Claret: Avoiding Contention in Distributed Transactions with Abstract Data Types

Paper #120

Abstract

Interactive distributed applications like Twitter or eBay are difficult to scale because of the high degree of writes or update operations. The highly skewed access patterns exhibited by real-world systems lead to high contention in datastores, causing periods of diminished service or even catastrophic failure. There is often sufficient concurrency in these applications to scale them without resorting to weaker consistency models, but traditional concurrency control mechanisms 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 communication, 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, bringing overwhelming forces to bear on fledgeling applications that put DDOS attackers to shame. 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, which brought unprecedented traffic spikes to BuzzFeed [31], 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 [6].

These high traffic events arise due to 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 can result in writes as providers track user behavior in order to personalize their experience, target ads, or collect statistics [8].

To avoid catastrophic failures and mitigate poor tail behavior, significant engineering effort must go into handling these challenging high-contention scenarios. The reason writes are such a problem is that 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: some actions are inherently acceptable to reorder. 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 cases with high contention. Instead, if systems could directly use these application-level constraints to expose concurrency and avoid over-synchronizing, they could eliminate many false conflicts and potentially avoid falling over during writing spikes, without sacrificing correctness. Databases and distributed systems have long used properties such as commutativity to reduce coordination and synchronization. The challenge is always in communicating these application-level properties to the system.

In this work, we propose a new way to express high-level application semantics for transactions through *abstract data types* (ADTs) and consequently avoid unnecessary synchronization in distributed transactional datastores. 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. Datastores 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 to match 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, 42] support simple collections such as *lists*, *sets*, and *maps*, and even custom objects (e.g. protocol buffers). Redis [32], 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.

In *Claret*, we expose the logical properties of these data types to the system, communicating properties of the application to the datastore 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 [33]. While Claret's optimizations help most in high-contention cases, its performance on read-heavy workloads with little contention is not affected. Additionally, on high-contention workloads, Claret's strongly consistent transactions can 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 the remainder of 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

Systems interacting with the real world often exhibit some common patterns which lead to contention: power-law distributions, network effects, and realtime events. However, much of this contention can be mitigated by understanding what semantics are desired at the application level.

2.1. Power laws everywhere

Natural phenomena have a tendency to follow power law distributions, from physical systems to social groups. Zipf's Law is the observation that the frequency of words in a natural language follows a power law (specifically, frequency is inversely proportional to the rank). The connectivity of social networks is another well-known example: a small number of nodes (people) account for a large fraction of the connections, while most people have relatively few connections. Power laws can play off of each other, leading to other interesting properties, such as low diameter or *small-world* networks (colloquially, "six degrees of separation"). Network effects serve to amplify small signals into massive amounts of activity, such as occurs when a meme goes viral. Finally, systems with a real-time component end up with spikes of activity as events occur in real life. For example, goals during World Cup games cause spikes in Twitter traffic, famously causing the "fail whale" to appear [23].

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, 28] 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 will be directly related to revenue, as well as being responsible for user satisfaction. Additionally, this situation is not suitable for

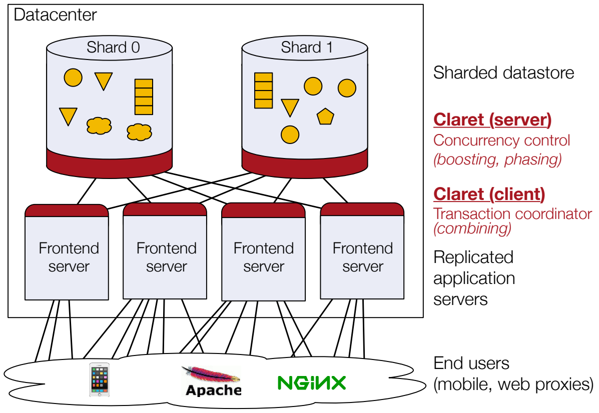


Figure 1. *System model:* End-user requests are handled by replicated stateless application servers which all share a sharded datastore. Claret operates between these two layers, extending the datastore with ADT-aware concurrency control (*Claret server*) and adding functionality to the app servers to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

weaker consistency. Therefore, we must find ways to satisfy performance needs without sacrificing strong consistency.

2.2. Application-level 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.

The problem is that 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.

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 typically 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 which might 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.

