Mitigating Contention in Distributed Systems

General Exam

Brandon Holt
University of Washington
bholt@cs.uw.edu

Abstract

The online world is a dangerous place for interactive applications. A single post by Justin Bieber can slow Instagram to a crawl; a black and blue dress going viral sends Buzzfeed scrambling to stay afloat; Star Wars movie ticket pre-sales cause service interruptions for Fandango and crash others. Developers of distributed applications must work hard to handle these high-contention situations because though they are not the average case, they affect a large fraction of users.

Techniques for mitigating contention abound, but many require programmers to make tradeoffs between performance and consistency. In order to meet performance requirements such as high availability or latency targets, applications typically use weaker consistency models, shifting the burden of reasoning about replication to programmers. Developers must balance mechanisms that reign in consistency against their effect on performance and make difficult decisions about where precision is most critical.

Abstract data types (ADTs) hide implementation details behind a clean abstract representation of state and behavior. This high-level representation is easy for programmers to build applications with and provides distributed systems with knowledge of application semantics, such as commutativity, which are necessary to apply contention-mitigating optimizations. With *inconsistent, probabilistic, and approximate (IPA) types*, we can express weaker semantics than with traditional ADTs, enabling techniques for weak consistency to be used and allowing precision to be explicitly traded for performance where applications can tolerate error. In previous work, we have demonstrated that ADT awareness can result in significant performance improvements: 3-50x speedup in transaction throughput for high-contention workloads such as an eBay-like auction service and a Twitter-like social network. IPA types are proposed for future work.

1. Introduction

Imagine a young ticket selling app, embarking on a mission to help people everywhere get a seat to watch their favorite shows. Bright-eyed and ready to face the world, it stores everything in a key/value store so that it will not forget a thing. It uses a distributed key/value store, so when it expands into new cities and reaches its millionth customer, the datastore continues to grow with it. The app uses strong consistency and transactions to ensure that no tickets are sold twice or lost, and its customers praise its reliability. "Five stars. That little app always gets my tickets, fast!" they say.

Then one day, pre-sales for the 7th Star Wars movie come out and suddenly it is inundated under a surge over 7 times the usual load, as a record-breaking number of people try to purchase tickets for this one movie. This concentrated traffic causes *hot spots* in the datastore; while most nodes are handling typical traffic, the traffic spike ends up being funneled to a handful of nodes which are responsible for this movie. These hotspots overload this app's "scalable" datastore, causing latencies to spike, users' connections to time out, and disappointed users

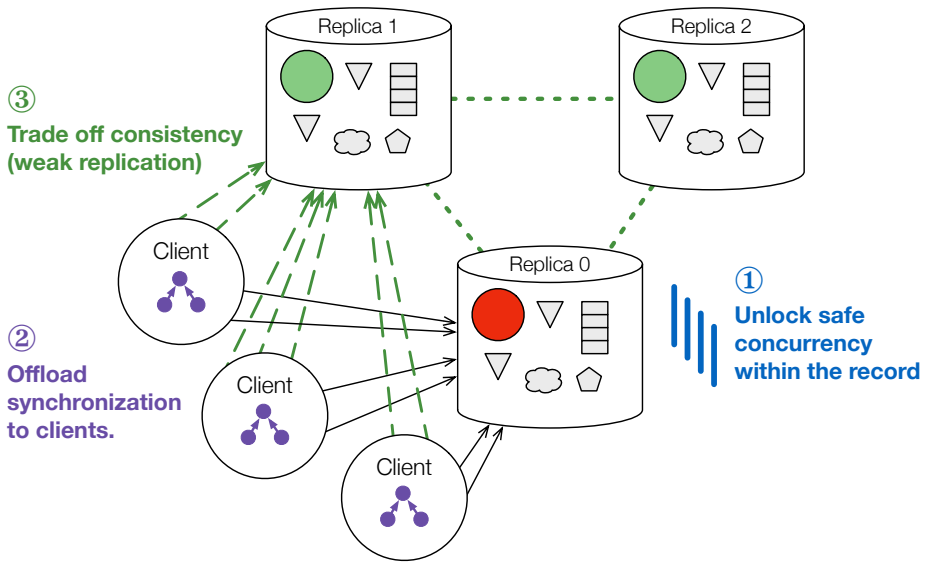


Figure 1. Overview of approaches for mitigating contention, such as the hotspot in red.

who will not get to see the movie on opening night because the app was not able to sell them a ticket. On this dark night for moviegoers, even major players like Fandango, Regal, and AMC are plagued with service interruptions; some sites even crash or lose data.

This kind of concentrated surge in traffic is a perfect illustration of the contention that occurs all the time in real-world workloads, from ticket sales to social networks. Power law distributions are everywhere, from popularity of pages to number of followers, leading to network effects which can magnify even the slightest signals. Events – such as the FIFA World Cup or the Oscars – happening in realtime drive spikes in traffic. All together, this can create memes that propagate virally through social media and news sites. Interactivity is crucial to all of this — it both fuels the propagation of memes and causes contention.

Paradoxically, though contention is not the average case, it is responsible for many of the most challenging problems: tail latency, failures, and inconsistency bugs. Most of the time, an application may work correctly, with low latency and no noticeable inconsistency. However, in that 99th percentile case, contention results in a hot spot, where latencies spike, failure rates go up, and consistency can degrade, exposing bugs or resulting in user-visible inconsistencies.

1.1. Mitigating contention

Many techniques, over many years, have tackled this problem from different angles, from research on escrow and fragmenting aggregate fields in the 80s [51], to modern research on auto-generated coordination based on annotations [15, 63]. Some require a variety of changes to the programming model, while others improve underlying protocols and mechanisms. All are focused on exposing concurrency and making tasks simpler for programmers, but they can be broken down into three broad approaches, shown in Figure 1.

1. Is there any concurrency within the contended record that can be exposed? If operations on the record are commutative, they can safely run concurrently without changing the semantics; *abstract locks* (§6.2.1) leverage this to allow transactions to overlap and avoid false conflicts. *Escrow* (§4.1.3) allows the record to be treated as if it was split into a pool of fragments.

2. If clients are multithreaded, as is the case with frontend web servers that typically han-

dle many end-user requests concurrently, then some of the synchronization work can be offloaded to them. *Combining* (§6.1) leverages associativity to allow operations to be merged locally and executed as a single operation remotely, reducing the amount of work done on the overloaded datastore. Other techniques like *leases* (§4.2.1) let client-side caches avoid costly invalidation messages.

3. If clients are allowed to interact with weakly-synchronized replicas, the load on the contended shard can be reduced. However, this comes at a significant programmability cost: the illusion of a single copy of data is broken and programmers must now reason about replicated state. Weakly synchronized replicas share updates asynchronously, and clients may communicate with multiple replicas, so they can observe effects out of order, or perform updates which conflict and result in inconsistent states.

To be clear, techniques in the first and second categories can use replicas for fault tolerance and still maintain strong consistency. Strong consistency requires writes to be linearizable [37], which can be accomplished with replicas either by funneling through a single master or by a consensus protocol like Paxos [43] that makes replicas appear like a single machine. Techniques like E-Paxos [48] would fall under (1) because they expose concurrency while maintaining the single-machine view.

The third category requires much more drastic changes to programming models than the first two because it forces programmers to sacrifice consistency guarantees. However, weak consistency unlocks further improvements to performance properties like availability and lower latency that are impossible with strong consistency. In this work I will focus mostly on techniques that fall into this category because these are the most challenging to understand and require the most significant changes for programmers.

1.2. Balancing requirements

Mitigating contention is just one of the many competing requirements placed on distributed applications. They are expected to be *scalable*, *fault tolerant*, and *highly available*. Users demand responsive applications, no matter how many other users are active or where they are. The common wisdom is that companies lose money for every increase in response latency. Many systems have service-level agreements (SLAs) promising responses within a certain latency for all but the 99th percentile of requests. Throughput during peak times must be able to keep up with the load.

These performance constraints compete with the desire for *programmability* and *correctness*. Fundamentally, strong consistency cannot be provided with high availability — replication must be exposed in some way. Strict serializability and ACID transactions are among the many useful programming abstractions that must be broken to achieve those performance properties. Applications are still expected to appear mostly consistent, so developers are forced to think carefully about how to build correct systems with weaker guarantees and choose where to focus their effort.

Luckily, not everything requires the same level of precision or consistency. Some actions, such as selling the last ticket to Star Wars' opening night, require precise, consistent execution. Other situations, such as viewing the number of retweets for a popular tweet, do not need to be exactly correct — users may be satisfied as long as the number is the right order of magnitude. These *hard* and *soft constraints* can be difficult or impossible to express in current systems. Furthermore, whatever tradeoffs are made to improve programmability or enforce constraints must be balanced against meeting the performance targets, though it is not easy to quantify how changes will affect them. Programmers of these distributed applications must constantly juggle competing correctness and performance constraints.

1.3. Overview

To understand how the various techniques for managing consistency and performance relate to one another, we will explore them in terms of the properties they trade off:

- *Ordering* and *visibility* constraints between operations
- *Uncertainty* about the state in terms of staleness and possible values

We will also discuss how each technique operates, in terms of:

- *Granularity:* Does it affect the whole system, specific records, or specific operations?
- *Knowledge vs control:* Are users granted additional information about performance or consistency or are they given explicit control?

