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

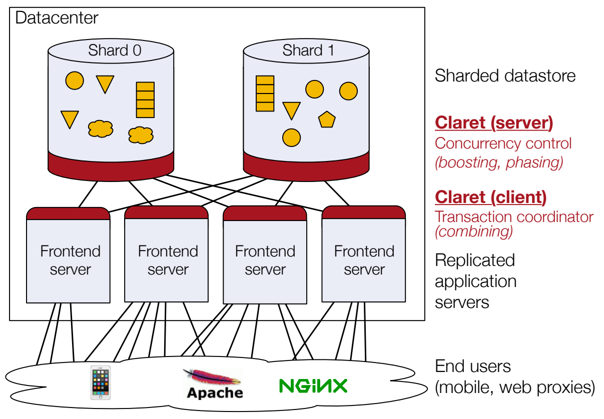

Figure 1. *System model:* End-user requests are handled by replicated stateless application servers which all share a sharded datastore. Claret operates between these two layers, extending the datastore with ADT-aware concurrency control (*Claret server*) and adding functionality to the app servers to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

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.