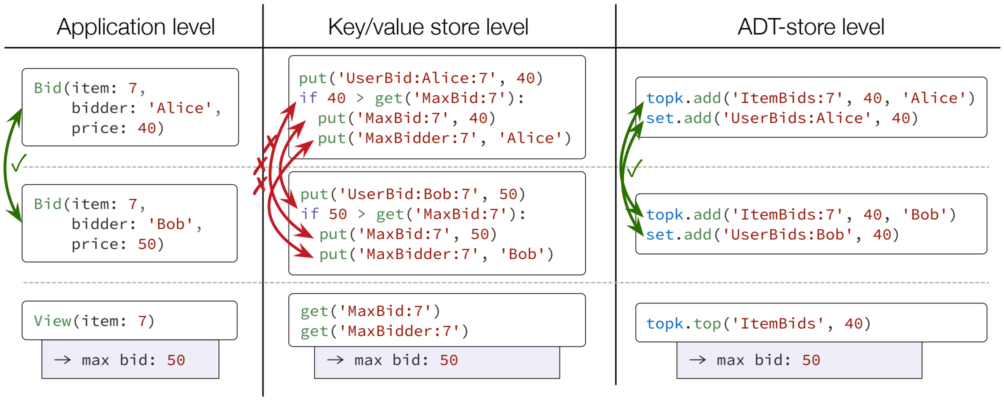
3.1. Programming model

The Claret programming model is not significantly different than traditional key/value stores, especially for users of Redis [32]. 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.

3.2. Consistency model

Rather than relying on weaker consistency models, Claret aims to use application semantics to make strong consistency practical. Instead of guarding actions in case of inconsistency, application programmers instead focus on exposing concurrency by choosing ADTs that best represent the desired behavior and expose concurrency.

Individual operations in Claret are strictly linearizable, committing atomically on the shard that owns the record. Each record, including aggregates, behaves as a single object living on one shard. Atomicity is determined by the granularity of individual ADT operations. Custom ADTs allow arbitrarily complex application logic to be performed atomically, provided they conceptually operate on a single "record". Composing actions between multiple objects requires transactions.



```
Bid(item: 7,
    bidder: 'Alice',
    price: 40)
```

```
Bid(item: 7,
    bidder: 'Bob',
    price: 50)
```

```
View(item: 7)
→ max bid: 50
```

```
put('UserBid:Alice:7', 40)
if 40 > get('MaxBid:7'):
  put('MaxBid:7', 40)
  put('MaxBidder:7', 'Alice')
```

```
put('UserBid:Bob:7', 50)
if 50 > get('MaxBid:7'):
  put('MaxBid:7', 50)
  put('MaxBidder:7', 'Bob')
```

```
get('MaxBid:7')
get('MaxBidder:7')
→ max bid: 50
```

```
topk.add('ItemBids:7', 40, 'Alice')
set.add('UserBids:Alice', 40)
```

```
topk.add('ItemBids:7', 40, 'Bob')
set.add('UserBids:Bob', 40)
```

```
topk.top('ItemBids', 40)
→ max bid: 50
```

Figure 2. At the application level, many transactions ought to commute with one another, such as these Bid transactions, but when translated down to put and get operations, this knowledge is lost.

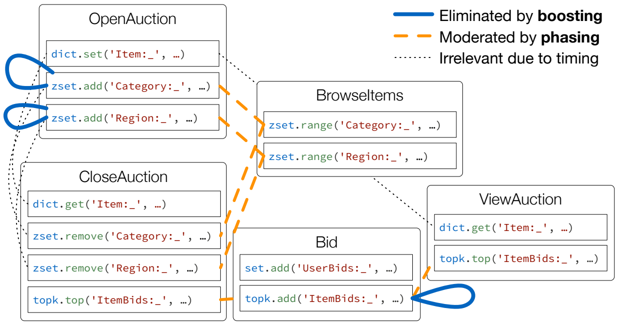


Figure 3. High-level overview of important Rubis transactions, implemented with ADTs. Lines show conflicts between operations, many of which are either eliminated due to commutativity by boosting or mediated by phasing.

3.3. Transactions

Claret implements interactive distributed transactions with strict serializable isolation, similar to Spanner [12], 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 and acquires locks for each record accessed during transaction execution.

To support arbitrary ADT operations in transactions, each operation is split into two parts, *prepare* and *commit*, which are executed on the shard holding the record. *Prepare* always starts by attempting to acquire the lock for the record. When the lock has been acquired, *prepare* may read any data it wishes from the record to compute a *result* which will be returned to the calling transaction. Operations returning *void* typically 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 transaction commit is sent to each participating shard, which executes the *commit* part of each of the *prepared* operations. The commit stage performs any necessary mutation on the record and releases the lock.

Similar to Spanner [12], 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 use a lock-based approach rather than optimistic concurrency control (OCC), but OCC should also benefit in similar ways from Claret's optimizations.

Claret does not require major application modifications to express concurrency. From the clients' view, there are no fundamental differences between using Redis 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 ways which we describe next.

4. Leveraging data types

In Claret, programmers express application-level semantics through ADTs. We know that the `Bid` transactions in Figure 2 should commute somehow, and we need to be able to determine the current high bid. A `topk` set meets our needs: it associates a score with each item, but is optimized to track those with the highest scores. Because `topk.add` operations commute, `Bid` transactions no longer conflict. Applications express their desired 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).

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 by the datastore in many ways to improve performance. First, we will show how commutativity can be used in the concurrency control system to avoid false conflicts (*boosting*) and ordering constraints (*phasing*). Then we will 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. Typically these are reader/writer locks, so multiple read operations can execute concurrently in different transactions. This means that transactions need not block if they are reading a record already being read by other transactions, but writes cause other transactions to block.

