CS 274
Project Report
Winter 2016

University of California, Santa Barbara

Lucid DB

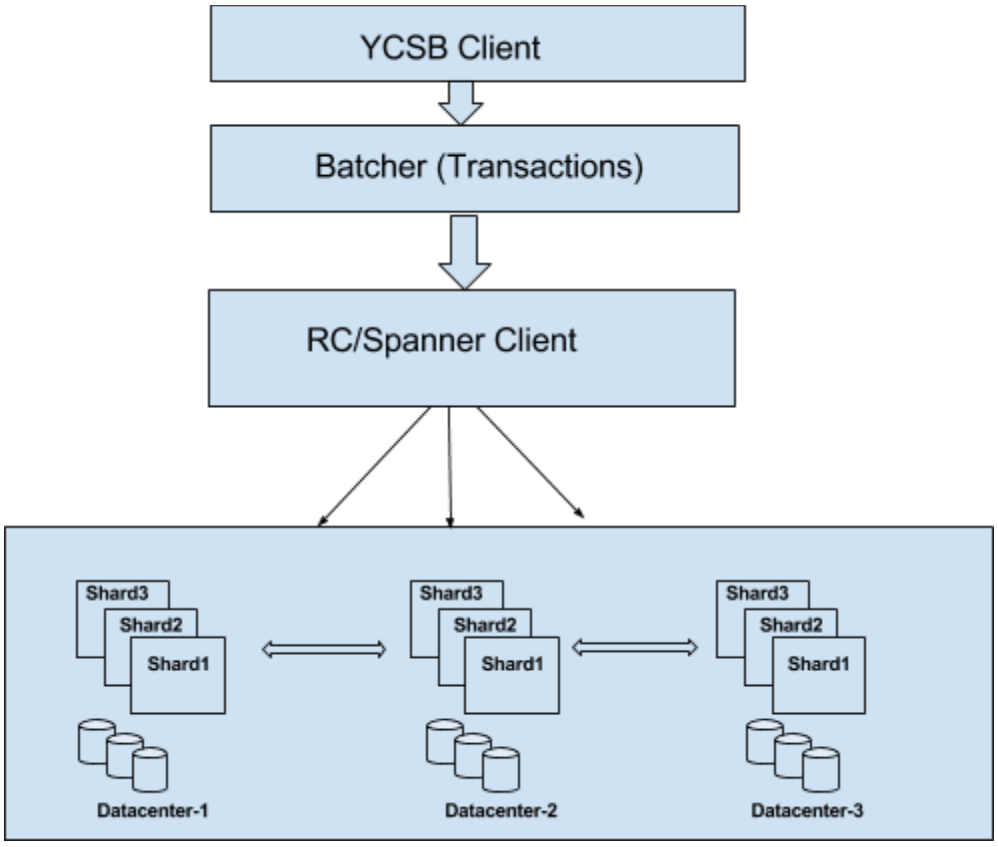
Tanuj Mittal, Sachin Rathod, Aviral Takkar

Overview

We implemented the following two protocols for geo-replicated data management:

1. Spanner [1]
2. Replicated Commit [2]

We used an open source implementation of Raft consensus protocol - copycat [3], as the underlying consensus mechanism for our Spanner 2PC implementation. For Replicated Commit, we used broadcasting between servers to determine majority vote. The datastore for both Spanner and Replicated Commit was implemented as in-memory hashmap for storing key-value pairs. So as to make the comparison fair, we used the in-memory structures for storing the log-replication data for copycat.



Architecture

We have Server and Client implementations for both the protocols. The Servers communicate with each other using custom communication channels asynchronously. The transactions are simulated by trapping all requests made using a batching mechanism. For inserts, YCSB Client calls into the DB implementation of our protocols which in turn calls into our Batcher implementation and wait for the result. Batcher uses the client implementations to perform actual insert operation. For reads, YCSB Client directly calls into the client implementation of protocol and return the result.

