# Claret: Using Data Types for Highly Concurrent Distributed Transactions

Brandon Holt, Irene Zhang, Dan Ports, Mark Oskin, Luis Ceze
University of Washington
{bholt,iyzhang,drkp,oskin,luisceze}@cs.washington.edu

March 7, 2015

**Abstract**

Out of the many NoSQL databases in use today, some that provide simple data structures for records, such as Redis and MongoDB, are now becoming popular. Building applications out of these complex data types provides a way to communicate intent to the database system without sacrificing flexibility or committing to a fixed schema. Currently this capability is leveraged in limited ways, such as to ensure related values are co-located, or for atomic updates. There are many ways data types can be used to make databases more efficient that are not yet being exploited.

We explore several ways of leveraging abstract data type (ADT) semantics in databases, focusing primarily on commutativity. Using a Twitter clone as a case study, we show that using commutativity can reduce transaction abort rates for high-contention, update-heavy workloads that arise in real social networks. We conclude that ADTs are a good abstraction for database records, providing a safe and expressive programming model with ample opportunities for optimization, making databases more safe and scalable.

# Claret

## Using Data Types for Highly Concurrent Distributed Transactions

Brandon Holt, Irene Zhang, Dan Ports, Mark Oskin, Luis Ceze
{bholt,iyzhang,drkp,oskin,luisceze}@cs.washington.edu

## Why NoSQL and eventual consistency?

– *Scalable:* horizontally scale services to billions of users
– *Flexible*, ad-hoc data model, no fixed schema

## Why *not?*

– No one understands eventual consistency
– NoSQL databases are blind to programmer intent

## Abstract data types as records

Abstract data types (ADTs) provide an abstraction that helps span the gap between serializability and eventual consistency. Building apps out of *semantically meaningful ADTs* rather than primitive strings and integers is *natural, flexible,* and *optimizable.*

ADTs are specialized for particular use cases, providing an expressive interface to the database. Each data type defines operations available on it. Concurrency control and replication decisions can be delegated to the data structure as well. Choosing the most specific ADT for the job maximizes optimization potential. Example: `UniqueIDGenerator` rather than `Integer`.

## Commutativity

If two operations *commute* then they executing them in either order produces the same result. Commutativity is a property of *pairs of operations on an ADT*, and is a function of the operations, their arguments, and the current abstract state. The complete set of rules for an ADT are its *commutativity specification*.

**Commutativity Specification for Set**

| method: | commutes with: | when: |
|---|---|---|
| `add(x): void` | `add(y)` | $\forall x, y$ |
| `remove(x): void` | `remove(y)` | $\forall x, y$ |
| | `add(y)` | $x \neq y$ |
| `size(): int` | `add(x)` | $x \in Set$ |
| | `remove(x)` | $x \notin Set$ |
| `contains(x): bool` | `add(y)` | $x \neq y \vee y \in Set$ |
| | `remove(y)` | $x \neq y \vee y \notin Set$ |
| | `size()` | $\forall x$ |

### Transaction boosting

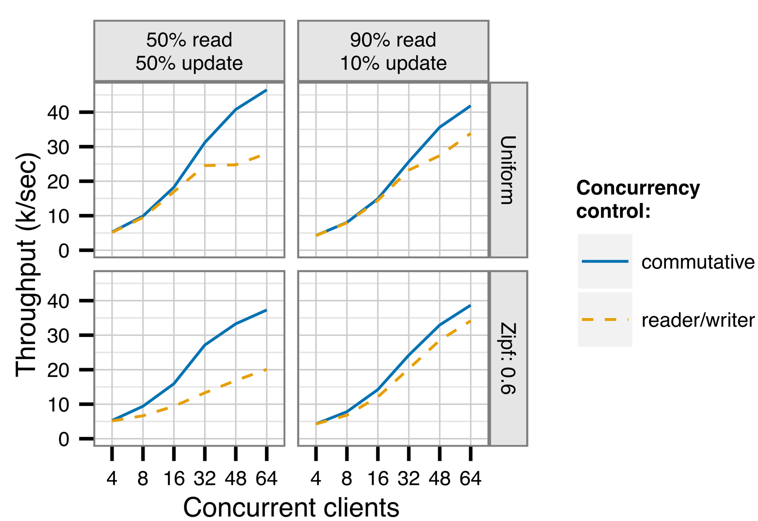
*Leveraging commutativity can reduce transaction aborts.* When operations in different transactions are known to commute, even if they are updates to the same record, the two transactions can safely execute concurrently.

