Claret: Avoiding Contention in Transactions with Abstract Data Types

Brandon Holt, Irene Zhang, Dan Ports, Mark Oskin, Luis Ceze
University of Washington

Abstract

The high degree of writes caused by interactive applications like Twitter or eBay, combined with the skewed access patterns exhibited by real-world systems, lead to contention in datastores that inhibit scalability and can cause periods of diminished service or even catastrophic failure. There ought to be sufficient concurrency in these applications to scale without resorting to weaker consistency models, but traditional concurrency control, operating on low level operations, cannot detect it.

We describe the design and implementation of Claret, a Redis-like data structure store which allows high-level application semantics to be communicated through abstract data types (ADTs). Using this abstraction, Claret is able to avoid unnecessary conflicts and reduce synchronization, while programmers continue to implement applications easily using whatever data structures are natural for their use case. Claret is the first datastore to use ADTs to improve performance of distributed transactions; optimizations include transaction boosting, phasing, and operation combining. On a transaction microbenchmark, Claret's ADT optimizations increase throughput by up to 49x over the baseline concurrency control and even up to 20% better than without transactions. Furthermore, Claret improves peak throughput on benchmarks modeling real-world high-contention scenarios: 4.3x speedup on the Rubis auction benchmark, and 3.6x on a Twitter clone, achieving 67-82% of the non-transactional performance on the same workloads.

1. Introduction

Today's online ecosystem is a dangerous place for interactive applications. Memes propagate virally through social networks, blogs, and news sites, inundating web services under heavy load without warning. In February 2015, a picture of a black and blue dress exploded across the internet as everyone debated whether or not it was actually white and gold, bringing unprecedented traffic spikes to BuzzFeed [32], the site responsible for sparking the viral spread. Even in its 8th year of dealing with unpredictable traffic, Twitter briefly fell victim in 2014 after Ellen Degeneres posted a selfie at the Oscars which was retweeted at a record rate [7].

These traffic spikes are the product of a number of factors in real world systems such as power law distributions and live events. The increasing interactivity of modern web applications results in significant contention due to writes in datastores. Even content consumption generates write traffic as providers track user behavior to personalize their experience, target ads, or collect statistics [9].

To avoid catastrophic failures and mitigate poor tail behavior, significant engineering effort must go into handling these challenging high-contention scenarios. Writes are such a problem because they impose ordering constraints requiring synchronization in order to have any form of consistency. Luckily, many of these orderings are actually irrelevant from the perspective of the application. For example, it is not necessary to keep track of the order in which people retweeted Ellen's selfie.

One way to avoid constraints is to use eventual consistency, but then applications must deal with inconsistent data, especially in high-contention cases. If the system knew which constraints were relevant to the application, then it could expose significantly more concurrency, allowing it to handle writing spikes without sacrificing correctness. Databases and distributed systems have long used properties such as commutativity to reduce coordination and synchronization, but they rely on having a predefined schema or restricting operations to only those that are coordination-free. This is particularly expensive in distributed systems where coordination and locking across a number of machines is required.

In this work, we propose a new way to leverage *abstract data types* (ADTs) to reduce synchronization in distributed transactions. ADTs allow users and systems alike to reason about their logical behavior, including algebraic properties like commutativity, rather than the low-level operations used to implement them. The datastore can leverage this higher-level knowledge to avoid conflicts, allowing transactions to interleave and execute concurrently without changing the observable behavior. Programmers benefit from the flexibility and expressivity of ADTs, reusing data structures from a common library or implementing custom ADTs for their specific use case.

Our prototype ADT-store, *Claret*, demonstrates how ADT awareness can be added to a datastore to make strongly consistent distributed transactions practical. It is the first non-relational system to leverage ADT semantics to reduce conflicts between distributed transactions. Rather than requiring a relational data model with a fixed schema, Claret encourages programmers to use whatever data structures naturally express their application.

Datastores supporting complex datatypes and operations are already popular. Many [5, 43] support simple

collections such as *lists*, *sets*, and *maps*, and even custom objects (e.g. protocol buffers). Redis [33], one of the most popular key/value stores, supports a large, fixed set of complex data types and a number of operations specific to each type. Currently, these datastores treat data types as just blackboxes with special update functions.

Claret uses the logical properties of data types to communicate application-level semantics to the system so it can perform optimizations on both the client and server side. In §4, we show how commutativity can be used to avoid false conflicts (*boosting*) and ordering constraints (*phasing*), and how associativity can be applied to reduce load on the datastore (*combining*).

On high-contention workloads, the combined optimizations achieve up to a 49x improvement in peak transaction throughput over traditional concurrency control on a synthetic microbenchmark, up to 4.3x on an auction benchmark based on Rubis [4], and 3.6x on a Twitter clone based on Retwis [34]. While Claret's optimizations help most in high-contention cases, its performance on workloads with little contention is unaffected. Claret's transactions achieve 67-82% of the throughput possible without transactions, which represents an upper bound on the performance of our datastore.

This work makes the following contributions:

- Design of an *extensible ADT-store*, Claret, with interfaces to express logical properties of new ADTs
- Implementation of optimizations leveraging ADT semantics: *transaction boosting*, *operation combining*, and *phasing*
- Evaluation of the impact of these optimizations on raw transaction performance and benchmarks modeling real-world contention

In this paper, we describe the design of the system and evaluate the impact ADT-enabled optimizations have on transaction performance. But first, we must delve more deeply into what causes contention in real applications.

2. Real world contention

Power laws are everywhere. Natural phenomena have a tendency to follow power law distributions: from Zipf's Law which observed that the frequency of words in natural language follows a power law, to the power-law degree distributions that cause low diameter networks (colloquially "six degrees of separation"). Network effects serve to amplify small signals into massive amounts of activity, such as when a meme goes viral. Finally, systems with a real-time component end up with spikes of activity as events occur in real life: in it's early days, goals during World Cup games famously caused Twitter to crash and show the "fail whale" [24].

To discuss this more concretely throughout the rest of this paper, we will use an eBay-like online auction service, based on the well-known RUBiS benchmark [4]. At its core, this service allows users to put items up for auction, browse auctions by region and category, and place bids on open auctions. While running, an auction service is subjected to a mix of requests to open and close auctions but is dominated by bidding and browsing actions.