Implementation Details

Batcher for YCSB:

We implemented a module called Batcher, which is responsible for batching all write operations from the YCSB client in order to simulate transactions. Only write operations are simulated as transactions. Read queries are treated as individual operations. As soon as YCSB Client calls a method in our implementation of DB for insert, we intercept the request, add it to the batch and make the YCSB Client thread wait. The transaction is started as soon as the batch size becomes 5 or batch times out. Batch times out when more than 50ms has been spent after the last add in the batch. Once the result for the transaction is received, the waiting YCSB Client threads for the transaction are notified and the result is returned.

Spanner:

We used an open-source implementation (Copycat [3]) of Raft consensus algorithm to replicate spanner state machine. Set of spanner servers responsible for the same shard at different datacenters (replicas) implement this. Each shards has one of the replicas as a leader. Spanner Client receives and handles reads and write transaction requests from Batcher. It first determines shards involved in a transaction and chooses one of the leader shards as a coordinator randomly (shard of first write object). The coordinator handles 2PC communication between Leaders. Each leader maintains a lock table for current transactions. Our lock table has fixed number of locks (configurable) with keys mapped to one of the locks. Deadlocks are not handled in the current implementation, but can be incorporated using timeouts. Reads are performed using Copycat which uses quorum protocol.

Replicated Commit:

We used a simple broadcasting mechanism to replicate the state machine between servers in a datacenter and across datacenters. RC Client receives and handles reads and write transaction requests from Batcher. For each transaction, a shard at each datacenter is chosen as coordinator by a RC client. The coordinator is responsible for ensuring that all servers in its datacenter process the transaction correctly. It also holds all the locks for that particular transaction in its datacenter. We maintain a fixed number of available locks, and each key is mapped into a single lock, upon which it may have to wait until it's available. Coordinators across datacenters broadcast their ability to commit and the decision to commit is taken when a coordinator hears from a majority of other coordinators. The client returns success when it hears from a majority of coordinators. This implementation does not guarantee freedom from deadlocks as there is limited deadlock detection at servers. Reads are performed using a quorum protocol, i.e. from majority of the servers and the latest version is chosen.

Benchmarks

We used YCSB [4] for benchmarking our implementations with two kind of workloads.

Workloads	readheavy1k	equal1k
recordcount	1000	1000
operationcount	1000	1000
readallfields	true	true
readproportion	**0.95**	**0.50**
updateproportion	0	0
scanproportion	0	0
insertproportion	**0.05**	**0.50**
requestdistribution	zipfian	zipfian

We used 3 eucalyptus instances with each one acting as a single datacenter and 3 shards each. Hence there were 9 server instances running in total, 3 on each eucalyptus instance. Ping delay between eucalyptus instances at the time of benchmarking was ~0.4ms. We used artificial delays for simulating datacenter latencies.

Communication Type	Latency (ms)
Inter-datacenter	50
Intra-datacenter	0
Client to datacenter	20

Single threaded for readheavy1k workload

Type	Benchmark	Spanner	Replicated Commit
OVERALL	Runtime (ms)	84669	86280
OVERALL	**Throughput (ops/sec)**	**11.81**	**11.59**
READ	Operations	949	959
READ	**Avg Latency (us)**	**72932**	**81295**
READ	Min Latency (us)	71040	78848
READ	Max Latency (us)	309247	405247
READ	95th %ile Latency(us)	74559	85375
READ	99th %ile Latency (us)	93311	88511
READ	Return=OK	949	959
INSERT	Operations	51	41
INSERT	**Avg Latency (us)**	**301532**	**201414**
INSERT	Min Latency (us)	269056	171648
INSERT	Max Latency (us)	528895	451839
INSERT	95th %ile Latency(us)	384767	250367
INSERT	99th %ile Latency (us)	402687	451839
INSERT	Return=OK	51	41

