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 behav-

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.

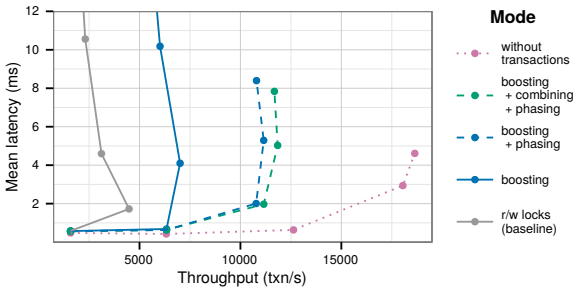

Figure 5. Performance of raw mix workload (50% read, zipf: 0.6) with increasing number of clients, plotted as throughput versus latency. Boosting (abstract locks) eliminate conflicts between `adds`, and phasing helps operations acquire locks in an efficient order, resulting in a 2.6x improvement in throughput, 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