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

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.

1. Introduction

The move to non-relational (NoSQL) databases was motivated by a desire for scalability and flexibility. People found that by giving up strong consistency, they could better scale services to millions or billions of users while meeting tight performance goals. Because of inherent uncertainty in timing and connectivity, in many cases users are likely to accept minor inconsistencies such as two tweets being out of temporal order or needing to retry an action. In such cases, relaxed consistency feels like a natural solution, but it leaves much to chance: there is likely no guarantee that more significant inconsistencies are impossible. When consistency is critical, developers can enforce stronger guarantees manually, or use serializable transactions in systems like Google's Spanner [4], but this leaves them with two extremes with a significant performance gap. If certain parts of an application can tolerate imprecision, why not capture those properties in the programming model? Is there a way programmers can express the semantics they desire succinctly and precisely, helping the database optimize performance and scalability, without sacrificing flexibility?

We propose abstract data types (ADTs) as the solution. Rather than limiting the records in databases to primitive types like strings or integers, raising them to more complex data types provides a richer interface, exposing ample opportunities for optimization to the database and a precise mechanism to express the intentions of programmers. In this work we explore several ways of leveraging commutativity and data types to improve database performance and allow programmers to make tradeoffs between performance and precision, starting by demonstrating one way of using commutativity to reduce transaction aborts.

2. Commutativity

Commutativity is well known, especially in distributed systems, for enabling important optimizations. Since the 80s, commutativity has been exploited by database systems designers [7, 19] within the safe confines of relational models, where complete control of the data structures allows systems to determine when transactions may conflict. Recently, commutativity has seen a resurgence in systems without a predefined data model, such as NoSQL databases and transactional memory. Eventually consistent databases use commutativity for convergence in work such as RedBlue consistency [14] and conflict-free replicated data types (CRDTs) [17]. Other systems specialize for commutative operations to improve transaction processing, such as Lynx [21] for tracking serializability, Doppel [15] for executing operations in parallel on highly contended records, and HyFlow [12] for reordering operations in the context of distributed transactional memory. We propose unifying and generalizing these under the abstraction afforded by ADTs.

PaPoC'15, April 21, 2015, Bordeaux, France.

ACM 978-1-4503-3537-9/15/04…$15.00.
http://dx.doi.org/10.1145/2745947.2745951

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()`	$\forall x$

Table 1: Commutativity Specification for Set.

Though *commutativity* is often discussed in terms of an operation commuting with all other operations, it is actually more nuanced. If a pair of operations commute, then executing them in either order will produce the same result. Using the definitions from [13], 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 1. There are actually many valid specifications which expose less than the maximum commutativity, but may be cheaper to implement.

Transaction boosting. If two operations on the same record in two different transactions commute, then the transactions can safely execute concurrently, even though they both update the record. This technique is known as *transactional boosting* [11]. This straightforward use of commutativity was shown to significantly improve abort rates in software transactional memory. In §4, we show how we applied it to distributed transactions.

Combining. Associativity, often paired with commutativity, allows some concurrent operations to be *combined* before being applied to the data structure itself. *Combining* [10, 18, 20] can drastically reduce contention on shared data structures. This technique could be applied to hot records, similar to *splitting* in Doppel [15], to avoid bottlenecking on a single shard.

3. Data type selection

Choosing an ADT with semantics specialized for a particular use case gives the system the best chance of scaling performance. For example, rather than using a counter, which must return the next number in the sequence (which is difficult to scale, as users of TPC-C [6] know well) For example, an application needing to generate unique IDs should not use a counter, which must return the next number in the sequence, because this is very difficult to scale (as users of TPC-C [6], which explicitly requires this, know well). Instead, a `UniqueID` type succinctly expresses that non-sequential IDs are okay, which can be implemented very efficiently. By allowing approximations or non-determinism, performance may be further improved.

Probabilistic data types such as *bloom filters* [2], *hyperloglogs* [8], and *count-min sketches* [5] trade off accuracy (within fixed bounds) for better performance or storage. Twitter's streaming analytics system [3] and many machine learning algorithms leverage these to handle high data volume, and we expect similar benefit.

