Stochastic Caching for Read Optimized Eventually Consistent Datastores

Abstract

In this work we describe a new cache mechanism to radically improve read performance of an eventually consistent database without sacrificing consistency guarantees and actually resulting in improved availability. Our system scales quadratically in the number of nodes and can support many hundreds of concurrent connections. Early results also suggest that bandwidth utilization is competitive with existing solutions and aligned to the capabilities of existing 2G cellular networks.

1. Introduction

Traditional databases have relied on strong consistency models with names like "Linearizability" and "Strict Serializability". However with the rise of a new wave of 'web scale' NoSQL databases, researchers have begun to examine relaxed consistency models. Eventual consistency has emerged as a popular choice due to its ease of implementation and abundant opportunities for optimization. Our major contribution is to show that there is considerable room for further improvement. Towards this end, we introduce StochasticDB, the first database to implement stochastic caching.

2. System Design

Our system consists of server component and a user space library which are described in the following sections.

2.1 Server

StochasticDB servers are assumed to reside in geographically distributed datacenters. This choice provides fault tolerance in the face of even catastrophic failures impacting entire regions.

Each replica keeps a full copy of the entire database state in memory to maintain data integrity even in the event of $n-1$ crash failures. In the event that memory capacity is insufficient, we rely on OS level paging capabilities. On our toy data sets we have not found this to be an issue.

DOI: http://dx.doi.org/10.1145/(to come)

2.2 Client Library

The StochasticDB client library provides a key value interface to the user application. Currently only PUT and GET functions are implemented. It will turn out that more exotic multi-put and multi-get operations are unnecessary to achieve satisfactory performance, though we plan to include them in future versions for completeness.

The key idea of StochasticDB is the stochastic caching algorithm implemented by the client library. All clients maintain a cache of recently queried documents which they use to fill subsequent requests, which initially starts out empty. While it would be possible to populate the cache with the contents of the database, this would introduce a high bandwidth cost.

Existing databases generally require at least one expensive network round trip to serve GET's. By contrast, in StochasticDB GET's are served directly out of the cache, and any items not present simply return NOT_FOUND.

Astute readers will notice that the design as described does not provide eventual consistency. This concern is easily reconciled: after serving each GET, the client library queries a pseudo random number generator and uses it to decide whether to refresh the associated cache entry. The entry is refreshed when the value returned is less than p. Selecting the proper value of p is essential for the successful operation of StochasticDB, and methods for doing so are left as future work. In the interim we direct the reader to a series of contradictory blog posts describing the issue.

For the remainder of the paper we use the following empirically derived formula:

$$p = \frac{h\sqrt{a_1 n + a_2^2}}{W * t}$$

where n is the average number of clients, t is the round trip latency, W is the total energy consumed processing a single client request, h is Planck's constant, and variables a_1, a_2 represent the real roots of the following polynomial:

$$3x^4 + 21x^3 + 2x - 1 = 0$$

2.3 Backoff Mechanism

To avoid overloading the server replicas all clients implement a specialized backoff strategy. Like in other similar systems, backoff prevents individual replicas from receiving requests at a higher rate than they are able to process. Normally this would result in degraded performance as clients were only allowed to make requests more slowly than they would like. However in StochasticDB, we devise a special scheme to retain throughput performance even when the client is being throttled.

The key insight comes from the fact that *any* requests can be filled directly from the client side cache. This observation enables us to have clients modulate the number of requests they submit to the server independently of their own load. By careful configura-

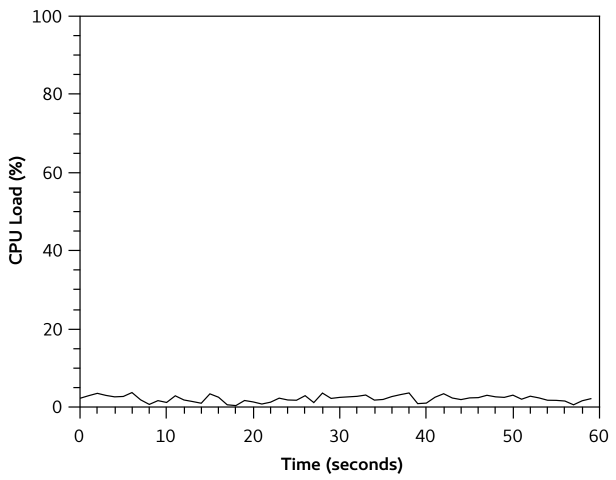

Figure 1. CPU load of a StochasticDB server while running a benchmark with 5000 clients (simulated).