Studies of real-world auction sites [1, 2, 29] have observed that many aspects of them follow power laws. First of all, the number of bids per item roughly follow Zipf's Law (a *zipfian* distribution). However, so do the number of bids per bidder, amount of revenue per seller, number of auction wins per bidder, and more. Furthermore, there is a drastic increase in bidding near the end of an auction window as bidders attempt to out-bid one another, so there is also a realtime component.

An auction site's ability to handle these peak bidding times is crucial: a slow-down in service caused by a popular auction may prevent bidders from reaching their maximum price (especially considering the automation often employed by bidders). The ability to handle contentious bids directly impacts revenue, as well as being responsible for user satisfaction. Additionally, this situation is not suitable for weaker consistency, so we must find ways to satisfy performance needs without sacrificing strong consistency.

Applications have commutativity. Luckily, auctions and many other applications share something besides power laws: commutativity. At the application level, it should be clear that bids on an item can be reordered with one another, provided that the correct maximum bid can still be tracked. When the auction closes, or whenever someone views the current maximum bid, that imposes an ordering which bids cannot move beyond. In the example in Figure 2, it is clear that the maximum bid observed by the `View` action will be the same if the two bids are executed in either order. That is to say, the bids *commute* with one another.

If we take the high-level `Bid` action and implement it on a typical key/value store, we lose that knowledge. The individual `get` and `put` operations used to track the maximum bid conflict with one another. Executing with transactions will still get the right result but only by ensuring mutual exclusion on all involved records for the duration of each transaction, serializing bids per item.

The rest of this paper will demonstrate how ADTs can be used to express these application-level properties and how datastores can use that abstraction to efficiently execute distributed transactions.

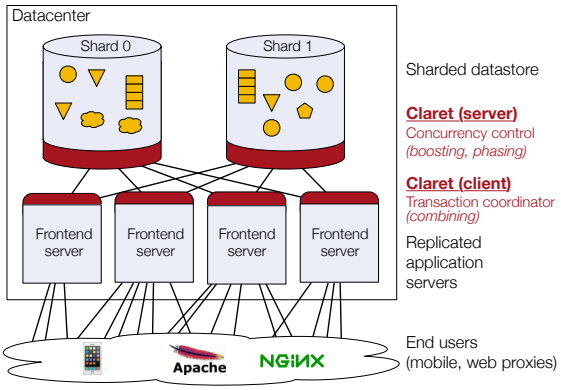

Figure 1. *System model:* End-user requests are handled by replicated stateless application servers which all share a sharded datastore. Claret operates between these two layers, extending the datastore with ADT-aware concurrency control (*Claret server*) and adding functionality to the app servers to perform ADT operations and coordinate transactions (*Claret client*).

3. System model

The concept of ADTs could be applied to many different datastores and systems. For Claret, we focus on one commonly employed system architecture, shown in Figure 1: a sharded datastore shared by many stateless replicated application servers within a single datacenter. For horizontal scalability, datastores are divided into many shards, each containing a subset of the key space (often using consistent hashing), running on different hosts (nodes or cores). Frontend servers are the *clients* in our model, implementing the core application logic and exposing it via APIs to end users that may be mobile clients or web servers. These servers are replicated to mitigate failures, but each instance may handle many concurrent end-user connections, mediating access to the backing datastore where application state resides.

Claret operates between application servers and the datastore. Applications model their state using ADTs and operations on them, as they would in Redis, but differing from Redis, Claret strongly encourages the use of transactions to ease reasoning about consistency. Clients are responsible for coordinating their transactions, retrying if necessary, using multi-threading to handle concurrent end-user requests. A new ADT-aware concurrency control system is added to each shard of the core datastore. ADT awareness is used in both the concurrency control system and the client, which will be explained in more depth in §4.

Programming model. The Claret programming model is not significantly different than traditional key/value stores, especially for users of Redis [33]. Rather than just strings with two available operations, `put` and `get`, records can have any of a number of different types, each of which have operations associated with them. Each record has a type, determined by a tag associated with its key so invalid operations are prevented on the client. The particular client bindings employed are not essential to this work; our code examples will use Python-like syntax similar to Redis's Python bindings though our actual implementation uses C++. An example of an ADT implementation of the Bid transaction is shown in Figure 2.

Consistency model. Weak consistency models require programmers to understand and guard against all potentially problematic interleavings. With Claret, programmers instead focus on choosing ADTs that best represent their desired behavior, naturally exposing opportunities for optimization. Individual operations in Claret are strictly linearizable, committing atomically on the shard that owns the record. Each record, including aggregate types, behaves as a single object living on one shard. Atomicity is determined by the granularity of ADT operations. Custom ADTs can allow arbitrarily complex application logic to be atomic, provided they can be localized to a single object, but in general, composing operations requires transactions.

Transaction model. Claret implements interactive distributed transactions with strict serializable isolation, similar to Spanner [13], with standard `begin`, `commit`, and `abort` functions and automatic retries. Claret supports general transactions: clients are free to perform any operations on any records within the scope of the transaction. It uses strict two-phase locking, acquiring a lock for each record before accessing it during transaction execution.

We support arbitrary ADT operation in transactions by splitting them into two parts, *execute* and *commit*, which both run on the shard holding the record. *Execute* attempts to acquire the lock for the record; when it succeeds, it executes the operation *read-only* to compute a result. Operations without a return value do nothing after acquiring the lock, simply returning control to the calling transaction. Once locks for all operations in a transaction have been acquired, a *commit* is sent to each participating shard to run the *commit* part of all the operations, performing any mutation on the record and releasing the lock. Similar to Spanner [13], clients do not read their own writes; due to the buffering of mutations, operations always observe the state prior to the beginning of the transaction.

We chose a lock-based approach; we briefly explain in §4.1 how this approach could similarly benefit optimistic concurrency control (OCC).