After exploring the space of existing techniques, we will propose a programming model that incorporates these disparate solutions into a single abstraction – using *abstract data types* (ADTs) to concisely describe application semantics and hide the details of the underlying consistency and coordination techniques. Our implementations, in a distributed data analytics system, *Grappa*, and a prototype ADT-store, *Claret*, show significant performance improvements, especially for high contention workloads. In future work, we propose using *inconsistent, probabilistic, and approximate (IPA) types* to trade off precision in order to take advantage of weak consistency and replication for high availability and scalability.

2. Trading off consistency for performance

In order to meet scalability, availability, and latency requirements, distributed systems programmers must routinely make tradeoffs between consistency and performance [11, 21, 32]. In the *typical* case, consistency does not pose a problem, even for geo-replicated systems – most requests are reads, requests for the same user typically go to the same server, and inconsistency windows after updates are usually small [11, 47]. Consistency becomes problematic in exceptional *high contention* cases and the *long tail* of disproportionately slow requests [28]. However, these cases are almost pathologically bad – many users together create conflicts leading to inconsistency, but this also means that many users are there to observe the inconsistency, so it is more likely to be noticed. The events most likely to cause problems are also the most newsworthy.

Targeting performance for the average case can lead to catastrophic failures in the contentious cases, but the handling all possible cases damages programmability. This burden will become abundantly clear over the next few sections as we look at the axes which programmers must traverse when making implementation choices.

3. Ordering and visibility constraints

Understanding the behavior of a sequence of operations in a program requires knowing all of the ordering and visibility constraints that govern what each operation returns and when each update actually runs. How these constraints are set is determined by the programming model.

3.1. Consistency models

Analogous to memory models in the field of computer architecture, consistency models refer to the allowable reorderings of operations and their visibility in a distributed system. Consistency models differ from memory models primarily in one way: exposing the existence of replication. Due to the reliability and speed of CPU cache hierarchies, memory models can afford to assume *coherence* will ensure that there appears to be only one copy of memory. In distributed systems, where failure is a possibility and synchronization is expensive, it is often necessary to expose

replication through the consistency model. This makes them significantly more difficult to reason about – as if memory models were not complex enough as it is.

The strongest consistency model, *strict serializability* (roughly defined as Lamport's *sequential consistency* [42] combined with Herlihy's *linearizability* [37]) guarantees that operations appear to occur in a global serial order that all observers agree on and that corresponds to real time. This, and any form of consistency that requires enforcing a *global total order* is theoretically impossible to enforce with *high availability* due to the possibility of network partitions (this is the essence of the CAP theorem [21, 32]). In practice, strict serializability may not be wholly impractical for the average case, but ensuring it in *all* cases is prohibitively expensive.

At the other extreme, *eventual consistency*, the least common denominator among consistency models, simply guarantees that if update operations stop occurring, all replicas will eventually reflect the same state [70]. Under this model, programmers cannot count on subsequent operations reflecting the same state, because those operations could go to any replica at any time, and those replicas are continuously receiving updates from other nodes.

There are a whole family of models similar to eventual consistency which add various ordering constraints:

- *Monotonic writes* (MW) ensure that writes from a client are serialized, enforcing *ordering* between writes.
- *Monotonic reads* (MR) ensures that reads will not observe earlier values than have been seen by a particular client already, strengthening *visibility*.
- *Read-your-writes* (RYW) ensures that a client will at least observe its own effects – primarily strengthening an aspect of *visibility*.
- *Causal consistency* ensures that operations from different clients causally following a write will observe that write (by some definition of *causation* which the system must track). This means that operations will be *visible to* and *ordered with* each other when applicable.

There are too many variations on these and other models to enumerate, including combinations of them. Each restricts *ordering* and *visibility* differently, making some cases easier for programmers to reason about, while reducing the flexibility and therefore performance of the system. For instance, some require *sticky sessions* [68], which forces clients to continue communicating with a particular replica, even if it is not the fastest, or lowest latency, or most up-to-date one available.

As a way of trading off consistency for performance, weak consistency models are a poor choice. A particular consistency model must often be chosen at a very coarse grain, possibly at the level of an entire database. Stronger guarantees can be enforced on top of a weaker model using quorums, but these must be chosen carefully, and the code to handle this is not easily adapted to changes in the underlying consistency model. In general, weak consistency models are not modular: adding or changing an operation in one place may require changing assumptions about ordering elsewhere. Understanding when an operation becomes *visible* to others is yet another source of confusion.

3.2. Transactions

Transactions are a well-established way to provide stronger guarantees among some operations. By choosing the type and granularity of transactions, programmers have some control over the *ordering* and *visibility* of their operations. Like strict serializability, full ACID transactions require a global order so are prohibitive to scaling and high availability, leading to the development of weaker transaction models. As the antithesis of the strong guarantees of ACID, some have termed these weaker semantics *"BASE"* (Basically Available, Soft state, Eventually Consistent) [53]. Luckily, programmers are not restricted to simply choosing between these two extremes; some have proposed ways to bridge the gap.

3.2.1. Transaction chopping and chaining

Some techniques can expose limited extra concurrency between transactions without requiring programmers to sacrifice ACID semantics. Transaction chopping [60] automates a task programmers could do by hand: breaking transactions into minimal-sized atomic pieces, determined by a static analysis of interleavings. A more recent system, Lynx [76] executes split transactions as a *chain*, hopping from shard to shard, coordinating the order of execution to allow conflicting transactions to safely interleave. Finally, Callas [74] groups transactions that commute with each other to allow them to execute concurrently. Transaction boosting [35] similarly allows transactions to overlap when they commute, but at the level of individual operations.

3.2.2. Salt: Combining ACID and BASE

Just as a small fraction of data items are responsible for the majority of contention, the same is true for transactions. Rather than forcing programmers to give up ACID semantics for their entire application, Salt [73] allows transactions with BASE semantics to coexist safely with ACID transactions. Using new locking schemes, they ensure that transactions executing with weaker BASE semantics cannot violate the strong safety guarantees of the ACID transactions.

Converting an ACID transaction to execute without those guarantees is an error-prone task; it involves considering all the new possible interleavings and establishing how to resolve all the possible conflicts without coordination. Salt's model means that programmers only need to "BASE-ify" the transactions causing performance or scaling problems. This makes it relatively straightforward to trade off consistency where necessary, but does not do much to help programmers deal with the weaker semantics.

3.2.3. RAMP transactions

Even without guarantees of a serializable total order, there are still benefits to supporting atomic updates: preventing foreign key constraint violations, and ensuring that indexes and other derived data are as up-to-date as the backing data. In support of these safety properties, *RAMP (Read Atomic Multi-Partition) transactions* provide coordination-free atomic visibility for multiple updates [12]. They work by *staging* updates on all participating shards so they can force the complete set of updates for a transaction to be made *visible* at one point in time. They dynamically detect racy reads and fix them by either selecting an appropriate staged version or waiting for another round of communication. Because these determinations are made locally, they cannot guarantee global mutual exclusion or serializability.

3.3. Models inspired by distributed version control

One of the troubles with weakly consistent replication is that it is often not possible to construct a serializable history of an execution, making it difficult to figure out which effects could have been visible to a particular client. Distributed version control systems (DVCS) like Git have inspired alternative ways of viewing concurrent execution. DVCSs allow individuals to work concurrently on *forks* or *branches* without interference and resolve conflicts at explicit *merge* points later. These histories are not serializable, but they still allow users to easily understand when effects become visible to different observers.

The *Push/Pull Model of Transactions* [40] formalized several consistency and transaction models in these terms. Another model called *branch consistency* proposed for a system called TaRDIS [27] uses the notion of branching and merging for isolation and conflict resolution in geo-replicated systems, delegating conflict resolution to applications.

Concurrent revisions [22] proposes an execution model built around forking and joining state along with concurrent execution to make sharing explicit. In this model, concurrent tasks *fork* a copy of the state they access. Changes to forked state are only visible to that task and its descendants until the concurrent task is *joined* back in. On join, changes to forked data items

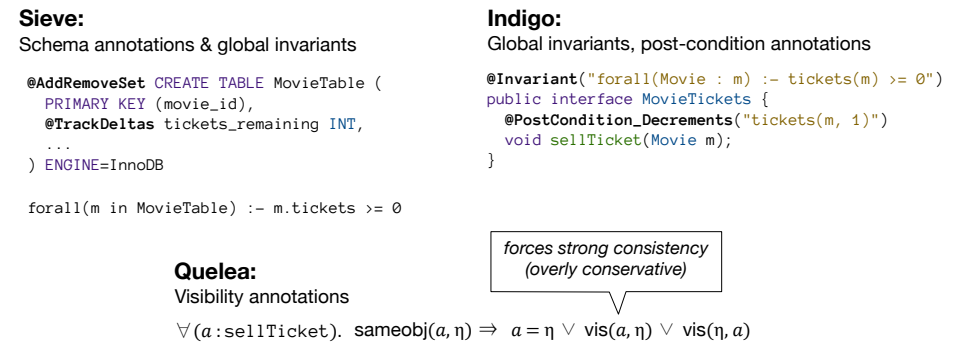


Figure 2. *Annotating application invariants.* Annotations are used to determine where coordination is necessary and what consistency is required to enforce it.

are *merged* according to their type. For example, as a cumulative type, a forked `Counter` tracks increments made to it, and when *joined*, adds those increments to the original value. In this way, multiple concurrent tasks can increment the shared `Counter` without conflicting and it is clear exactly when their effects are made visible. Follow-on work extended the ideas of concurrent revisions and revision diagrams to reason about eventual consistency: in eventually consistent transactions [23] and mobile/cloud applications [24].