Abstract locks [@] generalize the notion of reader/writer locks to any operations which can logically run concurrently on the same record. With an abstract lock for an ADT, the concurrency control system can allow any operations that commute to hold the lock at the same time. For `zset`, `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 optimistic concurrency control: operations only cause conflicts if the abstract lock doesn't allow them to execute concurrently with other outstanding operations.

Using abstract locks to improve concurrency in transactions is known as *transaction boosting* [19] in the transactional memory community. However, this technique can be even more valuable to distributed transactions because their performance depends on how long locks are held. Waiting for one lock causes a cascade of waiting as others wait on the locks the blocked transaction holds. Every operation that can share an abstract lock reduces the time the rest of its locks are held for, which can have a big impact on performance. In OCC-based systems, boosting reduces the abort rate because fewer operations conflict with one another.

Boosting is especially important for highly contended records. In the case of auctions, when a particularly hot auction is near closing time, it can expect to receive a huge number of bids. If all of the bids conflict with each other and serialize, it may cause some of them to not complete in time. However, if the bids are represented using a `zset`, then they naturally commute, the transactions can execute concurrently, and everyone gets to place their bids.

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. If they are interleaved in time with operations they do not commute with, then all of that cleverness is for naught. Borrowing the term from recent work on *phase reconciliation* [30], *phasing* is the idea of reordering operations so that commuting operations execute together.

To implement phasing, each ADT defines a *phaser* which is responsible for grouping operations into *phases* which can execute together. The interface will be described in more detail in §5.2. 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.

Phasing also helps ensure fairness and prevent starvation. When operations try to acquire a lock, usually they either succeed and get the lock or are denied and must retry. When they retry, they may find that the lock is again held by someone else, possibly someone who started later, but got lucky and arrived at the right time. Alternatively, a record may receive a steady stream of read-only operations and some mutating operations. If there is always at least one outstanding read, then the read lock will never be released and the writes may starve. Instead, phases are capped at a maximum duration as long as operations in other modes are queued up.

Reader/writer locks can also benefit from phasing; in that case, there are just two modes, reading and writing, though writing, of course, only allows one operation at a time.

Due to phasing, 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

Another useful property on operations is *associativity*. If we think of *commutativity*, used in boosting above, as allowing operations to be executed in a different order on a given record, then *associativity* allows us to merge some of those operations together *before* applying them to the record. This technique, known as *combining* [18, 37, 46] can drastically reduce contention on shared data structures and improve performance in situations where applying the combined operation is cheaper than applying the operations one-by-one. In distributed settings, combining can be even more useful as it effectively distributes synchronization for a single data structure over multiple hosts [22].

For distributed datastores where the network is typically the bottleneck, combining can help reduce the load on the server. If many clients all 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 shard 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 an operation that can be combined with one of the outstanding ones, it combines them and the combined operation no longer needs to talk to the server to acquire the lock itself.

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 locks sets of the two transactions whenever operations are combined and aborting a transaction and removing it from the set if it later performs an operation that conflicts with the others.

With all the overhead of tracking outstanding locks and merging transaction sets, it seems like this might be more work than it is worth, and it likely is in some cases. However, these frontend servers can often afford to spend this effort to try combining because they are easy to replicate to handle additional load, whereas the datastore shards they are saving work for cannot.

5. Expressing abstract behavior

Just as in any software design, building Claret applications involves choosing the right data structures. There are many valid ways of composing ADTs within transactions for a correct implementation, but to achieve the best performance, the programmer must express as much of the high-level abstract behavior as possible through ADT operations, such as atomicity and commutativity.

Typically, the more specialized an ADT is, the more concurrency it can expose, so finding the closest match for each use case is essential. For example, an application needing to generate unique identifiers should not use a counter, which must return the next number in the sequence, because this is very difficult to scale (as implementers of TPC-C [41], which explicitly requires this, know well). Instead, a `UniqueID` type succinctly expresses that non-sequential IDs are okay, and can implement it in a way that is fully commutative.

Claret has a library of pre-defined ADTs, shown in Table 1 which were 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 use in the optimizations previously described.

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 outcome. Using the definitions from [26], 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.

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	Commute 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.

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 pessimistic (locking) implementation, the client must acquire locks for its operations 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.