Claret does not require major application modifications to express concurrency. From the clients' view, there are no fundamental differences between using Re-

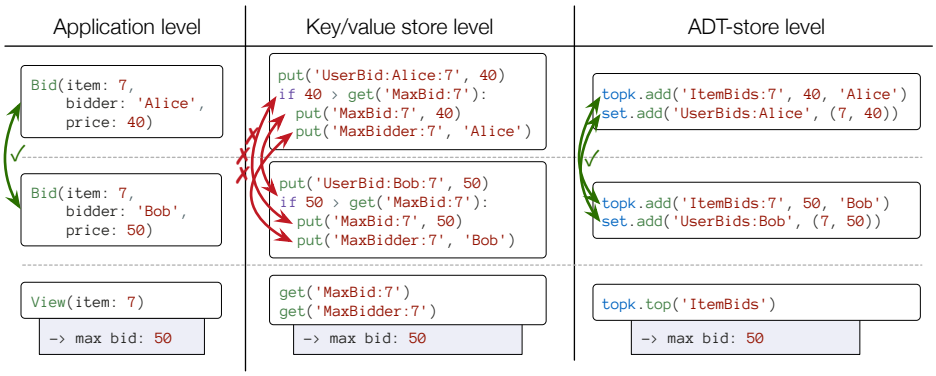

Figure 2. At the application level, it is clear that bid transactions commute, but when translated down to put and get operations, this knowledge is lost. Using an ADT like a `topk` set preserves this commutativity information.

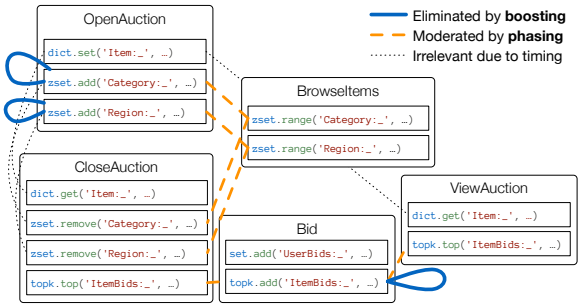

Figure 3. Overview of important Rubis transactions implemented with ADTs. Lines show conflicts between operations, many of which are either eliminated due to commutativity by boosting or mediated by phasing.

dis and Claret (except the addition of distributed transactions and custom types). Under the hood, however, Claret will use its knowledge about ADTs to improve performance in a number of ways that we describe next.

4. Leveraging data types

In Claret, programmers express application-level semantics by choosing the most specific ADT for their needs, either by choosing from the built-in ADTs (Table 1) or implementing their own (see §5). In Figure 2, we saw that `Bid` transactions should commute; we just need know the current high bid. Redis has a `zset` type representing a sorted or ranked set: it associates a score with each item and allows elements to be retrieved by score. A `topk` set is a specialization of `zset` optimized to keep track of only the highest-ranked items. A `topk` meets our needs for bids perfectly. Furthermore, `topk.add` operations commute so `Bid` transactions no longer conflict.

Abstract data types decouple their abstract behavior from their low-level concrete implementation. Abstract operations can have properties such as commutativity, associativity, or monotonicity, which define how they can be reordered or executed concurrently, while the concrete implementation takes care of performing the necessary synchronization.

Knowledge of these properties can be used in many ways to improve performance. First, we show how commutativity can be used in the concurrency control system to avoid false conflicts (*boosting*) and ordering constraints (*phasing*). Then we give an example of how associativity can be applied to reduce the load on the datastore (*combining*).

4.1. Transaction boosting

To ensure strong isolation, all transactional storage systems implement some form of concurrency control. A common approach is strict two-phase-locking (S2PL), where a transaction acquires locks on all records in the execution phase before performing any irreversible changes. However, in distributed systems, holding locks is costly because large round-trip latencies cause them to be held for long periods, depriving the system of much of its potential parallelism. Allowing operations to *share* locks is essential to providing reasonable throughput and latency for transactions. Reader/writer locks are commonly used to allow transactions reading the same record to execute concurrently, but they force transactions writing the record to wait for exclusive access.

Abstract locks [6, 10, 21, 35, 44] generalize reader-writer locks to any operations that can logically run concurrently. When associated with an ADT, they allow operations that commute to hold the lock at the same time. For the `topk` set, `add` operations can all hold the lock at the same time, but reading operations such as `size` must wait.

The same idea can be applied to OCC: operations only cause conflicts if the abstract lock doesn't allow them to execute concurrently with other outstanding operations.

Known as *transaction boosting* [20] in the transactional memory literature, using abstract locks is even more important for distributed transactions which support massively more parallelism but have much longer latencies. Abstract locks directly increase the number of transactions which can execute concurrently. In OCC-based systems, boosting can reduce the abort rate because fewer operations conflict with one another.

Boosting is essential for highly contended records, such as bids on a popular auction as it is about to close. If all the bids are serialized because they are thought to conflict, then fewer can complete and the final price could be lower. Using a `topk` set whose `add` operations naturally commute allows more bids to complete.

4.2. Phasing

Sometimes the order that operations happen to arrive causes problems with abstract locks. In particular, they only help if the operations that commute with each other arrive together; poor interleaving can result in little effect. *Phasing* reorders operations so that commuting operations execute together, similar to batching.

Each ADT defines a *phaser* that is responsible for grouping operations into *phases* that execute together. The interface will be described in more detail in 5.1.2. Each record (or rather, the lock on the record), has its own phaser which keeps track of which mode is currently executing and keeps queues of operations in other modes waiting to acquire the lock. The phaser then cycles through these modes, switching to the next when all the operations in a phase have committed. Abstract locks, which have more distinct modes, benefit more than reader/writer locks, which must still serialize all writes.

By reordering operations, phasing has an effect on fairness. Queueing on locks improves fairness compared to our baseline retry strategy that can lead to starvation. However, because phasing allows operations that commute to be executed before earlier blocked operations, it can lead to fairness issues. If a record has a steady stream of read operations, it may never release the lock to allow mutating operations. To prevent this, we cap phases at a maximum duration whenever there are blocked operations. In our experiments, we only observed this as an issue at extreme skew. The latency of some operations may increase as they are forced to wait for their phase to come. However, reducing conflicts often reduces the latency of transactions overall.

4.3. Combining

Associativity is another useful property of operations. If we consider *commutativity*, used in boosting and phasing above, as allowing operations to be executed in a different order on a record, then *associativity* allows us to merge operations together *before* applying them to the record. This technique, *combining* [19, 38, 47], can drastically reduce contention on shared data structures and improve performance whenever the combined operation is cheaper than all the individual operations. In distributed settings, it is even more useful, effectively distributing synchronization for a single data structure over multiple hosts [23].