These programming models show that there is hope for reasoning about weakly consistent replicated data. The *isolation types* and *cloud types* from concurrent revisions provide useful semantics for working with highly contended data. In these models, trading off consistency for performance is not as clear-cut; *revision diagrams* and DVCS histories imply strict coordination points. It is also unclear how to enforce global constraints on forked data. Consider again the ticket sales example: in order to ensure tickets are not over-sold, concurrent forks must somehow know how many of the remaining tickets they are allowed to sell, without knowing how many other forks exist, breaking the abstraction.

3.4. Annotating constraints

Several datastores allow consistency levels to be specified on a per-operation basis: research systems Gemini [44] (RedBlue consistency), and Walter [64], and production systems Cassandra [8] and Riak [18] (per object or namespace). However, they leave programmers to determine where to use stronger consistency in order to achieve their correctness goals, a very error-prone task. Recent work has explored ways of *automatically* choosing the correct consistency level or coordination strategy based on annotations.

Sieve [45] builds on top of Gemini, automatically determining how to implement the desired semantics with causal consistency and adding additional synchronization wherever strong consistency is needed. It relies on programmer-specified global invariants and annotations on the relational database schema to select the desired merge semantics in case of conflicts. These annotations echo the variants of CRDTs (see §4.1.1).

Quelea [63] has programmers write *contracts* to describe ordering constraints between operations and then automatically selects the correct consistency level for each operation to satisfy all of the contracts. Contracts are specified in terms of low-level consistency primitives such as *visibility* and *session order*. For example, to ensure a non-negative bank account balance, a contract indicates that all `withdraw` operations must be visible to one another, forcing the operation to be executed with sequential consistency. Because correctness properties are specified *independent of a particular consistency model*, or set of consistency levels, they are *composable* with each other and *portable* to other datastores supporting different consistency options. However, the low-level primitives used in contracts may not be intuitive for program-

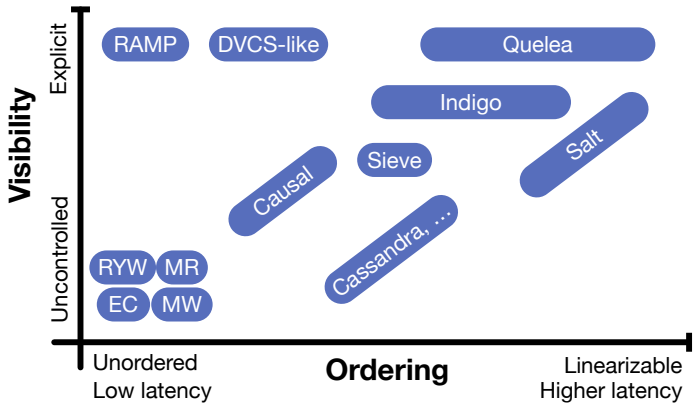


Figure 3. Ordering and visibility constraints.

mers and still require reasoning about all the possible anomalies between operations. These primitives are unable to capture other forms of coordination, sometimes leading to more conservative ordering constraints than necessary.

Indigo [15] takes a different approach to expressing application requirements: instead of specifying visibility and ordering constraints, programmers write *invariants over abstract state and state transitions*, and annotate *post-conditions* on actions to express their side-effects in terms of the abstract state. They then perform a static analysis to determine where concurrent execution could violate the invariants and add coordination logic to avoid those conflicts. Supported constraints include numeric constraints, such as lower or upper bounds on counts, as well as integrity constraints and general compositions of these.

Figure 2 shows how annotations would be written in these three systems to implement our running ticket sales example. In this case, the desired invariant is that tickets are not oversold – that is, the count of remaining tickets should be non-negative. Sieve and Indigo can enforce this invariant directly as written. Quelea's visibility-based contracts cannot tightly describe this invariant; instead, they must be conservative and force ticket sales to be strongly consistent.

Indigo's approach provides an excellent way to express application-level semantics and have the system automatically figure out how to enforce them. The primary downside is that abstract state must be modeled separately from the true application state. Additionally, though their invariants can specify hard constraints, they do not have a way to express the soft constraints we discussed in §1.2.

3.5. Review: Constraints

Figure 3 organizes the systems we have so far discussed along axes of increased ordering and visibility constraints. There is no *best* point in this space – increased ordering constraints come with restrictions which limit availability and increase latency. The best solutions, therefore, provide the most flexibility and control over the constraints they can enforce.

Weak consistency models provide little in the way of ordering or visibility guarantees. With eventual consistency at the very bottom, Read-Your-Writes additionally constrains operations within a session to be visible to one another, while Monotonic Writes force some order among writes, and Monotonic Reads the *visibility* of those ordered writes. Of the consistency models, Causal provides the most programmer control – essentially arbitrary constraints can be created by forcing the system to consider them causally linked.

RAMP transactions and DVCS-like techniques allow explicit control over visibility but lit-

G-Set	"Grow-only" set, `remove` is simply disallowed.
PN-Set	A counter per item matches `add`'s and `remove`'s, the set contains the item whenever there are more `add`'s.
OR-Set	"Observed-remove" set where causally-related `removes` are observed, but `add` wins over `remove` when concurrent.

Table 1. Example `Set` CRDTs. Variations are due to the fact that `add` and `remove` do not commute for a sequential `Set`.

tle control over ordering between transactions (or branches/forks), placing them at the top left of Figure 3. Because Salt allows weaker transactions to interoperate with ACID transactions, it allows programmers to select from range of ordering and visibility levels depending on need. Cassandra, Riak, Gemini, and Walter allow per-operation consistency levels to be set, which also provides a range of constraints, but with significant complexity for programmers.

The annotation-based techniques provide the most control over these constraints and also automate part of the task of choosing them. Quelea, with contracts specifically on these constraints, provides the most control, but at a lower level of abstraction than Indigo or Sieve's application-level invariants. So far, we have been dealing mostly in terms of plain reads and writes; the next section will show how to do better by placing bounds on the *allowable values* and *staleness*.

4. Uncertainty

Factoring out the knowledge provided by ordering and visibility constraints, the state observed by operations is *uncertain* – subject to constraints on what *values* the piece of data may hold and how out-of-date the replica is.

4.1. Restricted values

One of the problems with using eventual consistency is ending up with conflicting writes that overwrite one another in unpredictable ways. The solution to this without enforcing mutual exclusion somehow is to define commutative deterministic *merge functions* that resolve conflicts resulting from concurrent updates. The vision of Bayou [67] was that applications would define custom merge functions over all of the application state so that users could work offline and automatically synchronize their changes the next time they connected. Though this has not caught on in any major way due to the difficulty (and non-modularity) of handling all possible combinations of updates in a way that is satisfactory to users, it has led to the development of libraries of data structures with this property called CRDTs.

4.1.1. CRDTs

Convergent (or *conflict-free*) *replicated data types* (CRDTs) [59] are data types that have commutative merge functions defined for them. Resolving conflicts deterministically requires making choices about the semantics of concurrent updates, leading to a proliferation of CRDTs for various use cases. Even simple data structures like Sets must have multiple variants such as those in Table 1 that resolve non-commuting operations differently.

CRDTs can be enormously useful because they allow concurrent updates to accumulate. Riak [18] implements several data types and encourages their use. Like ADTs, CRDTs also provide a well-defined set of possible values that a variable can hold, restricted to changes made by supported operations. This is a clear advantage over simple *registers* that are completely overwritten on each write, making them unpredictable when the order of updates is uncertain. CRDTs can still suffer from many of the effects of eventual consistency. Updates applied to different replicas mean clients could see divergent changes for some time, and convergence

does nothing to change that. Keep in mind that divergence *could* continue indefinitely thanks to eventual consistency, we will get to techniques that quantify the actual time this takes.

4.1.2. Bloom

The philosophy of Bloom [5, 26] is to find ways to write programs that avoid the need for coordination as much as possible. The *CALM Principle* (Consistency And Logical Monotonicity) advocated by this work formalizes the requirements for an entire program to be eventually consistent, obviating any need for coordination. It is built around the notion of monotonicity—programs compute sets of facts that grow over time so that information is never lost and convergence can be guaranteed.

In the Bloom model, programmers express applications as *statements* about monotonically growing sets of *facts*. These facts can be encoded as sets or other collections with suitable *merge functions* ensuring values of the type have a well-defined partial order, such as CRDTs [26]. Bloom statically ensures that programs compose these types in ways that are monotonic.

In its pure form, Bloom's programming model requires substantial changes to most applications, so later work on Blazes [6] showed how the monotonicity analysis could be applied to find where coordination is necessary in existing distributed applications by annotating components with Bloom's properties. This style of programming is somewhat different than the other annotation-based approaches of Indigo and Quelea because it focuses on *sealing* streams at coordination points. Bloom and its variants can ensure *eventual* consistency, but again this says nothing about how long it will take or what intermediate states will be observable, so in practice, users would still have to worry about observing stale or inconsistent states.

4.1.3. Escrow and Reservations

Escrow is a term from banking and legal proceedings where some amount of money is set aside and held by a third party in order to ensure it will be available for use at a later time after some (typically legal) condition is met. In database systems, this term has been borrowed for use in concurrency control to refer to "setting aside" some part of a record to be later committed.