A traditional reader/writer lock can be implemented in this way by tracking all the transactions currently reading a record when it's in *reading* mode, only allowing a write (*exclusive* mode) when all readers have released their locks.

```
class ZSet:
  class AbstractLock:
    # 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.

More permissive abstract locks can be implemented simply by adding additional modes, such as *appending* for sets or lists, which allow all add operations. More fine-grained commutativity tracking can be done, at the cost of increased computation in the lock acquire step. For instance, a set add could acquire the lock during a read-only mode if the set already contains the item it adds.

5.1.2. Transaction boosting

Claret uses abstract locks to perform transaction boosting. With these locks, programmers can be assured that transactions will only conflict with each other on operations that do not commute. In our auction application, bids are performed by adding to a topk set (as in Figure 2). For popular items and near the end of auctions, the ItemBids zset will be the most highly contended, but because adds commute, those bid transactions will not conflict with each other.

5.2. Phaser

The phasing optimization described in §4.2 requires knowing how to divide operations for each ADT into phases. Claret accomplishes this by pairing a *phaser* with the abstract lock on each record. With phasing enabled, whenever an operation fails to acquire a lock, the concurrency controller *enqueues* the operation into the *phaser* for that record. When all operations in a phase have committed, the abstract lock *signals* the phaser, requesting operations to start a new phase. The simplest phaser implementations simply keep queues corresponding to *modes* (sets of operation types that always commute with one another). Listing 2, for example, shows adders, which will contain all operations that may insert into the list (just add), and readers, which includes any read-only operations (size, contains, range, etc). More complicated phaser implementations may allow operations to have multiple modes or use complicated state-dependent ways of determining which operations to signal.

```
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).

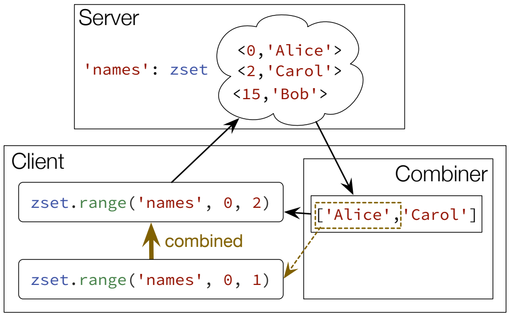


Figure 4. Combining range operations: the second operation's result can be computed locally from the first because its range is a subset of the first, so the two can be combined.

5.3. Combiner

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 §3.3 that operations are split into *prepare* and *commit*. Combining is only concerned with the *prepare* part of the operation. Operations that do not return a value (such as add) are simple to combine: any commuting mutating operations essentially just share the acquired lock and commit together. Operations that return a value in the *prepare* step (any read), can only be combined if they can share the result. For the `zset.size` operation, this is simple enough: 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`. The two operations can be combined because the second operation 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, as discussed earlier, keeps track of outstanding combinable operations with a map of combiners indexed by key. Typically, combiners are registered after a lock is acquired, and is removed when the operation finally commits. Before sending an acquire request for a combinable operation, the client checks the combiner map for that key. If a combiner is not found, or the combine fails, the operation is sent to the server as usual. If it succeeds, the result of the acquire stage is returned immediately, and Claret handles merging the two transactions as described before.

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 as in Redis to avoid serializing and deserializing except for operations from the clients. Clients are multi-threaded, modeling 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 a standard two-phase locking (2PL) scheme to isolate transactions. The baseline uses reader/writer locks which allow any number of read-only operations in parallel, but requires an exclusive lock for operations that modify the record. When boosting is enabled, these are replaced with abstract locks, which allow any operations that commute with one another to hold the lock at the same time. 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.

This prototype does not currently provide fault-tolerance or durability. These could be implemented by replicating shards or logging operations to persistent storage. Synchronizing with replicas or the filesystem is expected to increase the time spent holding locks. In 2PL, the longer locks are held, lower throughput and higher latency are expected. Claret increases the number of clients which can simultaneously hold locks, so providing fault tolerance should be expected to exaggerate the results we obtained.

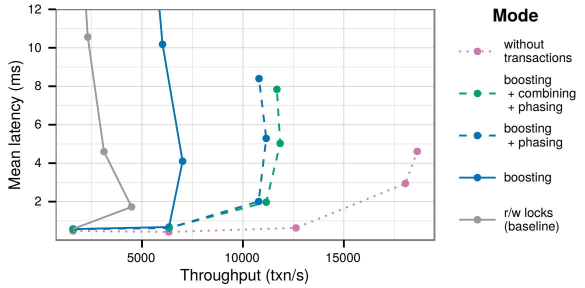


Figure 5. Performance of raw mix workload (50% read, zipf: 0.6) with increasing number of clients, plotted as throughput versus latency. Boosting (abstract locks) eliminate conflicts between `adds`, and phasing helps operations acquire locks in an efficient order, resulting in a 2.6x improvement in throughput, 63% of the throughput without transactions.

In our evaluation, we wish to understand the impact our ADT optimizations have on throughput and latency of transactions under situations with varied contention. First, we explore contention in a microbenchmark where we can easily tune the contention by varying operation mix and key distribution. We then move on to explore scenarios modeling real-world contention in two modified benchmarks; Rubis: an online auction service, and Retwis: a Twitter clone.