Single threaded for equal1k workload

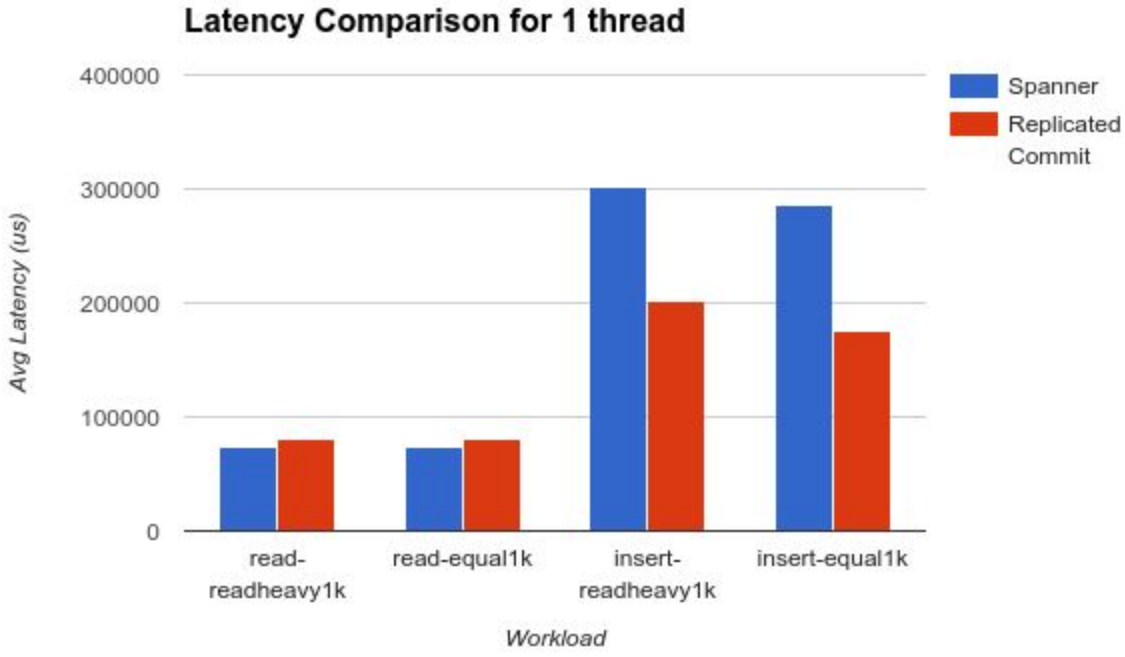
Type	Benchmark	Spanner	Replicated Commit
OVERALL	Runtime (ms)	183962	129408
OVERALL	**Throughput (ops/sec)**	**5.44**	**7.73**
READ	Operations	480	487
READ	**Avg Latency (us)**	**74025**	**81567**
READ	Min Latency (us)	71616	79104
READ	Max Latency (us)	262911	390655
READ	95th %ile Latency(us)	76223	84671
READ	99th %ile Latency (us)	107071	88383
READ	Return=OK	480	487
INSERT	Operations	520	513
INSERT	**Avg Latency (us)**	**285155**	**174710**
INSERT	Min Latency (us)	263424	169856
INSERT	Max Latency (us)	717311	426751
INSERT	95th %ile Latency(us)	324607	180095
INSERT	99th %ile Latency (us)	385535	216191
INSERT	Return=OK	520	513