Conflict-free replicated data types (CRDTs), which were invented for eventual consistency, can actually be fit into our model as well, as a new kind of data type. Copies of a record could exist in different shards, asynchronously updating each other. By defining the same kind of *merge* function as traditional CRDTs, these copies could ensure they all converge to the same state. Clients may find them more difficult to reason about but might make that tradeoff in parts of the application where it otherwise cannot scale.

4. Evaluation

To demonstrate the efficacy of leveraging commutative operations in transactions, we built a simple prototype key-value store, modeled after Redis, that supports complex data types for records, each with their own set of operations. Our experiments were carried out with 4 shards on 4 local nodes, each with 8-core 2GHz Xeon E5335 processors and standard ethernet connecting them.

4.1 Transaction protocol

The transaction protocol employs standard two-phase commit and two-phase locking with retries to ensure isolation, atomicity. We implement a number of standard optimizations, such as delaying acquiring locks for operations that don't return a value to the *prepare* step so that locks are held for as short a time as possible. However, there is one step that is non-standard in order to support complex data types where rolling back state changes would be non-trivial.

To support transactions with arbitrary data structure operations, each operation is split into two steps: *stage* and *apply*. During transaction execution, each operation's *stage* method attempts to acquire the necessary lock and may return a value *as if the operation has completed* (e.g. an "increment" speculatively returns the incremented value). When the transaction is prepared to commit, *apply* is called on each staged operation to actually mutate the underlying data structure. This allows operations to easily be un-staged if the transaction fails to acquire all the necessary locks, without requiring rollbacks.

Commutativity comes into play in the locking scheme. Using the algorithms from [13] and our commutativity specifications, we design an abstract lock for each record type. Our `SortedSet`, for instance, has an `add` mode which allows all insertion operations to commute, but disallows operations like `contains` or `size`. As a baseline, we implement a standard reader/writer locking scheme that allows all read-only operations to execute concurrently, but enforces that only one transaction may modify a record at a time.

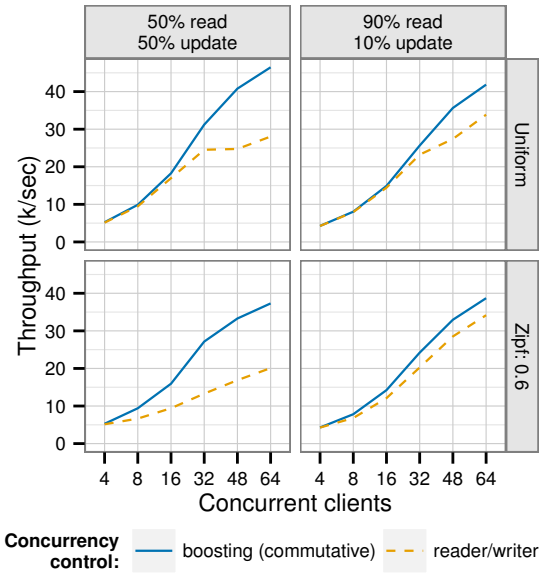

Figure 1: Throughput of raw Set operations.

4.2 Microbenchmark: Set operations

We first evaluate performance with a simple workload consisting of a raw mix of `Set` operations randomly distributed over 10,000 keys. We use both a uniform random distribution as well as a skewed Zipfian distribution with a coefficient of 0.6. In Figure 1, we see that commutative transactions perform strictly better, showing the most pronounced benefit over the more update-heavy, skewed workload.

4.3 Case study: Retwis

To understand performance on a typical web workload, we use *Retwis*, a simplified Twitter clone designed originally for Redis [16]. Data structures such as 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".

We simulate a realistic workload using a synthetic graph with power-law degree distribution and clients that randomly select between Retwis transactions including "add follower", "new post", and "repost", executing them as fast as they can.

Rather than simply approximating the skew in real-world workloads with a Zipfian distribution as many other systems do, we simulate the behavior of social networks with a realistic synthetic graph and a simple model of user behavior for posting and reposting.

