, her comment's immediate `happens-before` dependency could consist entirely of its `in-reply-to` field. Just as historians curate the entire history of the universe leading up to an event under study by extracting its relevant influences, explicit causality allows applications to pare the causality space. However, the application's task is often much easier than that of the historian (§4.3).

4.1 Explicit Causality in the Wild

While prior research briefly considered variants of explicit causality in the form of "state"- and "client"-level dependency tracking [4, 5, 16], its time has finally come: modern applications like social networking hugely benefit from semantic pruning of causal relationships. To quantitatively study the structure of explicit causality graphs, we surveyed existing literature on user behavior on several modern human-facing Internet services.

Transitive explicit causality relationships across operations are small: lengths are often in the tens of events and maximally several thousand events. In recent analyses, 28% of Tweets were part of conversations, with an average conversation depth of 10.7 Tweets [29]. 69% of conversations were of depth two (a Tweet and single reply) and the maximum observed length was 243 Tweets [23]. On Facebook, causality chains of page "fanning" (e.g., Patty "likes" Design) were maximally depth 82, with 75% of chains under length 3 and 98% of chains under depth 18 [27]. Applications such as blogs (average comment chain depth 6.3 to 93) [21, 28] , interactive websites (99th percentile chain depth $1,000$ comments) [11], corporate email (80th percentile thread size of 8, average 4.1) [14, 15], and chain emails (median depth 288) [18] exhibit similar trends.

Tracking explicit causality for human events is cheap: the number of writes in any given explicit causal history will be small (typically single-digit and, occasionally, hundreds or a few thousands of writes), limiting the graph sizes. Similarly, the number of participants in any given causality chain is also limited, helping to limit the degree to which individual graphs intersect. For major services like Twitter, the potential causality chains for even a year of operation are approximately nine orders of magnitude larger than a pessimistic explicit causality chain (e.g., $340M * 365$ vs. 100).

4.2 Benefits: Relieving the Pressure

Explicit causality's quantitatively different graph structure helps mitigate the potential dangers of causal consistency. Explicit causality does not directly address the problem of all-to-all replication (§3.2) but increases each datacenter's apply-capacity (§3.3).

Smaller Degree: Faster Checks Each new write depends on few others. This means that dependency checking will be faster. Instead of checking whether hundreds or thousands of other immediate dependencies in the causality graph have been applied within a remote datacenter, the datacenter need only check a small number. This lowers load on individual servers within each datacenter, increasing capacity and lowering visibility latency.

Smaller Degree: Decreased Metadata In addition to speeding up dependency checks, the decreased fanout of each write decreases metadata overheads. Simply, if each write depends on fewer others, the storage and communication overhead required for each write due to metadata will also decrease. If we assume 8 bytes per version identifier used to identify each dependency for a given write (say 4 for the key and 4 for the version), a modest degree of 10 results in a metadata overhead of 80 bytes. This is far

[4] To further illustrate the subtleties of this point, we cannot categorically reject all lunches as practically irrelevant to major scientific progress: Richard Feynman famously describes the experience of watching a plate's motion in a Cornell cafeteria as substantial motivation for the research that led to his 1965 Nobel Prize in Physics [8].

improved from the potential causality requirements. While potential causality overhead may be reduced during normal operation due to garbage collection (§3.3), we no longer need to rely on this mechanism to keep metadata small. Accordingly, in the presence of partitions, system behavior should degrade much more gracefully.

Fewer Vertices: Increased Concurrency Each update is part of a small causality graph. Instead of having a potential causality graph with a high degree of connectivity across all updates (likely completely connected), explicit causality results in several smaller, disjoint graphs. Semantically unrelated updates are disconnected, so we have more independence between sets of writes. Writes to each independent graph can be applied in parallel, decreasing serial dependency bottlenecks. If one graph is missing a critical update, it is likely that there will be another that can be applied while we wait.

4.3 Application Programming Model

Explicit causality is powerful because it tracks only relevant dependencies, yet this is potentially onerous for the application programmer. However, for many applications, explicit causality is easily captured and it is frequently already available in the data model. The studies in Section 4.1 were made possible because services already recorded dependency information, even if the dependencies are subtle, like Facebook "fanning" chains. Service operators can use this data for many tasks, like offline analyses of user behavior and targeted advertising and are also incentivized to collect it from a user experience perspective: presenting a user with all possibly relevant content can lead to information overload [13] (i.e., many times, users want to view a parent comment or follow a reference to another resource, and providing a direct hyperlink provides better navigation than having to perform a full search). While these dependencies are likely recorded, they are not necessarily (and, anecdotally, are not yet) used at runtime for consistency purposes.