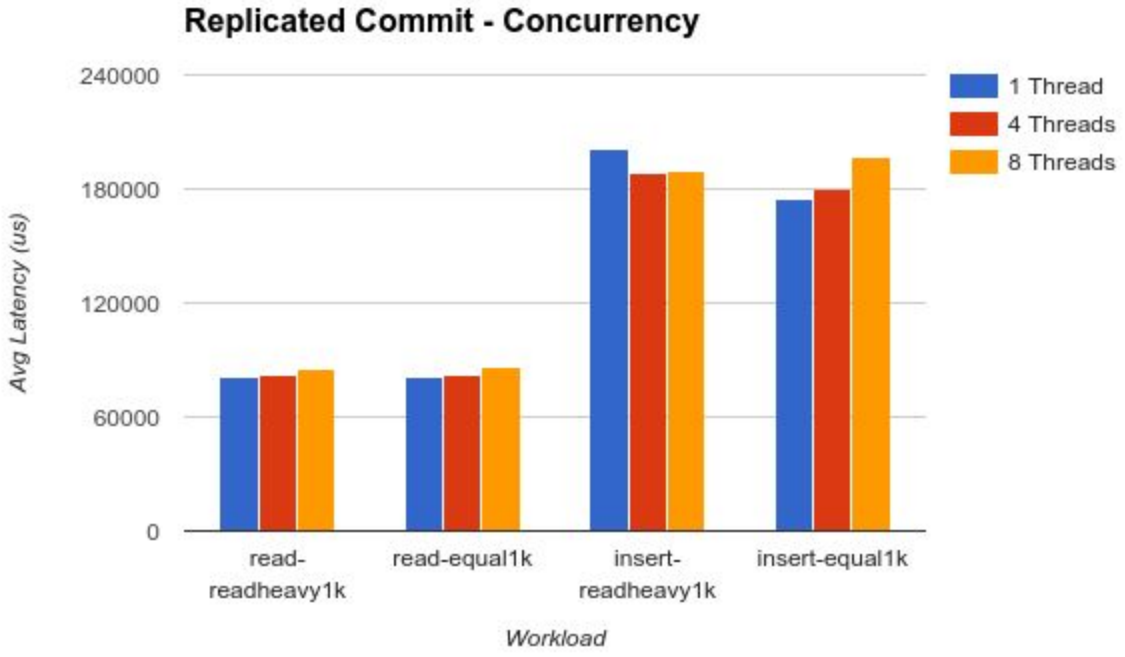
Replicated Commit for readheavy1k workload

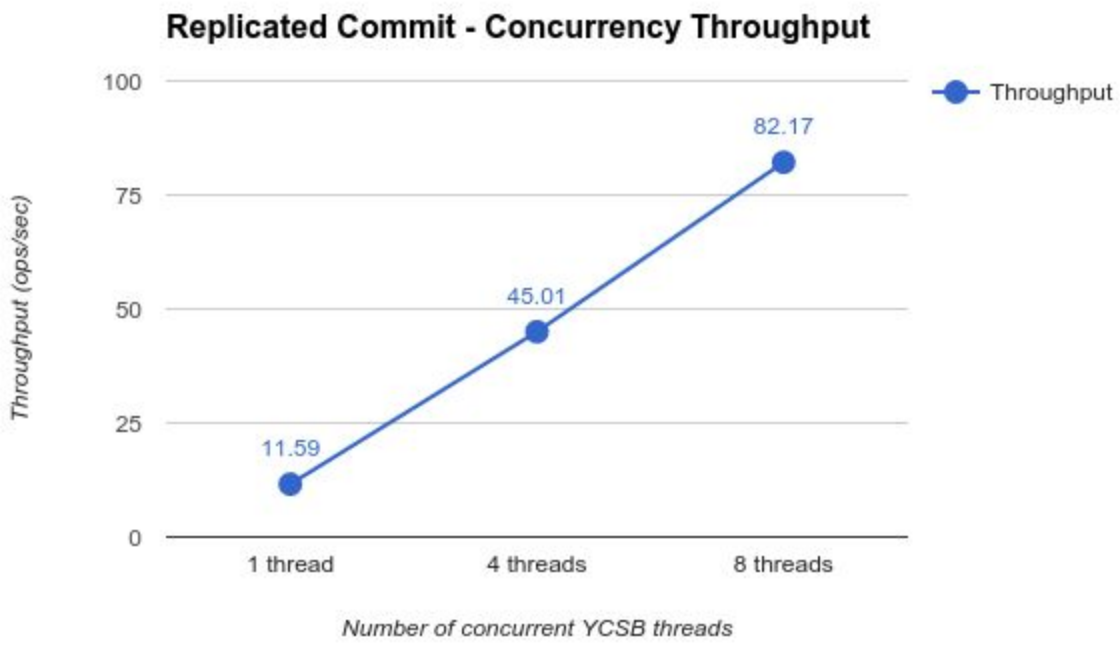
Type	Benchmark	1 thread	4 threads	8 threads
OVERALL	Runtime (ms)	86280	22219	12170
OVERALL	**Throughput (ops/sec)**	**11.59**	**45.01**	**82.17**
READ	Operations	959	953	945
READ	**Avg Latency (us)**	**81295**	**82221**	**84812**
READ	Min Latency (us)	78848	78720	78272
READ	Max Latency (us)	405247	409855	436223
READ	95th %ile Latency(us)	85375	87615	88959
READ	99th %ile Latency (us)	88511	92031	128831
READ	Return=OK	959	953	945
INSERT	Operations	41	47	55
INSERT	**Avg Latency (us)**	**201414**	**188715**	**189480**
INSERT	Min Latency (us)	171648	137472	135552
INSERT	Max Latency (us)	451839	755711	807423
INSERT	95th %ile Latency(us)	250367	216447	214783
INSERT	99th %ile Latency (us)	451839	755711	275455
INSERT	Return=OK	41	47	55