All the following experiments are run on our own cluster in RedHat Linux, un-virtualized. Each node has dual 6-core 2.66 GHz Intel Xeon X5650 processors with 24 GB of memory, connected with a 40Gb Mellanox ConnectX-2 InfiniBand network. Unless otherwise noted, 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

To best explore the impact of Claret's optimizations on high-contention workloads, we use a microbenchmark with transactions performing random operations, similar to YCSB or YCSB+T [11, 13], where contention can be explicitly controlled. 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 zipfian 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

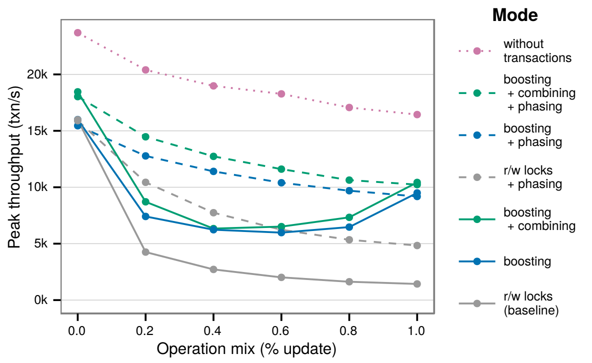


Figure 6. *Peak throughput, varying operation mix*. With even just 20% adds, throughput plummets without phasing, which helps adds get a chance to run between all the reads. With higher add ratios, boosting becomes important.

sets α near 1, but other prior work [9] found that lower values of α, ranging from 0.64 to 0.83, better characterized the workloads they studied. We explore a range of α's in Figure 7, but default to 0.6 elsewhere to be conservative, since Claret's optimizations perform better with more skew.

First, 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). We see immediately that the baseline, using traditional r/w locks, reaches peak throughput with few clients and then latencies go up and throughput suffers as more clients create more contention. Using abstract locks (*boosting*) to expose more concurrency increases peak throughput somewhat. Using phasing in addition (dashed lines) results in significant throughput improvement and lower latency. This is because phasing ensures that all transactions get their turn while also improving the chances that operations will commute.

The dotted pink line in Figure 5 shows performance of the same workload where operations are executed independently, without transactions (though performance is still measured in terms of the "transactions" of groups of 4 operations). These operations execute without conflicting, since they do not acquire locks, instead executing immediately on the records in a linearizable [21] fashion. This serves as a reasonable upper bound on the throughput possible with our servers. Claret achieves 63% of that upper bound on throughput on this workload, though of course with strong consistency.

6.1.1. Varying operation mix

Figure 6 shows throughput as we vary the percentage of commutable write operations (add). From this experiment, we can conclude that phasing is useful even for relatively light amounts of writes because it helps ensure that add operations get their turn amidst all the read operations, and phasing is made even more useful with abstract locks (boosting), where sets regularly alternate between add and read phases. Moreover, we observe that combining can benefit even read-only or write-only workloads because it allows transactions to share results, reducing load on the servers.

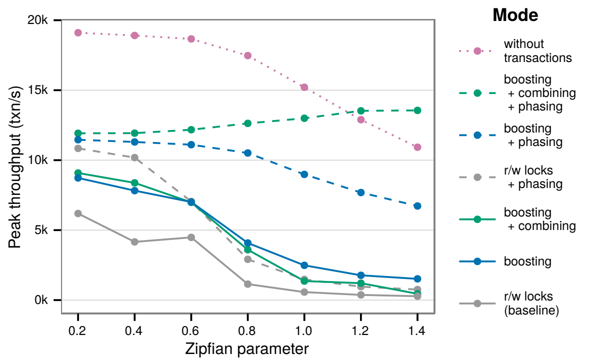


Figure 7. *Peak throughput, varying key distribution*. Higher zipfian results in more skew and greater contention; boosting is essential to expose concurrency on hot records. At extreme skew, combining becomes much more likely to occur, while our non-transactional mode is unable to benefit from it.

6.1.2. Varying key distribution

Our final way to control contention is to vary the zipfian parameter used to select keys; Figure 7 shows throughput results with 50% adds. 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, though because of the significantly write-heavy workload, Claret's optimizations do help modestly. As the zipfian distribution becomes more skewed, transactions contend more often on fewer, more popular, records. With less inter-record concurrency, we rely more on abstract locks (boosting) to expose concurrency within records.

For high skew, even without transactions we see a steady drop in performance simply due to serializing operations on the few shards unlucky enough to hold the especially popular keys. However, combining is able to save the day here, because at high skew, there is a very good chance of finding operations to combine with, offloading significant load from the otherwise-overloaded popular shards. Our implementation of combining requires operations to be split into the acquire and commit phases, so it cannot be used without transactions. Though workloads overall do not typically follow such extreme distributions, they may accurately model behavior during exceptional situations, such as BuzzFeed's dress moment.

Now that we understand how these optimizations perform under various degrees of contention, we turn to some benchmarks modeling contention in real-world scenarios.

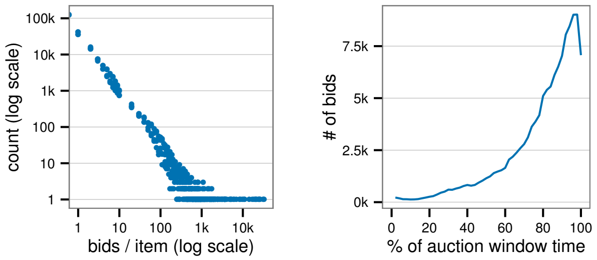


Figure 8. Our modified Rubis models real-world auctions with bids per item (*left*) following a power law, and the frequency of bids over the auction time window (*right*), with higher bidding rates closer to closing time.