### Combining

When operations are associative and commutative, they can be *combined* first before being applied to the data structure itself. This can drastically *reduce contention on hot records* (lots of concurrent updates).

## Evaluation

We compare concurrency control which employs transaction boosting to allow concurrent commutative operations against a more traditional system using reader/writer locks (so read-only operations can be concurrent, but only one updating operation in flight).

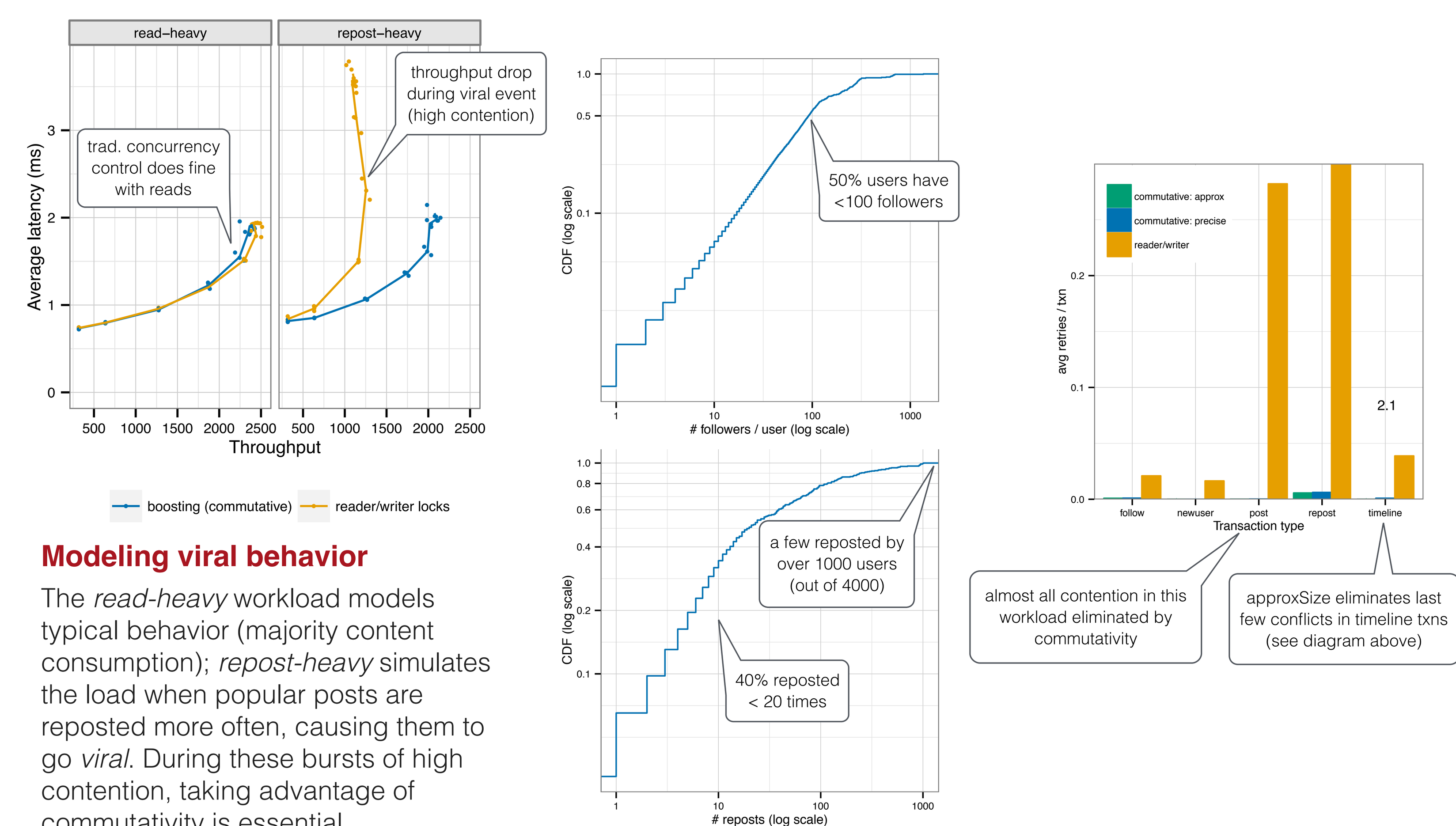



**Raw Operation Mix:** Set

Random set operations (add/remove/size/getRandom) over 10,000 keys either uniformly distributed or a skewed Zipfian distribution. Leveraging commutativity is more important on update-heavy workloads, provided they commute with each other. It is even more important for *skewed* workloads where some keys are hit much more often, leading to *high contention.*