For distributed datastores, where the network is typically the bottleneck, combining can reduce server load. If many clients wish to perform operations on one record, each of them must send a message to acquire the lock. Even if they commute and so can hold the lock concurrently, the server handling the requests can get overloaded. In our model, however, "clients" are actually frontend servers handling many different end-user requests. With combining enabled, Claret keeps track of all the locks currently held by transactions on one frontend server. Whenever a client performs a combinable operation, if it finds its lock already in the table, it simply merges with the operation that acquired the lock, without needing to contact the server again.

For correctness, transactions sharing combined operations must all commit together. This also means that they must not conflict on any of their other locks, otherwise they would deadlock, and this applies transitively through all combined operations. Claret handles this by merging the lock sets of the two transactions and aborting a transaction and removing it from the set if it later performs an operation that conflicts with the others.

Tracking outstanding locks and merging lock sets adds overhead but offloads work to clients, which are easier to replicate to handle additional load than datastore shards. In our evaluation (§6), we find that combining is most effective at those critical times of extreme contention when load is highly skewed toward one shard.

5. Expressing abstract behavior

As in any software design, building Claret applications involves choosing the right data structures. There are many valid ways to compose ADTs to model application data, but to achieve the best performance, one should express as much high-level abstract behavior as possible through ADT operations.

Typically, the more specialized an ADT is, the more concurrency it can expose, so finding the closest match is essential. For example, one could use a counter to gener-

Data type	Description
`UIdGenerator`	Create unique identifiers (not necessarily sequential) (`next`)
`Dict`	Map (or "hash") which allows setting or getting multiple fields atomically
`ScoredSet`	Set with unique items ranked by an associated score (`add`, `size`, `range`)
`TopK`	Like `ScoredSet` but keeps only highest-ranked items (`add`, `max`, ...)
`SummaryBag`	Container where only summary stats of added items can be retrieved (`add`, `mean`, `max`)

Table 1. Library of built-in data types.

Method:	And:	Commute 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.

ate unique identifiers, but counters must return numbers in sequence, which is difficult to scale (as implementers of TPC-C [42], which explicitly requires this, know well). A special `UniqueID` type succinctly communicates that non-sequential IDs are acceptable, allowing a commutative implementation.

Claret has a library of pre-defined ADTs (Table 1) used to implement the applications in this paper. Reusing existing ADTs saves implementation time and effort, but may not always expose the maximum amount of concurrency. Custom ADTs can express more complex application-specific properties, but the developer is responsible for specifying the abstract behavior for Claret.

The next sections will show how ADT behavior is specified in Claret to expose abstract properties of ADTs for our optimizations to leverage.

5.1. Commutativity Specification

Commutativity is not a property of an operation in isolation. A *pair* of operations commute if executing them on their target record in either order will produce the same

```
class zset:
  class abstract_lock:
    # return True if `op` can execute on `record`
    # concurrently with other lock holders;
    # adds txn_id to set of lock holders
    def acquire(record, op, txn_id):
      if (op.is_add() and self.mode == ADD) or
         (op.is_read() and self.mode == READ):
        self.holders.add(txn_id)
        return True
      # ...

    # called when a transaction commits or aborts,
    # releasing its locks, remove `txn_id` from
    # lock holders
    def release(txn_id):
      self.holders.remove(txn_id)
      if self.holders.empty():
        self.mode = None
```

Listing 1. Interface for expressing commutativity for a data type. Typical implementations use *modes* to easily determine sets of allowed operations, and a *set* of lock-holders to keep track of outstanding operations.

outcome. Using the definitions from [27], whether or not a pair of method invocations commute is a function of the methods, their arguments, their return values, and the *abstract state* of their target. We call the full set of commutativity rules for an ADT its *commutativity specification*. An example specification for a *Set* is shown in Table 2. However, we need something besides this declarative representation to communicate this specification to Claret's concurrency controller.

5.1.1. Abstract lock interface

In Claret, each data type describes its commutativity by implementing the *abstract lock* interface shown in Listing 1. This imperative interface allows data types to be arbitrarily introspective when determining commutativity. In our implementation, clients must acquire locks for each operation before executing them. When the datastore receives a lock request for an operation on a record, the concurrency controller queries the abstract lock associated with the record using its `acquire` method, which checks the new operation against the other operations currently holding the lock to determine if it can execute concurrently (commutes) with all of them.

Implementations of this interface typically keep a set of the current lock-holders. They determine which operations are currently permitted to share the lock by dividing them into *modes*. For example, reader/writer locks have a *read* mode for all read-only operations and an *exclusive* mode for the rest, while abstract locks have additional modes, such as an *append* mode for sets which allows all `adds`. More fine-grained tracking in `acquire` can ex-

```
class zset:
  class phaser:
    def enqueue(self, op):
      if op.is_read():
        self.readers.push(op)
      elif op.is_add():
        self.adders.push(op)
      # ...

    def signal(self, prev_mode):
      if prev_mode == READ:
        self.adders.signal_all()
      elif prev_mode == ADD:
        self.readers.signal_all()
      # ...
```

Listing 2. Phaser interface (example implementation): `enqueue` is called after an operation fails to acquire a lock, `signal` is called when a phase finishes (all ops in the phase commit and release the lock).

pose more concurrency; for instance, `contains` can execute during *append* if the item already exists.

5.1.2. Phaser interface

Phasing requires knowing how to divide operations into *phases*, similar to the *modes* for locks, but rather than tracking which operations currently hold the lock, the *phaser* associated with a record tracks all the operations waiting to acquire the lock. When an operation fails to acquire the lock, the controller *enqueues* it with the phaser. When a phase completes, the abstract lock *signals* the phaser, requesting operations for a new phase.

The simplest implementations keep queues corresponding to each mode. Listing 2, for example, shows `adders`, which will contain all operations that may insert into the set (just `add`), and `readers`, which includes any read-only operations (`size`, `contains`, `range`, etc). As with abstract locks, more complicated phaser implementations can allow operations to be in multiple modes or use more complex state-dependent logic to determine which operations to signal.

5.2. Combiners

Finally, ADTs wishing to perform combining (§4.3) must implement a *combiner* to tell Claret how to combine operations. Combiners only have one method, `combine`, which attempts to match the provided operation against any other outstanding operations (operations that have acquired a lock but not committed yet).