O'Neil's idea of *escrow* [51] came from work on Fast Path [31] and Reuter's Transactional Method [54]. The idea was to increase concurrency on *aggregate fields*, such as fields keeping track of a count or a sum, which could become hot spots because they were updated frequently. The idea of escrow is to split up an aggregate value into a *pool* of partial values and allocate parts from the pool to transactions when they execute so that when they are ready to commit, they are guaranteed to be able to. For example, if a transaction is going to decrement an account balance provided there are sufficient funds, it will hold the amount it wishes to decrement *in escrow*. Other transactions can also decrement the balance, as long as combined they will not leave the balance negative. If a transaction aborts, the escrowed values are returned to the pool.

Escrow can be extended to any *fragmentable object* [71], that is, any data type providing a way to *split* itself into fragments of the same type, and a way to later *merge* the fragments back together. This is essentially the inverse of the criteria for *monoids* used for aggregators in Summingbird [20], and similar to the isolation types in Concurrent Revisions §3.3.

Reservations can be thought of as escrow for replicated data. A reservation pre-allocates permission to do updates so that in the future they can be done without coordination. This moves coordination off the critical path and allows synchronization to be amortized when multiple updates share a reservation. For example, in Mobisnap [52] where they were first introduced, a salesman might reserve a number of tickets or a some quantity of a commodity, then while on the go, without connectivity, make a sale and know that it is safe to do so. Combining reservations with *leases* (discussed next in §4.2.1) allows them to be reclaimed automatically after a certain time has elapsed. Exo-leasing [62], not quite using *lease* the same way, further extends escrow and reservations to be decentralized and exchangeable among offline clients.

Reservations are easy to generalize — Indigo [15] uses them to implement its application-specific conflict avoidance logic that includes auto-generated code. A similar implementation

of numeric invariant preservation called *bounded counters* [16] was built on top of Riak.

Imbalance can be a problem when reservations are distributed: if some replicas receive more requests than others, they may use up all of their reserved updates before others do. In these situations, techniques such as the *demarcation protocol* [14, 17], or *handoff* [4] can allow replicas to redistribute permissions or re-balance per-replica limits. These techniques operate similar to *work-stealing* in Cilk [3].

4.2. Bounded staleness

In theory, eventually consistent systems provide absolutely no guarantees during execution because there is no bound on the time it must take for updates to propagate, and there are almost no situations where updates are guaranteed not to occur. In practice, however, programmers typically observe very few actual consistency errors, even at large scale [47]. This is because the propagation time, or *inconsistency window* is typically very small, on the order of tens of milliseconds [10], so few accesses observe the gap. However, programmers cannot rely on these observations because they do not hold in all cases. High contention situations are particularly problematic because with more concurrent updates and accesses, the chances of observing inconsistencies is much higher, and the value is also likely to be further from the correct value.

4.2.1. Leases

The problem with reads is that by the time the client gets the result, it could already be out of date, even without the additional complexity introduced by eventual consistency. Leases are a way of communicating how old a read is, by associating it with a time in the future when it should be considered stale. First proposed for file system caches to avoid needing to send explicit invalidations [33], they are now used in application caches in modern datacenters, such as in Facebook's Memcache system [50]. In addition to simply bounding staleness without explicit invalidation, leases can be used to indicate a promise that the value will not be updated for some time.

Warranties [46] combine leases with reservations to grant permission for holders to perform specific updates for a fixed amount of time. This has the benefit that permissions are automatically reclaimed after the time has elapsed, even if the holder crashed, and saving a reply message if the recipient does not wish to use the warranty.

4.2.2. Probabilistically bounded staleness

In order to help programmers reason about staleness, Bailis et al. [10] introduced a metric called *probabilistically bounded staleness* (PBS) which quantifies the staleness of accesses, either in terms of *time* or *versions*. By observing the distributions of propagation delays, round trip times, and rate of updates, their implementation builds a model of the system and uses it to predict staleness during execution. They also discussed how, by choosing the number of replicas to send writes to or consult for reads, one can control staleness and suggested how PBS could be used to select the right balance.

4.2.3. Consistency-based SLAs

With *consistency-based SLAs* [69], programmers can explicitly trade off consistency for latency. A consistency SLA specifies a target latency and a consistency level (e.g. 100 ms with read-my-writes). In this programming model, operations specify a set of desired SLAs, each associated with a *utility*. Using a prediction mechanism similar to PBS, the Pileus system attempts to determine which SLA to target to maximize utility, typically to achieve the best consistency possible within a certain latency.

Allowing users to specify their desired latencies and consistencies directly to the system is powerful. However, because it is so fine-grained, the burden of choosing target latencies and

```
sla = ShoppingCartSLA

# add item to cart
c.put("bob:cart:7", "Star Wars T-Shirt")

# get current cart
cart, consistency = c.get("bob:cart:*", sla)
```

Shopping Cart SLA

Consistency	Latency	Utility
Strong	300 ms	1.0
Eventual	300 ms	0.5

Figure 4. Example Consistency-based SLA: Shopping cart.

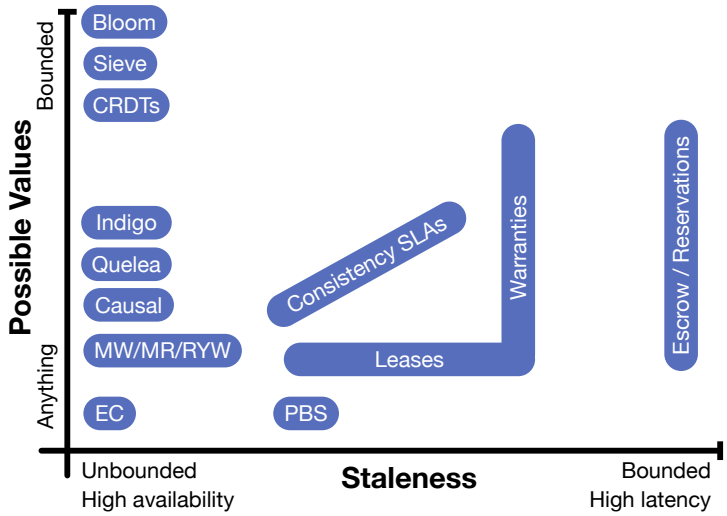


Figure 5. Bounding *uncertainty* in terms of staleness and restricting values.

consistency for each operation could be quite high, and it seems difficult to compose a sequence of operations and SLAs to achieve an overall target latency or correctness criteria.

4.3. Review: Uncertainty

Figure 5 maps out the techniques we have covered along two axes: how much they bound staleness versus how they restrict the set of allowable values. Again, eventual consistency provides the least guarantees. In fact, most of the techniques covered earlier do not affect *staleness* hardly at all. Instead, stronger consistency models and the annotation-based techniques restrict possible values by eliminating conflicts. The major differentiator comes with the switch to using CRDTs to restrict operations to those supported for each data type, like with ADTs. Bloom, by restricting programs to monotonic transformations over CRDTs, has the most bounded values.

Escrow and reservations can be very useful for bounding the uncertainty of replicated data. They allow hard bounds to be enforced without preventing parallelism in most cases. Consider again the Star Wars ticket sales example from §1. In order to handle the high load, we could replicate the tickets for this movie and allow clients to purchase tickets from any replica, but then we would not be able to prevent the same ticket from being sold to two different users. Using the concept of *escrow*, however, we can divide the tickets among the replicas, allowing clients to purchase them in parallel while there are many remaining, yet preventing tickets from over-selling when they begin to run out. We will discuss this more in §7.

Along the axis of *staleness*, PBS provides additional knowledge, but little control. Leases, on the other hand, allow for a range of information about staleness to be conveyed, and warranties

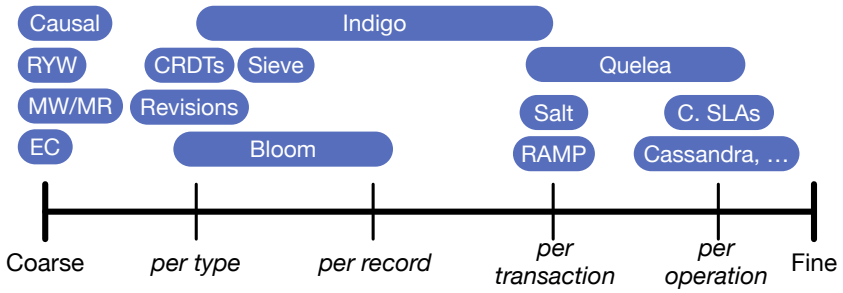


Figure 6. Granularity of various techinques. What is the best way for trading off consistency? For programmability?

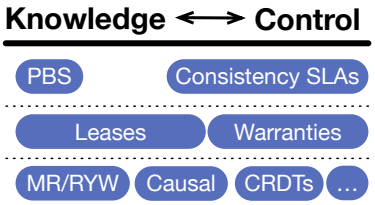


Figure 7. Which techniques provide additional knowledge (about uncertainty/staleness/visibility), and which give control over these properties?

provide additional control over how values change by controlling permission to perform updates. Consistency SLAs allow latencies and consistency levels to be traded off, giving control over both axes of uncertainty to applications.

5. Programming model comparison

We have now discussed many techniques for balancing the tradeoffs between consistency and performance in terms of how they manipulate constraints and bounds. However, the various programming models also differ how they expect programmers to express their desires and what control they provide. What is the best granularity to express the high-level requirements of an application to the system? Figure 6 organizes the programming models by levels of granularity: from coarse-grained full-system models to fine-grained per-operation control. Figure 7 categorizes the techniques by whether they provide additional knowledge about the system or whether they give explicit control to programmers.

We previously mentioned that consistency models are prohibitively coarse-grained, especially considering that contention is the exceptional case and that most transactions are not problematic for performance.