For our synthetic graph, we use the Kronecker graph generator from the Graph 500 benchmark [9]. This generator is designed to result in graphs with the same power-law degree distribution found in natural graphs. Figure 2 shows the cumulative distribution function (CDF) of the number of followers per user for the synthetic graph of approximately

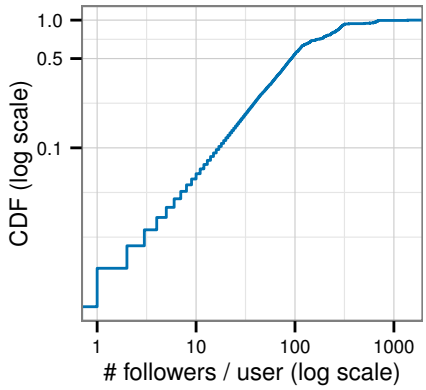

Figure 2: CDF of the number of followers for users generated by the Kronecker synthetic graph generator, matching the power-law degree distribution of natural graphs.

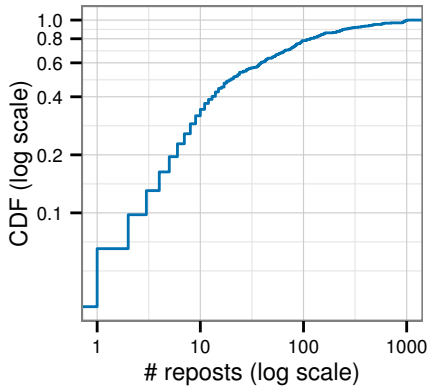

Figure 3: CDF of the number of times a post is reposted, matching traces from real workloads. Due to the power-law graph structure, if users tend to reposts popular recent posts on their timeline, it results in another power-law distribution. Note that that some posts are reposted to over a quarter of the graph (4000 total users).

4000 users, with an average number of followers of 16 (scale 12 with edgefactor of 16 in Graph500's terms). Most users should have relatively few followers; we see that roughly 50% have fewer than 100 followers, while a very small number of users have over 1000 followers.

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. This results in the "viral" propagation effect that is observed in real social networks. Figure 3 shows the distribution of the number of times a post was reposted, which is again a power-law distribution. Note that a small number of posts are reposted so much that they end up on over a quarter of users' timelines.

Figure 4: Throughput of social network workload (Retwis) with 4000 users. Leveraging commutativity prevents performance from falling over even when posts spread virally (repost-heavy).

Figure 4 shows the results of this simulation. When most of the traffic is content consumption (reading timelines), both systems perform well enough. However, when we simulate a workload where clients repost popular posts from their timelines, we see a viral propagation effect, where a large fraction of the users get and share a post. As Twitter came to a standstill when Ellen DeGeneres's Oscar selfie set a retweeting record [1], so too does our baseline fall over. But with commutativity, performance continues to scale even under this highly contentious load.

References

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

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[3] O. Boykin, S. Ritchie, I. O'Connell, and J. 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.

[4] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, 2012.

[5] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[6] T. P. P. Council. Tpc-c. http://www.tpc.org/tpcc/.

[7] A. Fekete, N. Lynch, M. Merritt, and W. Weihl. Commutativity-based locking for nested transactions. *Journal of Computer and System Sciences*, 41(1):65–156, Aug. 1990.

[8] P. Flajolet, Éric Fusy, O. Gandouet, and F. Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *International Conference on Analysis of Algorithms*, 2007.

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

[10] D. Hendler, I. Incze, N. Shavit, and M. 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.

[11] M. Herlihy and E. 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 '08, pages 207–216, 2008.

[12] J. Kim, R. Palmieri, and B. Ravindran. Enhancing Concurrency in Distributed Transactional Memory through Commutativity. In *EuroPar 2013*, pages 150–161. 2013.

[13] M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, and K. Pingali. Exploiting the Commutativity Lattice. In *Conference on Programming Language Design and Implementation*, PLDI '11, pages 542–555, 2011.

[14] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, 2012.

[15] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase Reconciliation for Contended In-Memory Transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 511–524, Broomfield, CO, Oct. 2014.

[16] S. Sanfilippo. Redis. http://redis.io/.

[17] M. Shapiro, N. Preguiça, C. Baquero, and M. 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.

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

[19] W. E. Weihl. Commutativity-based Concurrency Control for Abstract Data Types. In *Proceedings of the Twenty-First Annual Hawaii International Conference on Software Track*, pages 205–214, 1988.

[20] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributing hotspot addressing in large-scale multiprocessors. *Computers, IEEE Transactions on*, 100(4):388–395, 1987.

[21] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 276–291, 2013.