Remember from §3 that operations are split into *execute* and *commit*. Combining is only concerned with the *execute* part of the operation. Operations that do not return a value (such as `add`) are simple to combine: any

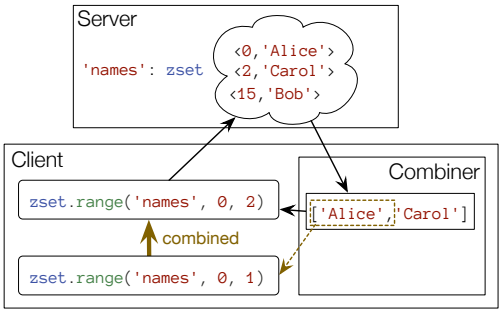

Figure 4. Combining `range` operations: the second operation's result can be computed locally because its range is a subset of the first, so the two can be combined.

commuting operations essentially share the acquired lock and commit together. Operations that return a value in *execute* (any read), can only be combined if they can share the result. For `zset.size`, all concurrent transactions should read the same size, so combined `size` ops can all return the size retrieved by the first one. Figure 4 illustrates a more complex example of combining two `range` operations on a `zset`. They can be combined because the second requests a sub-range of the first, so the combiner can produce the result.

To avoid excessive matching, only operations declared *combinable* are compared. The client-side library keeps track of outstanding combinable operations with a map of combiners indexed by key. Combiners are registered after a lock is acquired and removed when the operation commits. Before sending an acquire request for a combinable operation, the client checks the map for that key. If none is found, or the combine fails, it is sent to the server as usual. If it succeeds, the result is returned immediately, and Claret handles merging the two transactions as described before in §4.3.

5.3. Adding a custom ADT

The ADTs provided by Claret are all implemented using these interfaces to communicate their commutativity and associativity to Claret's concurrency control system. To implement a custom ADT that can take advantage of all of the optimizations, programmers must simply implement an abstract lock, phaser, and combiner. ADT designers could choose to use simple implementations similar to those in Claret that just divide operations statically into modes, or experts could use more state-dependent logic to provide fine-grained concurrency if needed.

6. Evaluation

To understand the potential performance impact of the optimizations enabled by ADTs, we built a prototype in-memory ADT-store in C++, dubbed Claret. Our design follows the architecture previously described in Figure 1, with keys distributed among shards by consistent hashing, though related keys may be co-located using tags as in Redis. Records are kept in their in-memory representations, avoiding serialization. Clients are multi-threaded, to model frontend servers handling many concurrent end-user requests, and use ethernet sockets and protocol buffers to communicate with the data servers.

As discussed before, Claret uses two-phase locking (2PL) to isolate transactions. The baseline uses reader/writer locks. When boosting is enabled, these are replaced with abstract locks. By default, operations retry whenever they fail to acquire a lock; with phasing, replies are sent whenever the lock is finally acquired, so retries are only used to resolve deadlocks or lost packets.

Our prototype does not provide fault-tolerance or durability. These could be implemented by replicating shards or logging to persistent storage. Synchronizing with replicas or the filesystem should increase the time spent holding locks, so lower throughput and higher latency are expected. Claret increases the number of clients which can simultaneously hold locks, so adding fault tolerance would be expected to reinforce our findings.

We wish to understand the impact our ADT optimizations have on throughput and latency under various contention scenarios. First, we use a microbenchmark to directly tune contention by varying operation mix and key distribution. We then move on to explore scenarios modeling real-world contention in two benchmarks – Rubis: an online auction service, and Retwis: a Twitter clone.

The following experiments are run un-virtualized on a RedHat Linux cluster. Each node has dual 6-core 2.66 GHz Intel Xeon X5650 processors with 24 GB of memory, connected by a 40Gb Mellanox ConnectX-2 InfiniBand network, but using normal TCP-over-Infiniband rather than specializing to leverage this hardware's RDMA support. Experiments are run with 4 single-threaded shards running on 4 different nodes, with 4 clients running on other nodes with variable numbers of threads. Average round-trip times between nodes for UDP packets are 150 μs, with negligible packet loss.

6.1. Raw Operation Mix

This microbenchmark performs a random mix of operations, similar to YCSB or YCSB+T [12, 14], that allows us to explicitly control the degree of contention. Each transaction executes a fixed number of operations (4), randomly selecting either a read operation (`set.size`), or a commutative write operation (`set.add`), and keys selected randomly with a Zipfian distribution from a pool of 10,000 keys. By varying the percentage of `adds`, we control the number of potential conflicting operations. Importantly, `adds` commute with one another, but not with `size`, so even with boosting, conflicts remain. The Zipf parameter, α, determines the shape of the distribution; a value of 1 corresponds to Zipf's Law, lower values are shallower and more uniform, higher values more skewed. YCSB sets α near 1; we explore a range of parameters.

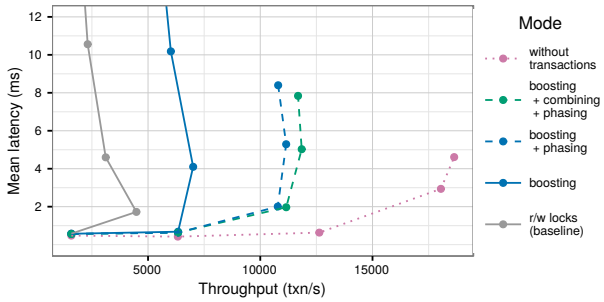

Figure 5. Raw mix workload (50% read, zipf: 0.6), increasing number of clients, plotted as throughput vs. latency. Boosting (abstract locks) eliminates conflicts between `adds`, phasing reorders operations more efficiently. Results in a 2.6x throughput improvement, 63% of the performance without transactions.

We start with a 50% read, 50% write workload and a modest zipfian parameter of 0.6, and vary the number of clients. Figure 5 shows a throughput versus latency plot with lines showing each condition as we vary the number of clients (from 8 to 384). The baseline, using traditional r/w locks, reaches peak throughput with few clients before latencies spike; throughput suffers as additional clients create more contention. Abstract locks (*boosting*) expose more concurrency, increasing peak throughput. Adding phasing (dashed lines) improvements peak throughput because it improves operation fairness while also improving the chances of commuting.