Under explicit causality, each application defines its own `happens-before` relationships. This means that each write to the data store must be accompanied by a (potentially empty) set of dependencies supplied by the programmer. We discussed one causal consistency violation in Section 2. Under explicit causality, the application would ensure that Lewis's event L^* `happens-before` Mary's reply M by specifying that parent comments `happen-before` their children. If a chat client wanted to enforce "program order" for each participant, it would place each new message `after` its immediate predecessor. More generally, applications can place earlier comments and the original post `before` each new posted comment. References are easily captured at the level of the data store (e.g., referencing a unique identifier such as a comment ID or, more generally, primary and foreign key references).

More complex scenarios are possible. For example, as we alluded to in Section 1, recent work frequently refers to an example of updating one's privacy settings and subsequently posting a sensitive update on a social network [6, 19]. If the privacy setting replicates slower than the new update, the new update might be shown to an unintended audience. Under explicit causality, the application would ensure that each user's most recent privacy policy `happens-before` each of their writes. This analogous to performing a memory `fence` instruction with respect to the privacy change.

4.4 Limitations

Explicit causality has limitations. Users can still circumvent causality-tracking mechanisms. For example, if a user's status update references an event in the system by name rather than by reference (e.g., Jeremy: "You should see the picture I just uploaded!"), the data store will not be automatically aware of the dependency and users may observe dangling references (e.g., Sue can't find Jeremy's picture). Moreover, this data modeling is not free; although many applications already capture their explicit dependencies, application writers must now consider explicit causality relationships as an integral part of their application logic. Finally, convergent explicit causality still faces the peak throughput problems of all-to-all replication, though the minimum datacenter throughput is likely increased due to the factors described in Section 4.2.

5 DISCUSSION

In this section, we discuss causality at the communication, storage, and application levels, hypothesize why potential causality is so prevalent in the literature, and comment on existing architectures for causal consistency.

Communication and Storage APIs Potential causality effectively presumes the existence of a "universal sensor" for all events in a system. This sensor typically records "physical" events such as sending and receiving messages or reading and writing data items. In contrast, explicit causality prescribes the voluntary use of a logging API for recording *relevant* events in a system. This logging is designed to capture higher-level "logical" events from the application (e.g. "reply" or "like"). The voluntary API is best utilized at system layers in which causality is easy to understand and inexpensive to capture. These layers are likely at a higher level than a storage or communication library; in our discussion, we have focused on application-level causality.

Potential Causality In light of the scalability bottlenecks we describe in this paper, why has potential causality been so popular? Much of the literature considers distributed registers or communication channels in isolation with arbitrary reads and writes (or messages); it is difficult to advocate the use of explicit causality without knowledge of an application or higher-level semantics. However, this lack of situational knowledge also helps explain why full potential causality is often included in discussions of distributed debugging [9]: by definition, the user does not know the root cause of the error she is debugging. She may be able to rule out large portions of the potential causes using expert knowledge, but this is not as easy to express as the dependencies we consider here. Finally, prior systems like Bayou [22] that implemented causal consistency had more limited scale. Web-scale services involve unprecedented volumes of data and user interaction, prompting a reassessment of prior approaches.

Existing Architectures This paper has exposed scalability limitations of causally consistent stores. These problems are fundamental to the formulation of causality: for effects to happen after their causes, we either need to perform dependency checking or synchronize before sending updates. Our goal in this work is *not* to criticize any particular existing architecture but instead the prevalent definition of the `happens-before` relation. Recent work—notably, the COPS architecture—performs well, achieving high throughput and low latency in the evaluation presented [19]. We address two facets of causal consistency that this work did not investigate: peak throughput in scaling to multiple datacenters and requirements for real-world causality graphs. We expect that, under explicit causality, COPS's partitioned log shipping will achieve higher throughput, better scale to 3 or more datacenters, and handle partitions without severe metadata growth. The behavior of convergent causal consistency under real-world (non-synthetic) workloads remains an open question.