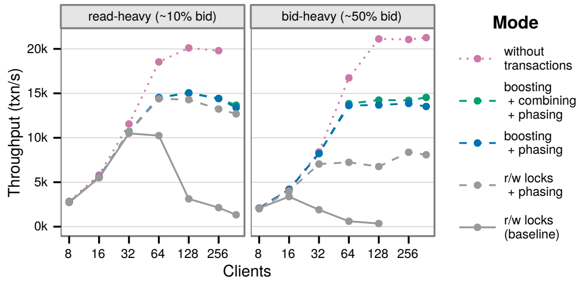


Figure 9. Throughput of Rubis. Contention between bids on popular auctions, especially close to their closing time, causes performance to drop for r/w locks, but bids commute, so boosting is able to maintain high throughput even under heavy bidding.

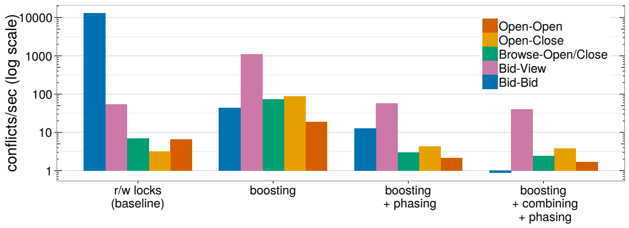


Figure 10. Breakdown of conflicts between Rubis transactions (minor contributors omitted) with 256 clients on bid-heavy workload (averaged). As predicted by Figure 3, boosting drastically reduces Bid-Bid conflicts, and phasing drastically reduces the remaining conflicts.

6.2. RUBiS

The RUBiS benchmark [4] imitates an online auction service similar to those described back in §2. There are 8 different kinds of transactions: `OpenAuction`, `ViewAuction`, `CloseAuction`, `Bid`, `Browse`, `AddComment`, `AddUser`, `ViewUser`, but `ViewAuction` and `Bid` dominate the workload. The benchmark specifies a workload consisting of a mix of these transactions and the average bids per auction that should result. However, the *distribution* of bids (by item and time) was unspecified.

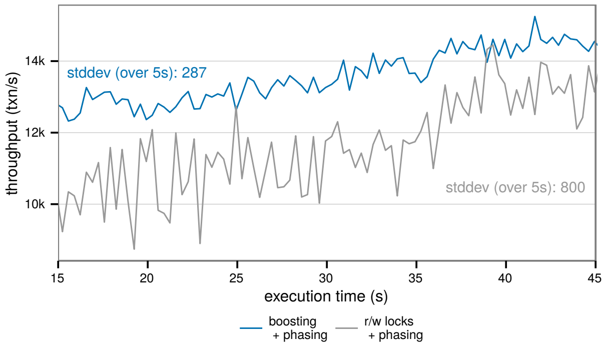


Figure 11. Trace of throughput over time for Rubis (both with phasing). The workload is characterized by bursts of high-contention bidding activity. Boosting smooths out the performance, reducing variability by 2.8x.

Our modified implementation more accurately models the bid distributions observed by subsequent studies [1, 2], with bids per item following a power law distribution and the frequency of bids increasing exponentially at the end of an auction (Figure 8 shows these distributions for one of our runs). 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 distribution with $\alpha = 1$.

Figure 9 shows the performance results for two different workloads: read-heavy (10% Bid transactions), and bid-heavy (50% bid transactions). For the read-heavy workload, phasing is sufficient to provide reasonable performance, as bids do not often come in at a high enough rate to require commutativity. However, during times of high bidding activity, modeled by our bid-heavy workload, it becomes important to allow bids to commute with one another, we are able to maintain the same throughput as the read-heavy workload, roughly 2x better than r/w locks and 68% of the non-transactional throughput. Considering the importance of getting bids correct, this seems like an acceptable tradeoff.

Our goal with Figure 10 is to compare our predictions of conflicts from Figure 3 with reality. We look at a subset of the conflicts between Rubis transactions, plotted with a log scale because of the extreme change in conflicts that occurs with our optimizations. The baseline is dominated by Bid-Bid conflicts which are drastically reduced by boosting (conflicts on `ItemBids` have gone away, the remaining conflicts come from other records not included in our simplified representation). On the other hand, the Bid-View conflicts have gone up; now that bids commute, there are more bids going

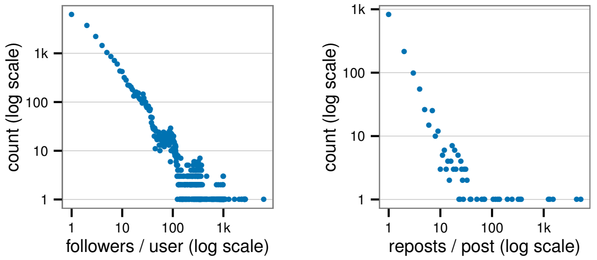


Figure 12. Power-law distributions in our Retwis benchmark. Distribution of followers per user (*left*) comes from the Kronecker synthetic graph generator. Number of reposts per post (*right*) is a result of the graph structure and our user model which favors reposting already-popular posts.