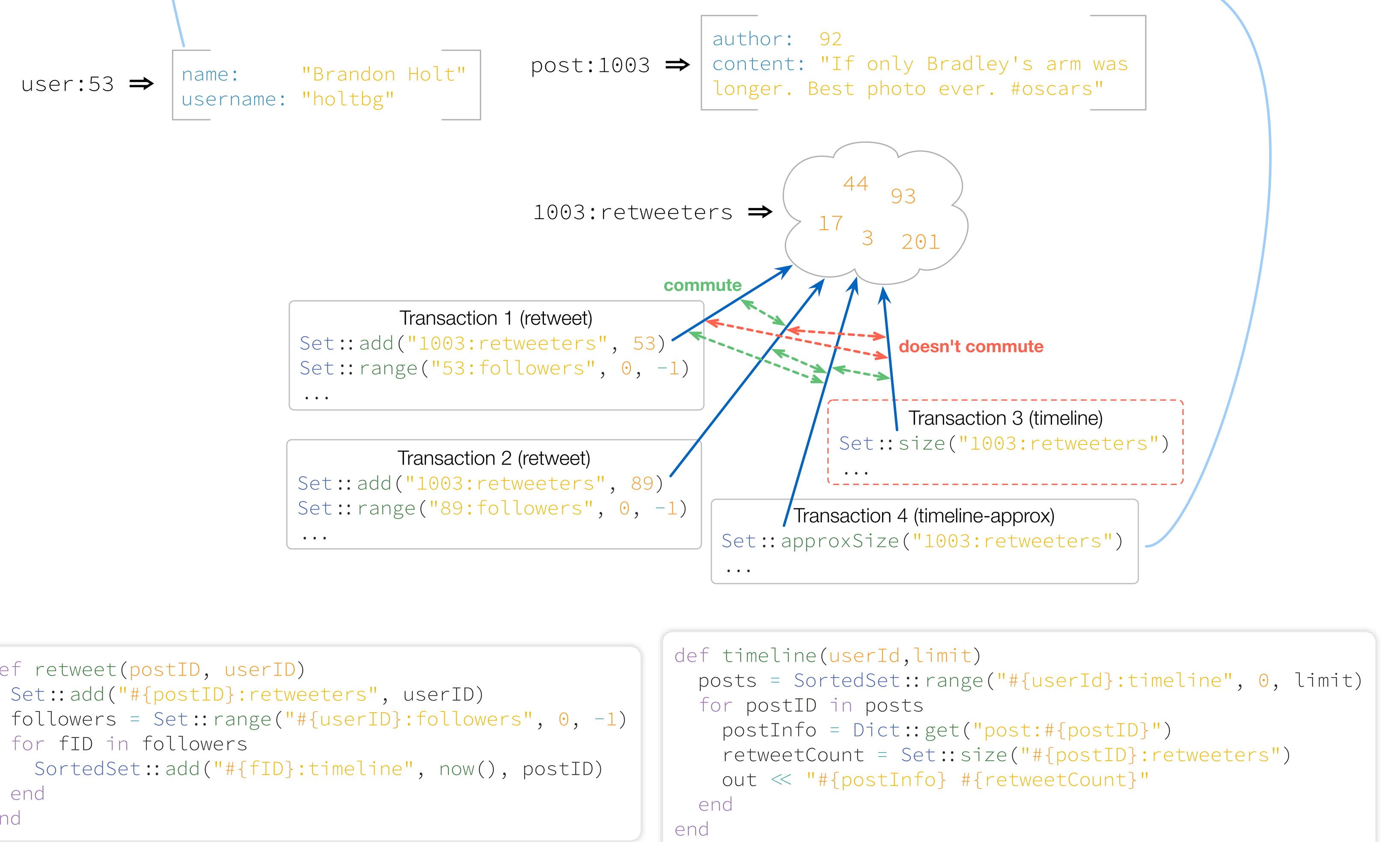
**Case study:** Twitter clone

We also evaluate on a Twitter-like app, Retwis, using a *realistic synthetic graph* (Kronecker), and workload generated by a simple user model (new user, follow, new post, repost, load timeline).