The dotted pink line in Figure 5 shows performance of the same workload with operations executed independently, without transactions (though performance is still measured in terms of the "transactions" of groups of 4 operations). These operations execute immediately on the records in a linearizable [22] fashion without locks. This serves as a reasonable upper bound on the throughput possible with our servers. Claret's transactions achieve 63% of that throughput on this workload.

Varying operation mix. Figure 6 shows throughput as we vary the percentage of commutative write operations (`add`), with keys selected with a modest 0.6 zipfian distribution. Boosting becomes more important as the fraction

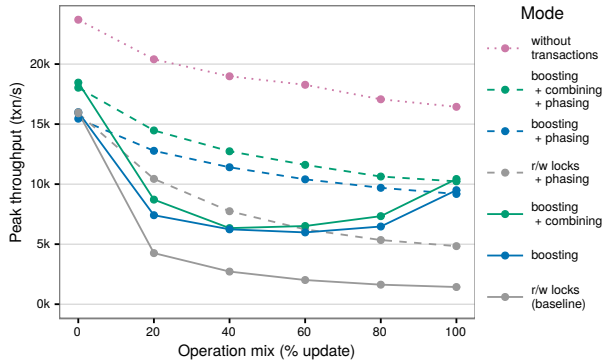

Figure 6. *Peak throughput, varying operation mix.* Boosting is increasingly important with a higher fraction of adds. Phasing is essential for any mixed workload.

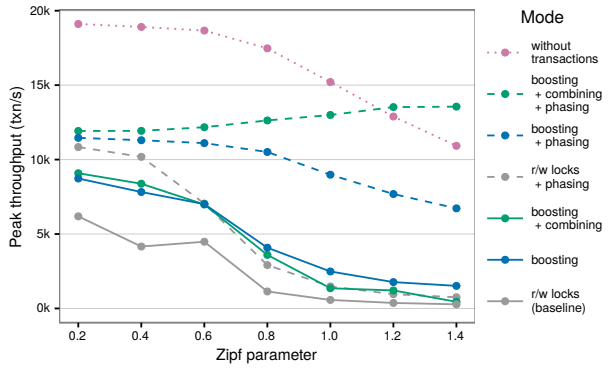

Figure 7. *Peak throughput, varying key distribution.* Higher Zipf parameter results in greater skew and contention; boosting and phasing together expose concurrency. At extreme skew, combining reduces load on hot records, which our non-transactional mode cannot do.

of commutative `adds` increases. Phasing has a significant impact for any mixed workload, as it helps commutative operations run concurrently; tracing the execution, we observed that records regularly alternated between `add` and `read` phases. Combining shows a modest improvement for all workloads, even for read-only and write-only, because it occasionally allows transactions to share locks (and the result of reads) without burdening the server.

Varying key distribution. Figure 7 shows throughput with a 50/50 operation mix, controlling contention by adjusting the zipfian skew parameter used to choose keys. At low zipfian, the distribution is mostly uniform over the 10,000 keys, so most operations are concurrent simply because they fall on different keys, and Claret shows little benefit. As the distribution becomes more skewed, transactions contend on a smaller set of popular records. With less inter-record concurrency, we rely on abstract locks (boosting) to expose concurrency within records.

At high skew, there is a steady drop in performance simply due to serializing operations on the few shards unlucky enough to hold the popular keys. However, skew increases the chance of finding operations to combine with, so combining is able to offload significant load from the hot shards. Our implementation of combining requires operations to be split into the acquire and commit phases, so it cannot be used without transactions. Though distributions during normal execution are typically more moderate, extreme skew models the behavior during exceptional situations like BuzzFeed's viral dress.

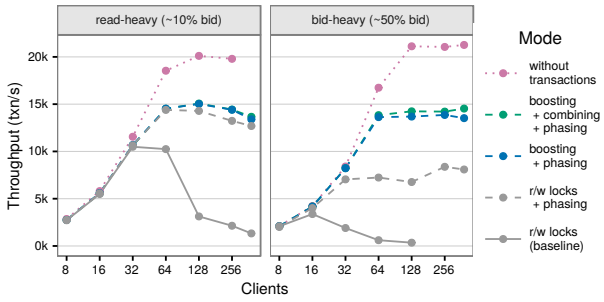

Figure 8. Throughput of Rubis. Contention between bids on popular auctions, especially close to their closing time, causes performance to drop for r/w locks, but bids commute, so boosting is able to maintain high throughput even under heavy bidding.

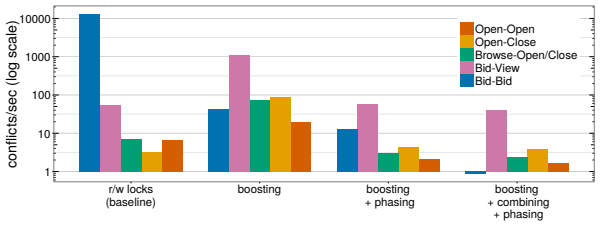

Figure 9. Breakdown of conflicts between Rubis transactions (minor contributors omitted) with 256 clients on bid-heavy workload (averaged). As predicted by Figure 3, boosting drastically reduces Bid-Bid conflicts, and phasing drastically reduces the remaining conflicts.

6.2. RUBiS

The RUBiS benchmark [4] imitates an online auction service like the one described in §2. The 8 transaction types and their frequencies are shown in Table 3; `ViewAuction` and `Bid` dominate the workload. The benchmark specifies a workload consisting of a mix of these transactions and the average bids per auction. However, the distribution of bids (by item and time) was unspecified.

Our implementation models the bid distributions observed by subsequent studies [1, 2], with bids per item following a power law and the frequency of bids increasing exponentially at the end of an auction. Otherwise, we follow the parameters specified in [4]: 30,000 items, divided into 62 regions and 40 categories, with an average of 10 bids per item, though in our case this is distributed according to a zipfian with $\alpha = 1$.

Figure 8 shows results for two different workloads: read-heavy and bid-heavy. In the read-heavy workload, bids do not often come in at a high enough rate to require commutativity, so phasing alone suffices. However, during heavy bidding times, commutativity is essential: Claret maintains nearly the same throughput in this situation as the read-heavy workload, roughly 2x better than r/w locks and 68% of non-transactional performance. Considering the importance of getting bids correct, this seems an acceptable tradeoff.