At the opposite extreme, some datastores, such as Cassandra, allow choosing a consistency level for each operation. This places significant burden on programmers to select the level that is right for each use case: being too strict will hurt performance, being too weak will result in errors. Moreover, this operation-centric approach is not modular, as choices must be reconsidered each time a new operation is introduced that may interleave.

CRDTs and isolation types from Concurrent Revisions occupy another spot in the granularity space – they specify constraints in terms of *types*. This allows them to couple consistency with state – a data-centric approach. Indigo and Bloom associate application-level invariants with types.

High-level approaches – Indigo, Quelea, Sieve, and Bloom – automate the process of analyzing where coordination must occur and selecting appropriate synchronization to ensure correctness. This automation relieves programmers of the error-prone task of selecting fine-grained consistency levels and allows them to better span the constraint axes. Automation aids *modularity* as well – synthesized coordination rules can be adjusted whenever new functionality is added. In order to do their job, these systems must know about the *abstract state* and *behavior* (together we will refer to these as *semantics*) which make up the application. In Indigo, semantics are exposed *manually* in pre- and post-condition annotations; similarly in Blazes. In Bloom, the semantics are implicit in the monotonic structures used to build the application, so no additional annotation is required.

Expressing abstract state and behavior *by construction*, as in Bloom, is powerful. It is less burdensome and error-prone than manual annotation and more reusable. We have chosen to focus on *abstract data types* because they support this by-construction expression of semantics but are more natural for programmers to use than Bloom's statements and facts. As with CRDTs, there may need to be many variants of data types in order to capture the desired consistency/performance properties desired by each application, but ADTs still allow significant re-use.

In the remaining sections of this paper we will show how we have used ADTs to express application semantics, describe prior techniques of leveraging ADTs, and show how we have extended them.

6. Mitigating Contention with Abstract Data Types

In order to perform contention mitigation techniques, the system must have knowledge of the desired application semantics. Our solution to this problem is an old one: *abstract data types* (ADTs). The core idea of ADTs is to present a clear *abstract model of state and behavior*, while hiding all the implementation details. With ADTs, applications describe their semantics to the underlying system *by construction*, allowing it to take advantage of properties, such as commutativity, to reduce coordination, avoid conflicts, and improve performance. All the details of ordering constraints, visibility, and coordination can be hidden behind the abstraction of data types with well-defined behavior.

ADTs are a natural interface for developers to express application semantics. They understand how a `Set` ADT behaves, and the system knows from a specification like Table 2 under which circumstances operations commute, or how to fragment the type for escrow. Programmers can maximize the optimizations available to the system by selecting the most specific ADT for their situation. For instance, incrementing a generic `Counter` must return the next number, but a `UniqueIDGenerator` lifts that restriction and so can generate non-sequential IDs in parallel. Programmers can even provide their own application-specific ADTs or customize existing ones to make them more suitable.

The concept of ADTs has long been used to extend databases: supporting indices and query planning for custom data types [65, 66], and concurrency control via abstract locks [9, 25, 36, 72]. Today's distributed systems deal with new challenges, and have evolved the many techniques described above to solve them. Unfortunately, many of the lessons learned about the benefits of the ADT abstraction were not carried forward into modern distributed systems. Many were lost in the move from relational databases to "NoSQL" datastores. We are just now beginning to figure out how to leverage type-level semantics in systems with weak replication through CRDTs and Bloom.

My work has pushed for the use of ADTs to allow systems to better mitigate the capricious, high-contention situations that cause so much trouble to applications. In Grappa, a high-performance system for irregular data analytics, we used *combining* to improve throughput on globally shared data structures. In *Claret*, we showed how ADTs can be used to improve performance of distributed transactions. Finally, I will propose *Disciplined Inconsistency*, a way to safely trade off consistency for performance with approximate ADTs.

6.1. Combining with global data structures in Grappa

Grappa [49] is a system we built for irregular data analytics. In order to tolerate the latency of communicating between nodes in commodity clusters, Grappa requires significant concurrency. Luckily, applications like graph analytics typically have abundant data parallelism that can be exploited. Such applications often require shared data structures to store the data itself (such as a graph), collect intermediate results, and support the underlying runtime. Because of the massive number of parallel threads needed for latency tolerance, these shared, distributed data structures are a source of significant contention. Naive locking strategies, even fine-grained, result in excessive serialization, preventing these data structures from being used as intended.

Combining [34, 61, 75] is a technique that can reduce contention on shared data by distributing synchronization. Basically, combining exploits the *associativity* of some ADT operations, merging them together into a single combined operation before applying the combined operation on the shared data structure. For example, individual `Set.add` operations can be combined into a single operation that adds multiple elements. Doing so moves some of the synchronization off of the hot data structure – now several separate synchronizations are just one. This is useful because combining can be done in parallel on many different threads. In some situations, operations even *annihilate* one another – that is, they cancel each other out, as is the case with a `push` and `pop` to a stack – which eliminates any need for global coordination of those particular operations.

Combining has been used in many different shared-memory systems to reduce contention and data movement. In a similar way, MapReduce allows a *combiner* to be defined to lift part of the reducer's work into the mapper [29]. We applied the concept to Grappa's distributed shared data structures and observed significant performance improvements [38]. In the distributed setting, combining can be even more effective as local synchronization within a node can eliminate many costly round-trip communications to other nodes. We also developed an extensible framework for developing data structures with combining for Grappa applications.

6.2. Claret: abstract data types for high-contention transactions

One of the most popular key/value stores in use today is Redis [55], which is special in that it supports a much wider range of complex data types and many operations specific to each type. However, Redis does not support general distributed transactions because they are considered too expensive. We observed that by treating Redis's data types as ADTs, we could expose significantly more concurrency between transactions to make them practical even for high-contention workloads. One technique crucial to this is *abstract locks*.

6.2.1. Abstract locks

Databases commonly use *reader/writer locks* to control access to records in conjunction with a protocol such as two-phase locking to ensure isolation between transactions. With reader/writer locks, multiple readers can hold the lock at the same time because they do not modify it, but anyone wishing to perform mutation must hold an *exclusive* writer lock. *Abstract locks* [9, 25, 36, 58, 72] generalize this notion to any operations which can logically run concurrently on the same object. Abstract locks are defined for a particular ADT in terms of a *commutativity specification* which describes with pairs of operations commute with one another: a function of the methods, arguments, return values, and abstract state of their target. An example specification for a `Set` is shown in Table 2.

When used in the context of transactions (termed *transaction boosting*), abstract locks can drastically reduce conflicts by allowing more operations to execute concurrently on the same record [35]. This is particularly crucial for highly contended records, where the chances of having concurrent operations is high, and serializing operations can become a bottleneck. In essence, abstract locks allow transactions to overlap more, only serializing when they absolutely must in order to ensure they cannot observe inconsistent state.

Method	Commutes with	When
`add(x): void`	`add(y)`	$\forall x, y$
`remove(x): void`	`remove(y)`	$\forall x, y$
	`add(y)`	$x \neq y$
`size(): int`	`add(x)`	$x \in Set$
	`remove(x)`	$x \notin Set$
`contains(x): bool`	`add(y)`	$x \neq y \vee y \in Set$
	`remove(y)`	$x \neq y \vee y \notin Set$
	`size()`	$\forall x$

Table 2. Abstract Commutativity Specification for Set.

6.2.2. Claret

Our prototype ADT-store, *Claret*, has a similar programming model to Redis. Underneath the abstraction afforded by the ADTs, Claret implements abstract locks, combining, and a form of lock reordering called phasing. On three transactional workloads simulating realistic contention – a microbenchmark similar to YCSB+T [30], an online auction service [7], and a Twitter-like social network [56] – Claret achieved a 3-50x speedup over naive transactions and within 67-82% of the performance without transactions.

6.3. Reveling in the bounty of inaccuracy

The ADTs used in Grappa and Claret exposed concurrency without sacrificing safety or correctness. This imposed a limit to the amount of concurrency they could exploit. We compared the performance of Claret's transactions against the same workloads without transactions. Though Claret's transactions were competitive, they fundamentally could not out-perform the non-transactional workload because they could not allow conflicting operations to be executed concurrently. For example, no matter how much commutativity abstract locks could expose, `Bid` and `ViewAuction` transactions had to be separated because `ViewAuction` required viewing the current maximum bid. What if we could relax this requirement and allow clients to view inaccurate results? What can be done to make this as safe as possible?

As we have discussed throughout this paper, there are significant performance benefits to be had by relaxing consistency and exposing weak replication. To give developers full control over these opportunities, we must allow them to trade off consistency in their applications. Our next project proposes to do just that by defining a new class of ADTs.

7. Disciplined Inconsistency

Our goal is to come up with a programming model that helps programmers balance all of these competing requirements; ideally, it should have the following properties:

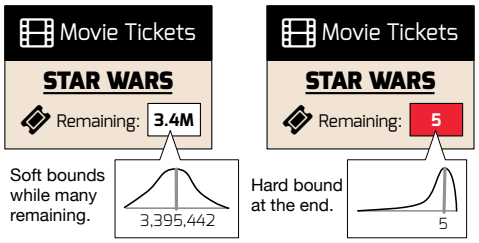
- Minimize unnecessary constraints by exposing safe concurrency.
- Express where and what errors can be tolerated.
- Communicate performance requirements such as target latency or availability.
- Be easy to reason about and modular.