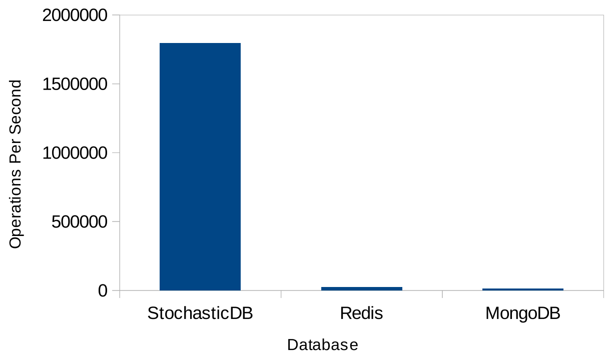

Figure 2. Throughput on workload C from the Yahoo Cloud Serving Benchmark

tion, we were able to tune the system to be almost infinitely salable in the number of clients.

3. Experiments

All experiments were conducted on a single laptop equipped with 16 GB of DDR3 RAM, an Intel Core i5-3230M CPU clocked at 2.60GHz, and an Intel Centrino Advanced-N 6205 Wi-Fi Adapter.

In Figure 1 we plot CPU load for a single StochasticDB server simultaneously instance serving requests for 5000 clients. Observe that the machine remains nearly idle for the duration of the test, just as was anticipated.

In Figure 2 we compare the throughput (expressed in operations per second) for a range of systems. The experiment was conducted using Workload C from the Yahoo Cloud Serving Benchmark suite, with an additional driver added to accommodate StochasticDB. The driver directly links in our client library forwarding requests accordingly. Figure 3 shows the 99th percentile latency running the same experiment.

Impressive performance numbers are not just limited to average cases. One might anticipate that stochastic caching could cause a small fraction of requests to take drastically longer than others. However, the reordering technique described above means that

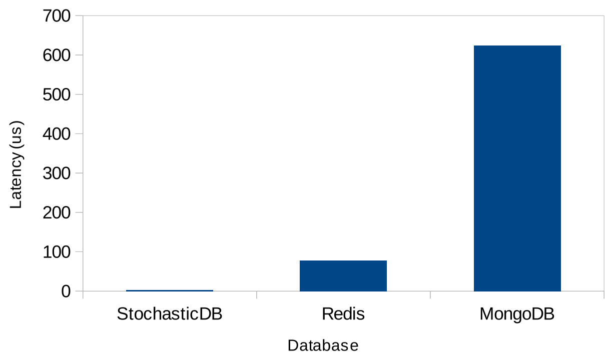

Figure 3. 99th percentile latency on workload C from the Yahoo Cloud Serving Benchmark

even 95% and 99% tail latencies are still only 3 microseconds (contrast this to over 600 microsecond 99% tail latency for MongoDB).

4. Related Work

Databases were one of the earliest kinds of systems to come into widespread use, and have been an active area of research ever since. Accordingly, the field has an abundance of seminal papers describing various advances that have been achieved over the years.

5. Future Work

The current prototype only provides eventual consistency. However, we believe that the same techniques could extend to more strict consistency models as well.

For instance, by modifying the cache implementation it should be possible to provide "read your own writes" (but not anybody else's) consistency with relative ease. Other consistency models should follow naturally.

6. Conclusion

We presented the first ever database to employ stochastic caching, and demonstrated some of the phenomenal properties that it can provide. Throughput was orders of magnitude higher than existing options, while latency declined to less than a tenth and was frequently below the cost of a single RPC. As far as we know, no other system has yet to break this significant latency barrier.

Finally, StochasticDB is open source software [link] and bindings are available for a wide variety of mainstream languages[1].

References

[1] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language ALGOL 60. *Commun. ACM*, 3(5):299–314, May 1960. ISSN 0001-0782. doi: 10.1145/367236.367262.

[2] G. E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, December 1960. doi: 10.1145/367487.367501

[3] L. Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1994.

[4] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, April 1960. doi: 10.1145/367177.367199

[1] By which we mean, available for volunteers to implement.