Prior work [1] observed contention spikes when popular auctions closed, leading to momentary performance drops which could be noticeable to even to users elsewhere on the site. Analyzing a trace of throughput over time, we observed that boosting significantly reduced variability in throughput by 2x. The minimum throughput (over 5-second windows) was also increased from 9k txn/s for the baseline with phasing to 12k.

Figure 3 showed which conflicts should be affected by Claret; in Figure 9 we validate those predictions by plotting the actual number of conflicts for the most significant edges in the conflict graph. Using a log scale, it is apparent that boosting all but eliminates Bid-Bid conflicts, but Bid-View conflicts have gone up; now that there are more bids, the chances of conflicting with `ViewAuction` have increased. The introduction of phasing and combining eliminate much of the remaining conflicts.

Overall, we can see that boosting and phasing are crucial to achieving reasonable transaction performance in Rubis even during heavy bidding. If an auction service is unable to keep up with the rate of bidding, it will result in a loss of revenue and a lack of trust from users, so a system like Claret could prove invaluable to them.

6.3. Retwis

Retwis is a simplified Twitter clone designed originally for Redis [33]. Data structures such as lists and sets are used track each user's followers and posts and keep a materialized up-to-date timeline for each user (as a `zset`). It is worth noting that in our implementation, `Post` and `Repost` are each a single transaction, including appending to all followers' timelines, but when viewing timelines, we load each post in a separate transaction.

Retwis doesn't specify a workload, so we simulate a realistic workload using a synthetic graph with power-law

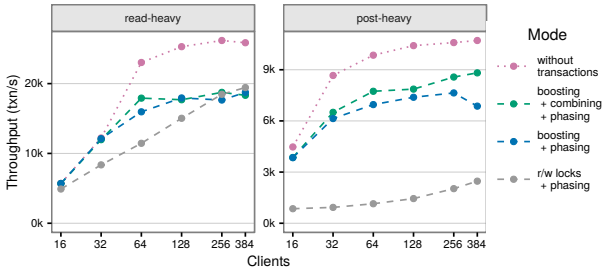

Figure 10. Throughput of Retwis. Boosting is essential during heavy posting because network effects lead to extreme contention on some records. Non-phasing results elided due to too many failed transactions.

RUBiS	`View/ Browse`	`Bid`	`Open/ Close`	`Comment`	`NewUser/ ViewUser`
bid-heavy	52%	45%	1%	1%	1%
read-heavy	82%	10%	4%	2%	2%

Retwis	`Timeline`	`Repost`	`Post`	`Follow`	`NewUser`
post-heavy	90%	6%	2%	1%	< 1%
read-heavy	98%	1%	0.5%	0.2%	< 0.1%

Table 3. Transaction mix for benchmark workloads.

degree distribution and a simple user model. We use the Kronecker graph generator from the Graph 500 benchmark [18], which is designed to produce the same power-law degree distributions found in natural graphs. These experiments generate a graph with approximately 65,000 users and an average of 16 followers per user.

Our simple model of user behavior determines when and which posts to repost. After each timeline action, we rank the posts by how many reposts they already have and repost the most popular ones with probability determined by a geometric distribution. The resulting distribution of reposts follows a power law, approximating the viral propagation effects observed in real social networks.

Figure 10 shows throughput on two workloads, listed in Table 3. The read-heavy workload models steady-state Twitter traffic, while the post-heavy workload models periods of above-average posting, such as during live events. We only show the results with phasing because the non-phasing baseline had too many failed transactions. On the read-heavy workload, r/w locks are able to keep up reasonably well; after all, reading timelines is easy as long as they do not change frequently. However, the post-heavy workload shows that when contention increases, the performance of r/w locks falls off much more drastically than with boosting. Combining even appears to pay off when there are enough clients to find matches.

Unlike auctions, many situations in Twitter are tolerant of minor inconsistencies, making it acceptable to implement without transactions in order to aid scalability. This tradeoff is clear when performance is as flat as the baseline performance is in these plots. However, with Claret's optimizations, it is able to achieve up to 82% of the non-transactional performance. In situations where inconsistent timelines are more likely to be noticed, such as conversations, it may be worth paying this overhead.

Evaluation summary. We find that leveraging commutativity via boosting and phasing is clearly beneficial under all of our simulated scenarios, showing greater benefit under more extreme contention resulting from high skew or heavy writing. Combining appears mostly ineffectual in our benchmarks, but does not hinder performance. In the most dire circumstances of extreme contention, having combining as an optional release valve for offloading work is useful. Moreover, these improvements come with a programming model largely identical to Redis's, which is sufficient for many applications that only require simple ADTs already built into Redis. More complex applications may require custom ADTs.

7. Related Work

Improving Transaction Concurrency. Several recent systems have explored ways of exposing more concurrency between transactions. An old technique, transaction chopping [37], statically analyzes transactions and splits them to reduce the time locks are held for. Recently, Lynx [48] introduced a form of runtime pipelining to allow conflicting transaction pieces to interleave.

Salt and Callas [45, 46] introduced ways of separating transactions that often conflict from the rest so they can be handled differently. In Salt, problematic transactions were rewritten to operate with weaker consistency, which Callas improved upon to maintain the same ACID properties by using runtime pipelining within a group. Boosting and phasing could both be applied within one of these groups along with runtime pipelining to expose concurrency among frequently conflicting transactions. It would be interesting to explore whether Claret's ADT knowledge could be used to inform Callas's grouping decisions, which are crucial to performance.

ADTs and Commutativity. The importance of commutativity is well known in databases and distributed systems. ADTs were first used in databases in the 1980s to support indices for custom data types [40, 41], and for concurrency control [17, 21, 35, 44]. That work introduced abstract locks to allow databases to leverage commutativity and also allowed user-defined types to express their abstract behavior to the database. Another classic system [39] used type-based locks in an early form of distributed transactions.

To improve scalability and flexibility, NoSQL stores gave up the knowledge afforded by predefined schemas, but recent work has shown how these systems can still leverage commutativity. Lynx [48] allows users to annotate parts of transactions as commutative. HyFlow [26] combines multi-versioning and commutativity to reorder commutative transactions before others, but requires annotating entire transactions as commutative. ADTs naturally express commutativity as part of application design and abstract locks expose more fine-grained concurrency.