6 CONCLUSION

Causal consistency results in a tension between visibility latency and throughput. Peak throughput remains constant as more datacenters are added to a convergent causally consistent system, and scaling throughput with datacenters requires quadratic hardware provisioning. To alleviate these concerns, we can decrease per-operation storage and processing costs via application-level explicit causality. Instead of tracking all potential influences, we advocate tracking only those that matter, allowing applications to define their own `happens-before` relations. Quantitative evidence from a variety of human-facing applications demonstrates that, for a wide range of modern services, explicit causality dependency graphs are a small fraction of the size and complexity of traditional potential causality graphs. This allows faster, more parallelizable dependency checking, helping mitigate the potential dangers we describe. Ultimately, all-to-all version propagation limits sustainable write throughput in current algorithms for convergent causal consistency, but, by exploiting semantic information, we can reduce the severity of the problem.

If convergent causal consistency is to be the preferred model for future weakly consistent distributed data stores, these dangers must be studied in greater detail. Explicit causality lowers the price of causal consistency, but whether even this decreased cost is offset by the benefits that causality provides over weaker forms of convergent consistency is unknown. A serious positive recommendation requires further consideration of application semantics, realistic workloads, and both expected and worst-case operating conditions.

ACKNOWLEDGMENTS

The authors would like to thank Neil Conway, Aurojit Panda, and Patrick Wendell for constructive commentary on earlier versions of this manuscript.

This work was supported by gifts from Google, SAP, Amazon Web Services, Blue Goji, Cloudera, Ericsson, General Electric, Hewlett Packard, Huawei, IBM, Intel, MarkLogic, Microsoft, NEC Labs, NetApp, NTT Multimedia Communications Laboratories, Oracle, Quanta, Splunk, and VMware. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant DGE 1106400, National Science Foundation Grants IIS-0713661, CNS-0722077, and IIS-0803690, the Air Force Office of Scientific Research Grant FA95500810352, and by DARPA contract FA865011C7136.

7 REFERENCES

[1] Twitter turns six, March 2012. http://blog.twitter.com/2012/03/twitter-turns-six.html.

[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] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementation and programming. *Distributed Computing*, 9(1), 1995.

[4] 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.

[5] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *SOSP '93*.

[6] 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. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[7] A. Fekete, D. Gupta, V. Luchangco, N. A. Lynch, and A. A. Shvartsman. Eventually-serializable data services. *Theor. Comput. Sci.*, 220(1):113–156, 1999.

[8] R. P. Feynman. *"Surely You're Joking, Mr. Feynman!": Adventures of a Curious Character*. W. W. Norton & Company, Inc., 1985.

[9] C. J. Fidge. Partial orders for parallel debugging. In *PADD 1988*.

[10] 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.

[11] V. Gómez, H. J. Kappen, and A. Kaltenbrunner. Modeling the structure and evolution of discussion cascades. In *ACM HT 2011*.

[12] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[13] S. R. Hiltz and M. Turoff. Structuring computer-mediated communication systems to avoid information overload. *Commun. ACM*, 28(7):680–689, July 1985.

[14] B. Kerr. Thread arcs: an email thread visualization. In *INFOVIS 2003*.

[15] B. Klimt and Y. Yang. The Enron corpus: A new dataset for email classification research. In *ECML 2004*.

[16] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: exploiting the semantics of distributed services. In *PODC 1990*.

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

[18] D. Liben-Nowell and J. Kleinberg. Tracing information flow on a global scale using internet chain-letter data. *PNAS*, 105(12):4633–4638, 2008.

[19] 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*.

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

[21] G. Mishne and N. Glance. Leave a reply: An analysis of weblog comments. In *Third Annual Workshop on the Weblogging Ecosystem*, 2006.

[22] 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*.

[23] A. Ritter, C. Cherry, and B. Dolan. Unsupervised modeling of Twitter conversations. In *HLT 2010*, pages 172–180, 2010.

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

[25] N. Schiper, P. Sutra, and F. Pedone. P-store: Genuine partial replication in wide area networks. In *IEEE SRDS 2010*, pages 214–224.

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

[27] E. Sun, I. Rosenn, C. Marlow, and T. Lento. Gesundheit! Modeling contagion through Facebook news feed. In *ICWSM 2009*.

[28] T. Yano, W. W. Cohen, and N. A. Smith. Predicting response to political blog posts with topic models. In *NAACL 2009*.

[29] S. Ye and S. F. Wu. Measuring message propagation and social influence on Twitter.com. In *SocInfo'10*, pages 216–231. Springer-Verlag, 2010.