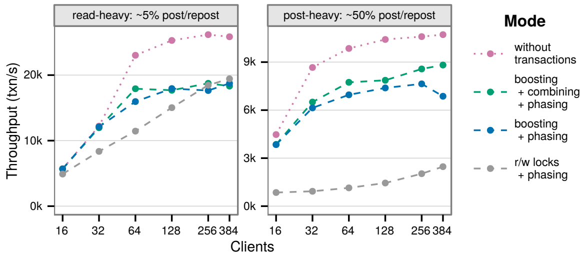


Figure 13. Throughput of Retwis. Boosting is essential during heavy posting because network effects lead to extreme contention on some records. Non-phasing results elided due to too many failed transactions.

through, so more chances of `ViewAuction` conflicting with them. Finally, we can see that with phasing and combining, all of the conflicts are further reduced.

Another interesting feature of auctions is the spikes in contention due to auctions closing, observed by prior work [1]. By looking at a trace of throughput over the execution of our benchmark (Figure 11), we observe those contention spikes as momentary drops in throughput. We see less variance over time with boosting enabled because the concurrency between bid transactions allows the datastore to keep up with the increased demand better.

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 [32]. 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 (represented as a sorted set). On top of Retwis's basic functionality, we added a "repost" action that behaves like Twitter's "retweet". Not being a true benchmark itself, Retwis doesn't specify a workload, so we simulate a realistic workload using a synthetic graph with power-law degree distribution and a simple model for user behavior for posting and reposting.

For our synthetic graph, we use the Kronecker graph generator from the Graph 500 benchmark [17] which is designed to produce the same power-law degree distributions found in natural graphs. We use a Kronecker graph of approximately 65,000 users, with an average number of followers of 16 (scale 16 with an edge-factor of 16). Figure 12 (left) shows that the distribution of followers per user follows a power law as intended.

We use a simple model of user behavior to determine when and which posts to repost. Each time we load the most recent posts in a timeline for a random user (uniformly selected), they are sorted by the number of times they have already been reposted, and a discrete geometric distribution (skewed toward 0) is used to select the number of these to repost. The distribution of reposts resulting from our model is shown in Figure 12 (right), which follows a power law distribution: a decent approximation of the "viral" propagation effect observed in real social networks. Note that a small number of posts are reposted so much that they end up on over a quarter of users' timelines.

Figure 13 shows throughput on two different workloads. Read-heavy models steady-state Twitter traffic, while the post-heavy workload models periods of above-average posting, such as during live events. We show only the results with phasing because the non-phasing baseline had too many transactions it was unable to complete. On the read-heavy workload, which models steady-state Twitter traffic, 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, Twitter is typically characterized as a non-transactional application because users are willing to tolerate minor inconsistencies. This tradeoff is clear when performance is as flat as the reader/writer performance is in these plots. However, when we look at the non-transactional throughput in these plots, we see that Claret achieves up to 82% of its throughput. Even without transactions, it suffers from the additional work caused by the higher ratio of posting, causing throughput (in terms of completed actions) to go down overall during high posting. With performance that close, it may be worth considering the benefits of using stronger consistency more often.

This embodies the philosophy behind Claret: if the true concurrency in the application can be exposed, then correctness need not be sacrificed, and applications which require correctness, like auctions, need not suffer for it. By leveraging the data structures programmers naturally choose to use, Claret exposes concurrency without burdening developers. It is worth noting that these experiments do not include weaker consistency that can be enabled by replicating shards in an eventually consistent way. Leveraging ADTs in such settings is something we plan on investigating next.

7. Related Work

7.1. ADTs in databases

ADT concepts were first used in databases to support custom indices [39, 40]. The first algorithms for leveraging ADT semantics for concurrency control come from the mid 1980s, by Schwarz, Herlihy and Weihl [16, 20, 34, 43]. That work introduced abstract locks to allow databases to leverage commutativity and also allowed user-defined types that expressed their abstract behavior to the database. Another contemporary system [38] used similar type-based locks with an early form of distributed transactions. However, since then, the concept has lost popularity. Claret revives these ideas and incorporates them into the modern world of key-value stores and web applications.

7.2. Improving Transaction Concurrency

Several recent systems have explored ways of exposing more concurrency between transactions. An old technique, transaction chopping [36], statically analyzes transactions and breaks them into smaller pieces that reduces the scope where locks are held, but potentially resulting in cascading rollbacks. A more recent system, Lynx [47], extends this with additional checks for commutative operations and allows conflicting transactions to interleave safely by enforcing consistent ordering. Salt [44] borrows the notion of chopping, but allows programmers to choose some transactions to execute using weaker consistency and isolation (termed *BASE* transactions). By focusing on "baseifying" only the highly contentious transactions, programmers can get most of the benefits of NoSQL systems while maintaining the strong ACID guarantees on the rest.