Replicated Commit for equal1k workload

Type	Benchmark	1 thread	4 threads	8 threads
OVERALL	Runtime (ms)	129408	33890	18375
OVERALL	**Throughput (ops/sec)**	**7.73**	**29.51**	**54.42**
READ	Operations	487	482	496
READ	**Avg Latency (us)**	**81567**	**82552**	**85950**
READ	Min Latency (us)	79104	78656	78272
READ	Max Latency (us)	390655	441087	458239
READ	95th %ile Latency(us)	84671	88127	91327
READ	99th %ile Latency (us)	88383	94527	102911
READ	Return=OK	487	482	496
INSERT	Operations	513	518	504
INSERT	**Avg Latency (us)**	**174710**	**180354**	**196393**
INSERT	Min Latency (us)	169856	130624	91840
INSERT	Max Latency (us)	426751	2637823	2664447
INSERT	95th %ile Latency(us)	180095	199935	228863
INSERT	99th %ile Latency (us)	216191	749567	2547711
INSERT	Return=OK	513	518	504


Latency Comparison for 1 thread
400000
300000
200000
100000
0
Avg Latency (us)
Spanner
Replicated Commit
read-readheavy1k
read-equal1k
insert-readheavy1k
insert-equal1k
Workload


Replicated Commit - Concurrency
240000
180000
120000
60000
0
Avg Latency (us)
1 Thread
4 Threads
8 Threads
read-readheavy1k
read-equal1k
insert-readheavy1k
insert-equal1k
Workload



Conclusion

- We observed that read latencies for both Spanner and Replicated commit are similar. This was because in our implementation, reads are done from a majority.
- Writes, on the other hand, are about 1.5x faster in Replicated commit. This can be attributed to less number of inter-datacenter communication trips in replicated commit. Reads are about 2x-3x faster than write as it does not require 2PC communication.
- The test for concurrency in replicated commit shows increased throughput, while read latencies remain the same and write latencies increase slightly (due to contention).

We conclude that Replicated Commit outperforms Spanner in terms of Throughput and average read/write latencies. Our future work would include comparing handling of concurrent transactions between these two protocols.

References

1. 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. 2012. **Spanner: Google's globally-distributed database.** In*Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*(OSDI'12). USENIX Association, Berkeley, CA, USA, 251-264.

2. Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. 2013. **Low-latency multi-datacenter databases using replicated commit.** *Proc. VLDB Endow.* 6, 9 (July 2013), 661-672. DOI=http://dx.doi.org/10.14778/2536360.2536366

3. Copycat - An open source implementation of the Raft consensus algorithm. https://github.com/atomix/copycat

4. Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. **Benchmarking cloud serving systems with YCSB.** In *Proceedings of the 1st ACM symposium on Cloud computing* (SoCC '10). ACM, New York, NY, USA, 143-154. DOI=http://dx.doi.org/10.1145/1807128.1807152