At this point we have well established that trading off consistency for performance is tricky business, involving making many decisions about what reorderings of operations should be allowed, when updates must be visible in order to ensure correct execution, or how consistent a read can be and still meet its latency SLA. Furthermore, programmers must make these decisions while keeping in mind that due to real-world effects, some data items will be significantly

```
// Pool supporting unique `take`s
// bounded by initial size, and
// reading an approximate size.
template< typename T,
          Time Latency >
class Pool : IPAType {

  T take();

  // return the range of possible
  // current sizes
  Range<int> size();
};
```




```
// derive app-specific type from generic IPA type
using MovieTickets = Pool<TicketID, Millis(100)>;
```

Figure 8. A `MovieTicket` ADT can be implemented using a more generic `Pool` type. Ticket sales must be strongly consistent, but the number of remaining tickets can be approximate, provided it gets more precise when there are few remaining.

more contentious and inconsistent than others. The promise of ADTs is to hide implementation details – can we use them to hide some of these concerns?

Yes! *Inconsistent, Probabilistic and Approximate (IPA) types* can express weaker constraints on ADTs, unlocking the possibility of using the techniques we have discussed to trade off consistency for performance. Operations on IPA types only allow views of the state that can assure correct semantics, which may mean disallowing operations that would expose too much, or exposing values that encode a range or distribution rather than a precise value. IPA types encapsulate the ordering and visibility constraints necessary for whatever operations they allow, so users do not need to interact with complex consistency models, and these types can subsequently be composed. Of course, it leaves programmers with a different set of problems to deal with, which we will get to, but which we posit are a better alternative.

Let us consider again the scenario posed at the beginning, about a ticket sales app that failed to handle the load when the new Star Wars movie came out. Its programmers likely had several requirements in mind for the app's behavior:

1. Do not sell more tickets than are available.
2. Let users see an estimate of how many tickets remain when they load the page.
3. Initial page load must have a 99th percentile latency of 100 ms.

If we allow two replicas to both sell the last ticket, we will violate the first requirement, so our model must express this as a hard constraint. Using *escrow*, we can distribute permissions to sell tickets among the replicas so that while there are many tickets remaining, the requests can be handled by any replica safely. However, now providing a precise count of remaining tickets requires synchronizing all replicas. Luckily, (2) tells us that the count does not need to be precise, which allows us to meet (3)'s latency requirement by accessing whichever replica is nearest or fastest.

Figure 8 shows a possible way of specifying this movie ticket sale as an IPA type. In this example, we derive our `MovieTickets` type from a generic `Pool` that ensures `take` operations are unique and upper-bounded by the number of items in the pool and supports reading an approximate size. Without some form of bound, the approximate size is somewhat underspecified. In this case the `MovieTickets` type requested a bounded latency of 100 milliseconds, which serves to determine how the size will be obtained and how approximate it will be. Alternatively, we could imagine a different application wishing to instead specify a target value bound, such as a maximum 10% error. We address this duality between performance and precision next.

7.1. Duel of duals

Consistency and performance are coupled – more consistency requires coordination which costs performance; achieving performance targets may require sacrificing consistency. We have covered many techniques that acknowledge this tradeoff and allow applications either to specify weaker consistency in order gain performance or specify coordination requirements and give up some performance. However, only Pileus's consistency-based SLAs [69] made performance targets explicit and provided *feedback* to programs about the consistency achieved.

It is common today for the specifications for applications to include *performance bounds*, such as target latencies specified by an SLA, or the requirement that a service be *highly available*. More strict performance bounds imply more uncertainty in terms of values – consistency is weaker, reads are forced to take whatever they can get, even if it is stale. Because it is not surfaced explicitly, this increase in uncertainty can go unaddressed, leading to noticeable consistency issues later. Instead, can we make these uncertainties explicit in the programming model, ensuring programmers handle them correctly?

In other situations, perhaps only a certain amount of error can be tolerated. For example, when viewing a tweet with only a few retweets, one would expect the count to be exact, because being off by even 1 would be obvious. However, when viewing a super popular tweet, like a Justin Bieber selfie, the number of retweets could be in the millions, so it can be off by thousands. In these situations, it would be useful to be able to specify an *error tolerance*, such as a 5% tolerance on the count. As the dual of performance-bounded operations, perhaps the programming model ought to provide estimates of the *performance uncertainty* for operations with bounded error.

In this work, we aim to provide all of these as options so that no matter the situation, programmers have the tools they need to make those tradeoffs.

As an aside, the goals of BlinkDB [2] bear a lot of similarity to ours, except they trade off the amount of data touched rather than consistency. In BlinkDB, SQL queries with aggregates (such as `count` or `average`) are annotated with either *time bounds* or *error bounds*, and the database uses sampling to get the best answer it can within the bounds. Their approach requires partial knowledge of the distribution and maintenance of "stratified samples" in order to be able to estimate the error with confidence. We will need different approaches to estimate error under weak consistency.

7.2. Programming model

What kinds of approximations do we need to be able to express? How do they manifest themselves as ADTs? In the *Disciplined Inconsistency* programming model, bounds are defined on ADTs, and operations return *IPA types*.

7.2.1. Specifying bounds on ADTs

There are several forms of constraints which we will aim to support. These include some of the constraints supported by Indigo [15] that are compatible with ADTs, extended with new hard and *soft* constraints. We expect bounds to typically be expressed on application-specific ADTs, as in Figure 8 and Figure 9.

Numeric constraints. These can apply either to simple standalone numeric values, or more commonly to integer quantities associated with data structures, such as the cardinality of a Set. Constraints can be hard upper or lower bounds, or a *tolerance* of some distance from the precise value (such as the 10% tolerance we mentioned earlier). *Uniqueness*, a common desirable constraint [13, 15], is a degenerate case with an upper bound of 1.

Filter or membership constraints. Akin to bloom filters, applications may wish to ensure with some probability that a set contains all of the correct values. Using approximate reservations or PBS, the system could have some idea of whether any new items had been added

to a container ADT. Applications wishing to use even more fine-grained "filters" could specify that items fitting a certain description should be handled differently. For example, Facebook may wish to guarantee that close friends' posts appear consistently on a user's timeline, while others can be opportunistic.

Latency bound. To meet latency SLAs, operations can have specified latency bounds. Using a mechanism like Pileus's predictive monitors, the system could choose which replicas to use in order to meet certain latency requirements. These bounds cannot be hard bounds, so, like real SLAs, they would typically be associated with a target percentile.

Availability requirement. If a situation demands high availability, regardless of cost, then it could use a constraint like this to ensure that the system returns a result from any available replica without mediation.

7.2.2. IPA Types

Most operations with a performance or value bound will return a value with some form of uncertain type, depending on how the bound is fulfilled. These types fall into 3 different categories: *inconsistent*, *probabilistic*, or *approximate*.

Approximate types. These types are the simplest for programmers to use, and should be preferred. Approximate types encode the set of all possible correct values. Note that in situations of replication, there is likely no single globally *correct* value – different replicas can hold different values simultaneously, and there could be any number of staged operations whose final commit order is yet unknown. Example approximate types include `Interval<T>`, which specifies an upper and lower bound on possible types. Any type with a defined *partial order over values* (a lattice) can be represented as an interval: numeric types are obvious, but a *set*, for example, could have an interval defined by a number of items that may or may not be in the set. An interval could also be defined as a *point* and *radius* – for example, a mean ± 5% – depending on the desired semantics.

Probabilistic types. In some situations, guaranteeing the hard bounds encoded by approximate types is too expensive. Bounds can be weakened by providing a probabilistic guarantee instead. These typically take the form of a distribution, such as a gaussian defined by a mean and standard deviation, with a certain confidence level. For example, PBS's models do not ensure hard bounds on staleness but rather a probability distribution that defines whether it is correct or not. We expect programmers to use these types for simple inference questions, such as "Is the value greater than 100 with 95% confidence?", in order to determine if something should be displayed. This is closely related to how `Uncertain<T>` [19] allows programmers to reason about uncertain values coming from sensors.

Inconsistent types. Finally, in the worst cases, there may be no way to bound the potential values. For instance, any technique providing hard bounds may need to limit the rate at which updates are applied to replicas. If absolute maximum availability or throughput is required, then those restrictions cannot be applied. However, we can still help programmers be *disciplined* about these unsafe cases using *inconsistent types*. These types are the most opaque. At best, they may provide a measure of staleness, such as a `Stale<T>`, which would allow users to know how out of date the value is, and potentially surface this to the user. An example of this is shown in Figure 9. However, even a nearly completely opaque `Inconsistent<T>` can help protect inconsistent values from accidentally flowing into consistent computations, similar to how *taint analysis* prevents secure values from being leaked.

7.3. Implementation

Our implementation involves developing the frontend programming model – IPA types with value and performance bounds – and a backend *enforcement system* which integrates many of the prior techniques covered in this document.

7.3.1. Enforcement system

The prior techniques discussed in this document provide ample ways to enforce invariants and perform efficient coordination. We will focus primarily on *escrow*, *reservations*, and *leases*, integrated with ideas from *abstract locks* to implement most of the coordination required for *approximate types*. Indigo [15] demonstrated one way of combining abstract locks ("multi-level locks" in their terminology) with reservations to implement the wide range of constraints they supported. For instance, numeric constraints can be implemented using *escrow* to distribute permissions among all the replicas. For example, to implement our `Pool` type from Figure 8, we could create an escrow *piece* for each ticket ahead of time, then distribute those among the replicas. When clients execute a `take`, a piece is allocated from the pool of remaining operations. If the `take` aborts before finishing, the piece is simply returned to the pool. Likewise if a replica goes offline, the pieces are not actually lost – they can be reclaimed after a period of time and re-used.