Most recently, Callas [45] introduced modular concurrency control which operates similarly to Salt but maintains ACID semantics for all transactions. Callas places transactions which may commute with each other into a separate group, which can use specialized concurrency control to execute them concurrently when it is safe to do so. Abstract locks, employed by Claret, cleanly expose concurrency among transactions, avoiding false conflicts from commuting operations, without needing to divide transactions into arbitrary groupings and without the complex multi-tiered locking strategy needed in Callas to manage inter-group dependencies. Other concurrency control techniques introduced by Callas, such as runtime pipelining are orthogonal and could usefully be applied in Claret.

7.3. Commutativity

Commutativity is well known, especially in distributed systems, for enabling important optimizations. Though the original work leveraging commutativity was in relational database systems with highly sophisticated concurrency control, recently, there has been a resurgence of systems without a predefined data model, such as NoSQL databases and transactional memory, leveraging commutativity.

Several NoSQL systems specialize for commutative operations to improve performance of serializable distributed transactions. Lynx [47] statically splits transactions into chains, but allows users to annotate parts of transactions as commutative to improve performance. Doppel [30], a multicore in-memory database, allows commutative operations on highly contended records to be performed in parallel *phases* with a technique called *phase reconciliation*. In the context of distributed transactional memory, HyFlow [25] combines multi-versioning and commutativity to reorder commutative transactions before others. HyFlow effectively combines boosting and phase reconciliation, but only if entire transactions commute. Claret exposes commutativity as a natural part of the application design, and allows more nuanced commutativity to be exploited within transactions through abstract locks.

Commutativity has been used in eventually consistent datastores to improve convergence. RedBlue consistency [27] treats commutative (blue) and non-commutative (red) operations differently, exploiting the convergence guarantees of blue operations to avoid coordination. Conflict-free (or convergent) replicated data types (CRDTs/CvRDTs) [35] force operations to commute by defining merge functions that resolve conflicts automatically. Riak [7] has implemented CRDTs for production use cases. Claret's strictly linearizable model exposes concurrency without relaxing consistency because CRDT behavior is often still counterintuitive.

Bloom and BloomL [3, 10] help programmers design applications in ways that do not require distributed coordination by restricting them to ensure monotonicity. This lets programmers avoid reasoning about inconsistency, but not all programs can easy be expressed in this way, whereas Claret can support any design, making the question of commutativity a performance issue only.

7.4. Complex datatypes in NoSQL

There is a recent trend toward supporting complex data types in NoSQL datastores. At the forefront is Redis [32], one of the most popular key/value stores, which supports many data types and complex operations on them, exposed through a very wide but fixed set of commands, but does not support distributed transactions. Hyperdex [14, 24] supports atomic operations on lists, sets, and maps, as well as JSON-style documents. Hyperdex's Warp transaction system [15] reasons dynamically about dependencies between transactions to avoid unnecessary conflicts, but does not consider operation commutativity. Many other popular datastores such as MongoDB [29] support atomically updating parts of JSON documents but do not leverage commutativity.

In these systems, data types are used primarily to provide programmers with a library of functionality they can reuse. Claret leverages those same data structures to expose concurrency optimizations, and provides interfaces for programmers to extend it with custom types.

8. Conclusion

Reading is easy. If all applications were purely browsing static content, then contention would not be a problem; geo-replication and caching could handle most problems with skew. However, the reality is that interaction is key to these applications, so they must deal with real-world contention. Claret helps mitigate contention by allowing programmers to expose application-level concurrency through ADTs, which the datastore leverages in the transaction protocol and client library. Together, these optimizations lead to significant speedups for transactions in high contention scenarios, without hurting performance on lighter workloads, 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] Lisa Baertlein. Ellen's Oscar 'selfie' crashes Twitter, breaks record. http://www.reuters.com/article/2014/03/03/us-oscars-selfie-idUSBREA220C320140303, March 2014.

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

[8] 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.

[9] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *IEEE INFOCOM Conference on Computer Communications*. Institute of Electrical & Electronics Engineers (IEEE), 1999. doi:10.1109/infcom.1999.749260.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

[20] 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. URL http://doi.acm.org/10.1145/308386.308440.

[21] 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. URL http://dx.doi.org/10.1145/78969.78972.

[22] 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.

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

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

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

[33] 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.

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

[35] 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.

[36] 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. URL http://dx.doi.org/10.1145/211414.211427.

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

[38] 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. URL http://dx.doi.org/10.1145/323627.323641.

[39] 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.

[40] 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.

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

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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 Prin-*

ciples (SOSP), SOSP '13, pages 276–291, 2013. ISBN 978-1-4503-2388-8. doi:10.1145/2517349.2522729.