**Modeling viral behavior**

The *read-heavy* workload models typical behavior (majority content consumption); *repost-heavy* simulates the load when popular posts are reposted more often, causing them to go *viral.* During these bursts of high contention, taking advantage of commutativity is essential.

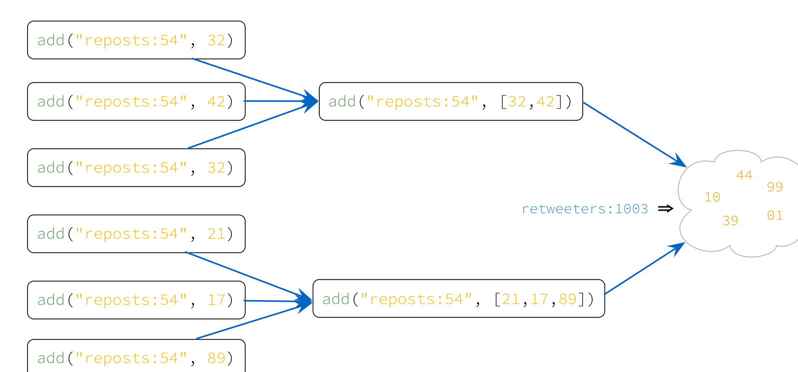



```
def retweet(postID, userID)
  Set::add("#{postID}:retweeters", userID)
  followers = Set::range("#{userID}:followers", 0, -1)
  for fID in followers
    SortedSet::add("#{fID}:timeline", now(), postID)
  end
end
```

```
def timeline(userId,limit)
  posts = SortedSet::range("#{userId}:timeline", 0, limit)
  for postID in posts
    postInfo = Dict::get("post:#{postID}")
    retweetCount = Set::size("#{postID}:retweeters")
    out << "#{postInfo} #{retweetCount}"
  end
end
```

**Building an app:** Twitter

The diagram above shows a sketch of the Twitter clone, Retwis, used in our evaluation. The social graph is composed of Dict records holding user data, and Sets tracking each user's followers. We sketch the code for performing a retweet and reading a user's timeline. Two retweet transactions commute with each other. Reading the timeline, however, which reads the exact size of the set, doesn't commute and must abort. However, if the "retweet count" can be approximated, then the transaction succeeds.



**Combining:** merge commutative & associative operations together (in parallel) and apply them in bulk to the contended record.

## Approximate data types

Machine learning and data analytics algorithms use *probabilistic data types (hyperloglog, bloom filter, count min sketch)* to make it feasible to compute statistics over extremely large datasets. For example, *HyperLogLog* can be used to count the number of distinct elements (approximate set cardinality).

We propose relaxing an ADT's semantics in a similar way to permit more concurrency and therefore improve scalability. Each operation can define an error tolerance, and the ADT's tracks the bounds of all in-flight operations and uses them to determine when new operations can be permitted.

### Example: approximate set size in Retwis

Imagine we allow a 5% error tolerance when getting the size of a set. A set encoding who has retweeted one of *my* tweets (~0 elements) would have to be precise, but the retweet set for Ellen's selfie has 3.4M entries, so we could allow 170k `add` or `remove` ops while the `approxSize` is in flight. That's a lot more concurrency, which would allow many of those transactions to succeed that wouldn't have before.

```
template< class T >
class Set {
  Interval<int>       size(float error); // [ value, value+size()*error )
  Probabilistic<int>  size(float error); // hyperloglog
  Probabilistic<bool> contains(T o);     // bloom filter
  Inconsistent<T>     random();          // racy read
};
```

## CRDTs

One can also implement data types that behave like CRDTs (conflict-free replicated data types) do in eventual consistency. These could be freely replicated on multiple shards and lazily kept up to date. Behavior will be more difficult to reason about, but it will be *limited to just records of this type,* and if implemented correctly, these records should perform comparable to eventually consistent systems.