Another component of the enforcement system will behave similarly to PBS and Pileus's consistency SLA system, to provide *probabilistic* uncertainty. In these systems, hard guarantees are not enforced as with escrow reservations. Instead, this subsystem monitors various health metrics about the system: replication latency, write load, etc. These factors can then be fed into simple predictive models to make judgements about the probability that a given value is being updated currently.

Largely unmodified CRDTs provide the basis for data types that do not prevent conflicts. Operations on these types should return some form of IPA type – by default, just an `Inconsistent<T>`, but if version vectors or other staleness information is available, more expressive IPA types can be used.

7.3.2. Implementation strategy

As a first pass, the programming model will simply be hand-written data types supporting a small number of configurable latency and value bounds. These data types will at first manually specify how they are to be enforced using components from the enforcement system described above.

Once we have established the potential using manual implementation, we will investigate ways of automatically generating the correct enforcement given specified latency bounds and IPA types. Another avenue of potential investigation is to see if the bounds annotations can be used to infer the correct IPA types, or at least do static type checking to ensure that the interface is enforceable with the desired bounds.

7.4. Case studies

7.4.1. Twitter

Twitter is subject to all kinds of extreme contention resulting from realtime events, and power law effects. Famously, early in Twitter's lifetime, it would go down any time traffic spiked, such as during World Cup goals; each time showing the now famous Fail Whale [39]. Even late in its life, Twitter was slowed to a standstill at the 2014 Oscars when Ellen Degeneres tweeted a selfie with several celebrities which was retweeted at record-breaking speed. Luckily, there are many aspects of Twitter that are amenable to inconsistency. In fact, most of the real Twitter application is served out of their eventually consistent datastore, Manhattan [57]. However, if we can quantify specific places where accuracy is unnecessary, perhaps we will not need to give up consistency everywhere else.

```
# derive app-specific types from generic IPA types
using Timeline = SortedSet<ID,Range<Millis(100)>>;

using Retweets = Set<ID,Size<Tolerance(0.01)>>;

def timeline(user):
    # Timeline.range: 100 ms latency bound
    tweets: Stale<List<ID>> = Timeline(user).range(0, 10)

    # check if timeline may be missing tweets older than 5s
    if posts.older_than(Seconds(5)):
        display_warning("May be out of date.")

    for t in tweets:
        tweet: Map = Tweet(t).get()

        # Retweeters.size: 1% tolerance
        retweets: Range<Int> = Retweets(t).size()

        display_tweet(tweet, retweets)
```

Figure 9. *Twitter clone:* Loading a user's timeline with IPA types.

Followers, retweets, favorites counts. As mentioned before, for an extremely popular tweet, these counts are truncated – Justin Bieber's profile lists that he has "69.8M" followers rather than the precise value, because it is constantly changing (but not often by more than 100,000…), and his most recent tweet was retweeted "20K" times. On the other hand, most tweets have few favorites, and if those counts are off they will be noticed. This is a perfect place for a tolerance-based constraint that will scale with the size of the count.

Missing tweets. Another possible relaxation is which tweets appear in a timeline. If a timeline is missing a recent tweet, odds are good the user will not be able to tell. This may be a situation where a performance bound is called for – no one wants to wait a long time for their timeline. This could then return a measure of the *staleness* of the timeline, such as the latest point in time before which tweets are guaranteed to be included.

For Claret, we implemented a simplified Twitter clone based on a Redis application called Retwis [56]. This application will be extended to use disciplined inconsistency to further improve performance and availability.

7.4.2. Auction

Auction services are generally considered a class of applications requiring strong consistency. Finding the correct maximum bid, in spite of any amount of traffic, is crucial for fairness. However, they also have significant contention problems as some auctions receive far more bids than the average (they follow a power law), and bidding ramps up right before the auction closes. Allowing as many bids as possible during that late stage is crucial to keeping users happy and maximizing revenue. We base our implementation on the functionality of Rubis [7].

Current high bid. While an auction is ongoing, users typically want to know what the going rate is so they can decide if they are willing to over-bid. If this is changing frequently, the high bid is going to change rapidly. At that point, users are unable to really tell what the current bid will be when they actually make their own bid. Therefore, they probably do not need to see the most precise version of the bid. Some estimate of the current maximum bid, provided it is reasonably accurate, is acceptable because the true high bid can still be found later. This is an ideal situation for a latency-bound operation that returns a range of possible values or a probability distribution.

Stale listings. Rubis allows clients to browse currently open auctions by region or category. In this view, one could imagine wanting to show an estimate of the price and other relevant

information about the auctions. However, it could be prohibitively expensive to get accurate, up-to-date bid information. Furthermore, in large deployments, popular categories could have many new auctions opening constantly. These both seem like situations where a stale version, either of the list of current auctions, or the current bids for each auction, would be acceptable.

7.4.3. Ticket Sales

We have used the movie ticket sales example throughout this paper, but to reiterate, there are a number of interesting challenges in supporting ticket sales. The movie ticket example is a bit contrived because no one theater will have all that many seats available, so contention will be distributed even for popular movies. However, ticket sales for other events with much larger venues can experience serious problems with contention. We already discussed how the *remaining tickets* field is a candidate for approximation.

FusionTicket [1] is an open source web application for selling tickets to events that has been used in some recent transactions research as a benchmark [73, 74]. We could use this benchmark as a starting point and look for additional places where contention is a problem and where error can be tolerated.

7.4.4. Streaming analytics

Trending topics, realtime recommendations, and performance monitoring are just a few examples of the kinds of analytics services like Twitter or Facebook perform continuously. These features can be crucial to user engagement. However, they can also typically be treated as a best-effort or "nice to have" addition to the experience; whenever performance becomes a problem, these features can be dialed back. Furthermore, they are typically informed by machine learning algorithms which introduce significant noise. Therefore, they are prime candidates for disciplined inconsistency.

One concrete example: Twitter's revamped streaming analytics platform, Heron [41], supports a feature called *backpressure*. When a downstream processing element becomes overloaded and is unable to keep up with the rate of incoming data, it will fall behind and no longer be *realtime*. Backpressure tells upstream data generators to dial back their output, typically done by sampling, until the backlog is under control and processing returns to normal. This can cause discontinuities in the output of analytics which may persist if programmers do not handle it correctly. Perhaps some form of probabilistic data type could be used to communicate the sampling rate of data items, making it simpler for programmers to handle these cases.

8. Conclusion

To navigate the treacherous seas of the Internet, today's distributed applications must be ready to cut through the waves of contention. Many capable techniques, like abstract locks, escrow, and consistency-based SLAs, are on hand, but the ship that holds them together is the abstraction afforded by ADTs. ADTs give programmers a powerful way to express application semantics by construction; IPA types allow them to trade off performance for precision in a disciplined way. Rather than battening down the hatches just to stay afloat, by adjusting the sails just right to handle the gusts, applications can sail on through the storm.

References

[1] Fusion ticket. http://fusionticket.org.

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: Queries with bounded errors and bounded response times on very

large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1994-2. doi:10.1145/2465351.2465355.

[3] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. Adaptive work stealing with parallelism feedback. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 112–120, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-602-8. doi:10.1145/1229428.1229448.

[4] Paulo Sérgio Almeida and Carlos Baquero. Scalable eventually consistent counters over unreliable networks. *CoRR*, abs/1307.3207, 2013. URL http://arxiv.org/abs/1307.3207.

[5] Peter Alvaro, Neil Conway, Joe Hellerstein, and William R Marczak. Consistency analysis in bloom: a calm and collected approach. In *Conference on Innovative Data Systems Research (CIDR)*, CIDR, pages 249–260. Citeseer, 2011.

[6] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. Blazes: Coordination analysis for distributed programs. In *IEEE International Conference on Data Engineering*. Institute of Electrical
& Electronics Engineers (IEEE), March 2014. doi:10.1109/icde.2014.6816639.

[7] Cristiana Amza, Anupam Chanda, Alan L. Cox, Sameh Elnikety, Romer Gil, Karthick Rajamani, Willy Zwaenepoel, Emmanuel Cecchet, and Julie Marguerite. Specification and implementation of dynamic web site benchmarks. In *2002 IEEE International Workshop on Workload Characterization*. IEEE, 2002. doi:10.1109/wwc.2002.1226489.

[8] Apache Software Foundation. Cassandra. http://cassandra.apache.org/, 2015.

[9] B. R. Badrinath and Krithi Ramamritham. Semantics-based concurrency control: beyond commutativity. *ACM Transactions on Database Systems*, 17 (1): 163–199, March 1992. doi:10.1145/128765.128771.

[10] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5 (8): 776–787, April 2012. doi:10.14778/2212351.2212359.

[11] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment*, 7 (3): 181–192, November 2013. ISSN 2150-8097. doi:10.14778/2732232.2732237.

[12] Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. Scalable atomic visibility with RAMP transactions. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD 14*. Association for Computing Machinery (ACM), 2014. doi:10.1145/2588555.2588562.

[13] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Feral concurrency control. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD 15*. Association for Computing Machinery (ACM), 2015. doi:10.1145/2723372.2737784.

[14] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Towards fast invariant preservation in geo-replicated systems. *ACM SIGOPS Operating Systems Review*, 49 (1): 121–125, January 2015a. doi:10.1145/2723872.2723889.

[15] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys, pages 6:1–6:16, New York, NY, USA, 2015b. ACM. ISBN 978-1-4503-3238-5. doi:10.1145/2741948.2741972.