Doppel [31], a multicore in-memory database, allows commutative operations on highly contended records to be performed in parallel *phases* with a technique called *phase reconciliation*; Claret's *phasers* are more general in that they expose pairwise operation commutativity on arbitrary ADTs, but do not parallelize across cores.

In eventually consistent datastores, commutativity can improve convergence. RedBlue consistency [28] exploits the convergence of commutative "blue" operations to avoid coordination. Conflict-free (or convergent) replicated data types (CRDTs/CvRDTs) [36] force operations to commute by defining merge functions that resolve conflicts automatically and have been implemented for production in Riak [8]. Claret's strictly linearizable model exposes concurrency without relaxing consistency because CRDT behavior is often still counterintuitive.

Bloom [3, 11] helps programmers design applications in ways that do not require distributed coordination, but requires substantial changes to programs. Claret's model is natural for anyone familiar with Redis.

Complex data structures in NoSQL stores. There is a recent trend toward supporting complex data structures in NoSQL datastores. Redis [33], one of the most popular key/value stores, has a broad command set supporting operations on several complex data types, but cannot be easily extended with new data types, nor does it support transactions. Hyperdex [15, 25] supports atomic operations on common data structures, and transactions [16]. MongoDB [30] supports atomically updating documents.

These systems support data types to make data modeling easier, not for performance. Claret leverages the same data types to expose concurrency and is designed to be extended with custom types.

8. Conclusion

Claret mitigates contention in real-world workloads by allowing programmers to expose application-level semantics with ADTs, which the datastore leverages in the transaction protocol and client library. These optimizations lead to significant speedups for transactions in high contention scenarios, without hurting performance on lighter workloads, making them competitive with non-transactional performance.

References

[1] Vasudeva Akula and Daniel A Menascé. An analysis of bidding activity in online auctions. In *E-Commerce and Web Technologies*, pages 206–217. Springer, 2004.

[2] Vasudeva Akula and Daniel A. Menascé. Two-level workload characterization of online auctions. *Electronic Commerce Research and Applications*, 6 (2): 192–208, jun 2007. doi:10.1016/j.elerap.2006.07.003.

[3] Peter Alvaro, Neil Conway, Joe Hellerstein, and William R Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260. Citeseer, 2011.

[4] 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.

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

[6] 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.

[7] Lisa Baertlein. Ellen's Oscar 'selfie' crashes Twitter, breaks record. http://www.reuters.com/article/2014/03/03/us-oscars-selfie-idUSBREA220C320140303, March 2014.

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

[9] Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. *Proceedings of the 40th International Conference on Very Large Data Base (VLDB 2014)*, 7 (13), 2014.

[10] 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.

[11] 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.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing - SoCC 10*. Association for Computing Machinery (ACM), 2010. doi:10.1145/1807128.1807152.

[13] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, pages 251–264, 2012. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387905.

[14] 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)*, mar 2014. doi:10.1109/icdew.2014.6818330.

[15] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex. In *Proceedings of the ACM SIGCOMM Conference*. Association for Computing Machinery (ACM), August 2012. doi:10.1145/2342356.2342360.

[16] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Warp: Multi-key transactions for key-value stores. Technical report, Cornell University, November 2013.

[17] Alan Fekete, Nancy Lynch, Michael Merritt, and William Weihl. Commutativity-based locking for nested transactions. *Journal of Computer and System Sciences*, 41 (1): 65–156, August 1990. doi:10.1016/0022-0000(90)90034-i.

[18] Graph500. Graph 500. http://www.graph500.org/, July 2012.

[19] 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*, pages 355–364. ACM, 2010.

[20] 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.

[21] 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.

[22] 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, jul 1990. doi:10.1145/78969.78972.

[23] 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, 10 2013. URL http://sampa.cs.washington.edu/papers/holt-pgas13.pdf.

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

[25] Hyperdex. Hyperdex. http://hyperdex.org/, 2015.

[26] Junwhan Kim, Roberto Palmieri, and Binoy Ravindran. Enhancing Concurrency in Distributed Transactional Memory through Commutativity. In *EuroPar 2013*, pages 150–161. 2013. doi:10.1007/978-3-642-40047-6_17.

[27] Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui, and Keshav Pingali. Exploiting the Commutativity Lattice. In *Conference on Programming Language Design and Implementation*, PLDI, pages 542–555, 2011. ISBN 978-1-4503-0663-8. doi:10.1145/1993498.1993562.

[28] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, OSDI, pages 265–278, 2012. ISBN 978-1-931971-96-6.

[29] Daniel A. Menascé and Vasudeva Akula. Improving the performance of online auctions through server-side activity-based caching. *World Wide Web*, 10 (2): 181–204, feb 2007. doi:10.1007/s11280-006-0011-8.

[30] MongoDB, Inc. Mongodb. https://www.mongodb.org/, 2015.

[31] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase Reconciliation for Contended In-Memory Transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, SOSP, pages 511–524, Broomfield, CO, October 2014. ISBN 978-1-931971-16-4.

[32] Dao Nguyen. What it's like to work on buzzfeed's tech team during record traffic. http://www.buzzfeed.com/daozers/what-its-like-to-work-on-buzzfeeds-tech-team-during-record-t, feb 2015.

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

[34] 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.

[35] Peter Martin Schwarz. *Transactions on Typed Objects*. PhD thesis, Pittsburgh, PA, USA, 1984. AAI8506303.

[36] 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 '11, pages 386–400, 2011. ISBN 978-3-642-24549-7.

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

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

[39] Alfred Z. Spector, Dean Daniels, Daniel Duchamp, Jeffrey L. Eppinger, and Randy Pausch. Distributed transactions for reliable systems. *ACM SIGOPS Operating Systems Review*, 19 (5): 127–146, dec 1985. doi:10.1145/323627.323641.

[40] 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.

[41] 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.

[42] Transaction Processing Performance Council. TPC-C. http://www.tpc.org/tpcc/, 2015.

[43] Voldemort. Voldemort. http://www.project-voldemort.com/voldemort/, 2015.

[44] 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.

[45] 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.

[46] 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 '15, pages 276–291, 2015. ISBN 978-1-4503-2388-8. doi:10.1145/2517349.2522729.

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

[48] 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 '13, pages 276–291, 2013. ISBN 978-1-4503-2388-8. doi:10.1145/2517349.2522729.