# 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

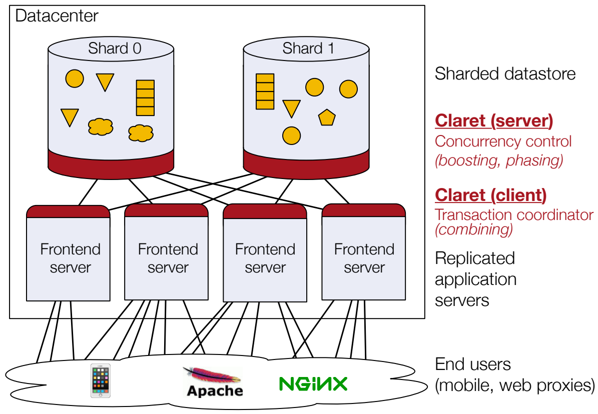


**Figure 1.** *System model:* End-user requests are handled by replicated stateless application servers which all share a sharded datastore. Claret operates between these two layers, extending the datastore with ADT-aware concurrency control (*Claret server*) and adding functionality to the app servers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

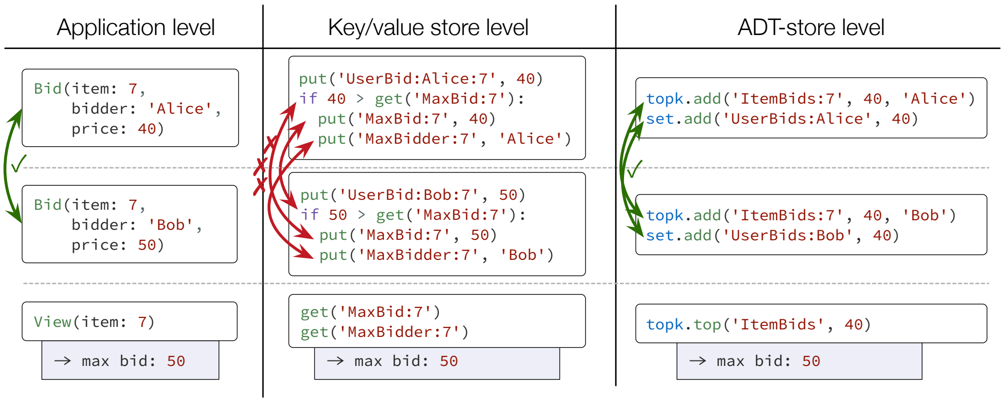


**Figure 2.** At the application level, many transactions ought to commute with one another, such as these Bid transactions, but when translated down to put and get operations, this knowledge is lost.

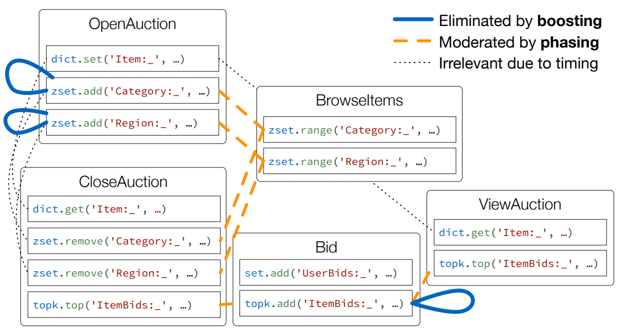


**Figure 3.** High-level overview of important Rubis transactions, implemented with ADTs. Lines show conflicts between operations, many of which are either eliminated due to commutativity by boosting or mediated by phasing.

### 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.