[16] Valter Balegas, Diogo Serra, Sergio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguiça. Extending eventually consistent cloud databases for enforcing numeric invariants. *34th International Symposium on Reliable Distributed Systems (SRDS 2015)*, September 2015c.

[17] Daniel Barbará-Millá and Hector Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal*, 3 (3): 325–353, July 1994. doi:10.1007/bf01232643.

[18] Basho Technologies, Inc. Riak. http://docs.basho.com/riak/latest/, 2015.

[19] James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. Uncertain<T>: A First-Order Type for Uncertain Data. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS 14*, ASPLOS. Association for Computing Machinery (ACM), 2014. doi:10.1145/2541940.2541958.

[20] Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. Summingbird: A framework for integrating batch and online mapreduce computations. *Proceedings of the VLDB Endowment*, 7 (13): 1441–1451, August 2014. ISSN 2150-8097. doi:10.14778/2733004.2733016.

[21] Eric A. Brewer. Towards robust distributed systems. In *Keynote at PODC (ACM Symposium on Principles of Distributed Computing)*. Association for Computing Machinery (ACM), 2000. doi:10.1145/343477.343502.

[22] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications - OOPSLA'10*, OOPSLA. Association for Computing Machinery (ACM), 2010. doi:10.1145/1869459.1869515.

[23] Sebastian Burckhardt, Manuel Fahndrich, Daan Leijen, and Mooly Sagiv. Eventually consistent transactions. In *Proceedings of the 22n European Symposium on Programming (ESOP)*. Springer, March 2012a. URL http://research.microsoft.com/apps/pubs/default.aspx?id=158085.

[24] Sebastian Burckhardt, Manuel Fahndrich, Daan Leijen, and Benjamin P. Wood. Cloud types for eventual consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP)*, ECOOP. Springer, June 2012b. URL http://research.microsoft.com/apps/pubs/default.aspx?id=163842.

[25] Panos K. Chrysanthis, S. Raghuram, and Krithi Ramamritham. Extracting concurrency from objects. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data (SIGMOD'91)*, SIGMOD. Association for Computing Machinery (ACM), 1991. doi:10.1145/115790.115803.

[26] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing - SoCC 12*, SoCC. ACM Press, 2012. doi:10.1145/2391229.2391230.

[27] Natacha Crooks, Nancy Estrada, Lorenzo Alvisi, and Allen Clement. Tardis: Transactional storage with parallel worlds. Technical Report 2186, University of Texas at Austin, January 2015. URL http://apps.cs.utexas.edu/tech_reports/reports/tr/TR-2186.pdf.

[28] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56 (2): 74, February 2013. doi:10.1145/2408776.2408794.

[29] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51 (1): 107–113, 2008.

[30] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Rohm. YCSB+T: Benchmarking web-scale transactional databases. In *IEEE International Conference on Data Engineering Workshops (ICDEW)*, March 2014. doi:10.1109/icdew.2014.6818330.

[31] Dieter Gawlick and David Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Database Engineering Bulletin*, 8 (2): 3–10, 1985.

[32] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33 (2): 51, June 2002. doi:10.1145/564585.564601.

[33] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP. Association for Computing Machinery (ACM), 1989. doi:10.1145/74850.74870.

[34] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 355–364. ACM, 2010.

[35] Maurice Herlihy and Eric Koskinen. Transactional Boosting: A Methodology for Highly-concurrent Transactional Objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 207–216, 2008. ISBN 978-1-59593-795-7. doi:10.1145/1345206.1345237.

[36] Maurice P. Herlihy and William E. Weihl. Hybrid concurrency control for abstract data types. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS, pages 201–210, New York, NY, USA, 1988. ACM. ISBN 0-89791-263-2. doi:10.1145/308386.308440.

[37] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12 (3): 463–492, July 1990. doi:10.1145/78969.78972.

[38] Brandon Holt, Jacob Nelson, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Flat combining synchronized global data structures. In *International Conference on PGAS Programming Models (PGAS)*, PGAS, October 2013. URL http://sampa.cs.washington.edu/papers/holt-pgas13.pdf.

[39] Mat Honan. Killing the fail whale with twitter's christopher fry. http://www.wired.com/2013/11/qa-with-chris-fry/, November 2013.

[40] Eric Koskinen and Matthew Parkinson. The push/pull model of transactions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, PLDI. Association for Computing Machinery (ACM), 2015. doi:10.1145/2737924.2737995.

[41] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi:10.1145/2723372.2742788.

[42] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28 (9): 690–691, September 1979. doi:10.1109/tc.1979.1675439.

[43] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32, 2001.

[44] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, Hollywood, CA, 2012. USENIX. ISBN 978-1-931971-96-6. URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li.

[45] Cheng Li, Joao Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 281–292, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-10-2. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_2.

[46] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 503–517, Seattle, WA, April 2014. USENIX Association. ISBN 978-1-931971-09-6. URL https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/liu_jed.

[47] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: Measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP, pages 295–310, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3834-9. doi:10.1145/2815400.2815426.

[48] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP. Association for Computing Machinery (ACM), 2013. doi:10.1145/2517349.2517350.

[49] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Brigg, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, July 2015. URL http://sampa.cs.washington.edu/papers/grappa-usenix-2015.pdf.

[50] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX. ISBN 978-1-931971-00-3. URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

[51] Patrick E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11 (4): 405–430, December 1986. doi:10.1145/7239.7265.

[52] Nuno Preguiça, J. Legatheaux Martins, Miguel Cunha, and Henrique Domingos. Reservations for conflict avoidance in a mobile database system. In *Proceedings of the 1st international conference on Mobile systems, applications and services - MobiSys 03*, MobiSys. Association for Computing Machinery (ACM), 2003. doi:10.1145/1066116.1189038.

[53] Dan Pritchett. Base: An acid alternative. *Queue*, 6 (3): 48–55, May 2008. ISSN 1542-7730. doi:10.1145/1394127.1394128.

[54] Andreas Reuter. *Concurrency on high-traffic data elements*. ACM, New York, New York, USA, March 1982.

[55] Salvatore Sanfilippo. Redis. http://redis.io/, 2015a.

[56] Salvatore Sanfilippo. Design and implementation of a simple Twitter clone using PHP and the Redis key-value store. http://redis.io/topics/twitter-clone, 2015b.

[57] Peter Schuller. Manhattan, our real-time, multi-tenant distributed database for twitter scale. https://blog.twitter.com/2014/manhattan-our-real-time-multi-tenant-distributed-database-for-twitter-scale, April 2014.

[58] Peter M. Schwarz and Alfred Z. Spector. Synchronizing shared abstract types. *ACM Trans. Comput. Syst.*, 2 (3): 223–250, August 1984. doi:10.1145/989.1188.

[59] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS, pages 386–400, 2011. ISBN 978-3-642-24549-7.

[60] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction chopping: algorithms and performance studies. *ACM Transactions on Database Systems*, 20 (3): 325–363, September 1995. doi:10.1145/211414.211427.

[61] Nir Shavit and Asaph Zemach. Combining funnels: A dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60 (11): 1355–1387, 2000.

[62] Liuba Shrira, Hong Tian, and Doug Terry. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In *Middleware 2008*, Middleware, pages 42–61. Springer Science $$ Business Media, 2008. doi:10.1007/978-3-540-89856-6_3.

[63] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2015*, PLDI. Association for Computing Machinery (ACM), 2015. doi:10.1145/2737924.2737981.

[64] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *ACM Symposium on Operating Systems Principles - SOSP'11*, SOSP. Association for Computing Machinery (ACM), 2011. doi:10.1145/2043556.2043592.

[65] Michael Stonebraker. Inclusion of new types in relational data base systems. In *Proceedings of the Second International Conference on Data Engineering, February 5-7, 1986, Los Angeles, California, USA*, pages 262–269, 1986.

[66] Michael Stonebraker, W. Bradley Rubenstein, and Antonin Guttman. Application of abstract data types and abstract indices to CAD data bases. In *Engineering Design Applications*, pages 107–113, 1983.

[67] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *ACM Symposium on Operating Systems Principles - SOSP'95*, SOSP. Association for Computing Machinery (ACM), 1995. doi:10.1145/224056.224070.

[68] 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 *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, PDIS. Institute of Electrical & Electronics Engineers (IEEE), 1994. doi:10.1109/pdis.1994.331722.

[69] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP 13*. ACM Press, 2013. doi:10.1145/2517349.2522731.

[70] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52 (1): 40, January 2009. doi:10.1145/1435417.1435432.

[71] G.D. Walborn and P.K. Chrysanthis. Supporting semantics-based transaction processing in mobile database applications. In *Proceedings. 14th Symposium on Reliable Distributed Systems*. Institute of Electrical
& Electronics Engineers (IEEE), 1995. doi:10.1109/reldis.1995.518721.

[72] W. E. Weihl. Commutativity-based Concurrency Control for Abstract Data Types. In *International Conference on System Sciences*, pages 205–214, 1988. ISBN 0-8186-0842-0.

[73] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining acid and base in a distributed database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 495–509, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/xie.

[74] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-Performance ACID via Modular Concurrency Control. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP, pages 276–291, 2015. ISBN 978-1-4503-2388-8. doi:10.1145/2517349.2522729.

[75] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, 100 (4): 388–395, 1987.

[76] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP, pages 276–291, 2013. ISBN 978-1-4503-2388-8. doi:10.1145/2517349.2522729.