Case Study:

Required a database with high write throughput while maintaining good random read latency. The system will run on commodity hardware so the performance should not degrade too much if there are failures. Sample application for such a case might be recording user behavior on a website, where logs are organized by subdomain. Activity will be visible to the user so random reads are required, and there may be engineers using the database to find buggy user interfaces.

Performance Discussion:

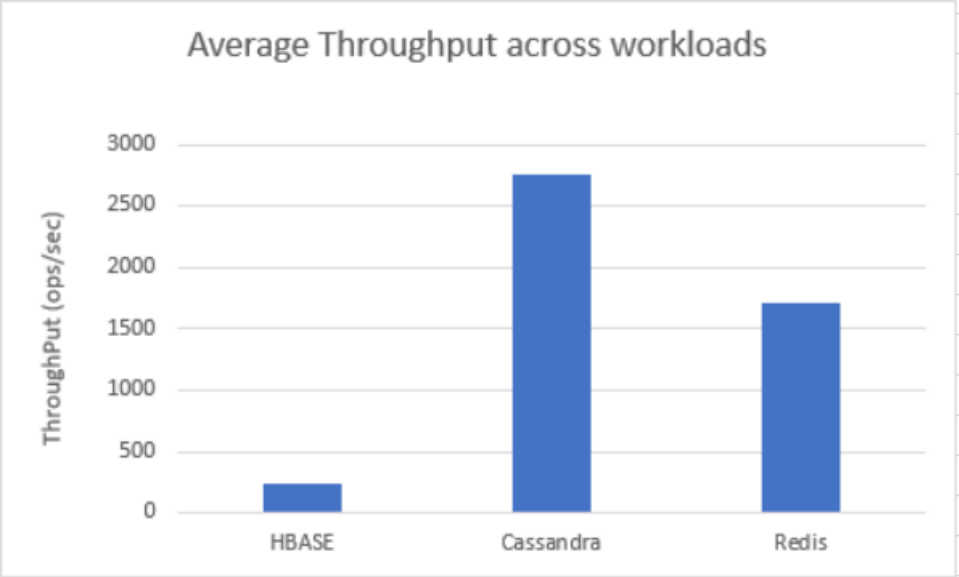
We installed the database systems on KVM virtual machines, each simulating a single core 3.4GHz cpu with a cache size of 4096kb and 4gb of RAM. For the performance tests we used YCSB with 6 different workloads to test the performance characteristics under each use case. The metrics reported are the 95^{th} percentile, we chose this metric as we wanted that under most cases the user behavior would be reported, but timeouts would be aggressive to prevent the user experience from degrading.

Both cassandra and hbase were tested with two different configurations to simulate node failure, but redis was tested with only one, because if there is a node failure then it requires manual intervention to bring the cluster back to a valid state.

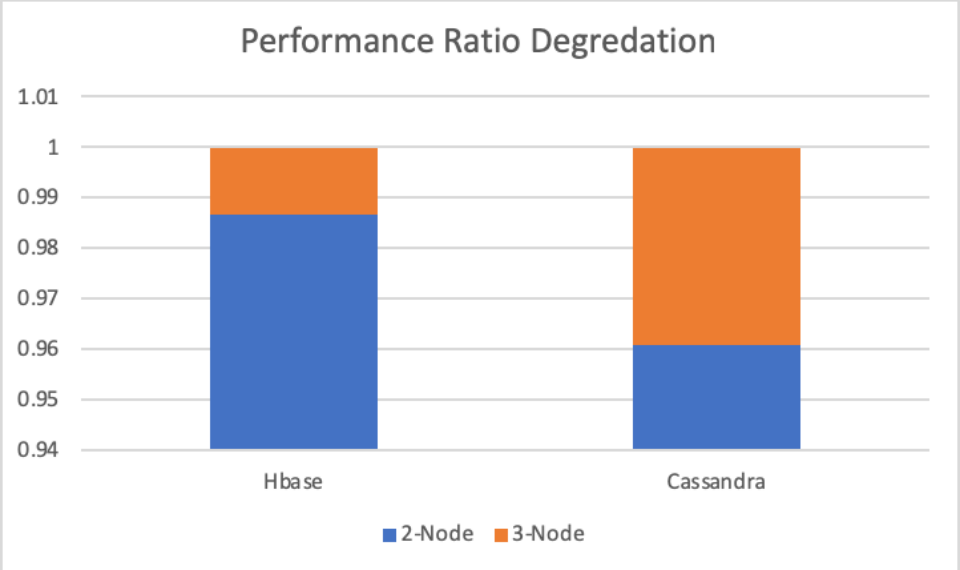
HBase was tested with 3 regionservers and one nameserver, each regionserver was also running the HDFS process. After running the tests for 3 regionservers, one of the servers was killed and the tests were run for a second time.

Cassandra was run with 3 nodes in the cluster (this was to give a fair test so that cassandra did not have more throughput by the virtue of having more nodes). Then a node was killed and the tests were run again.

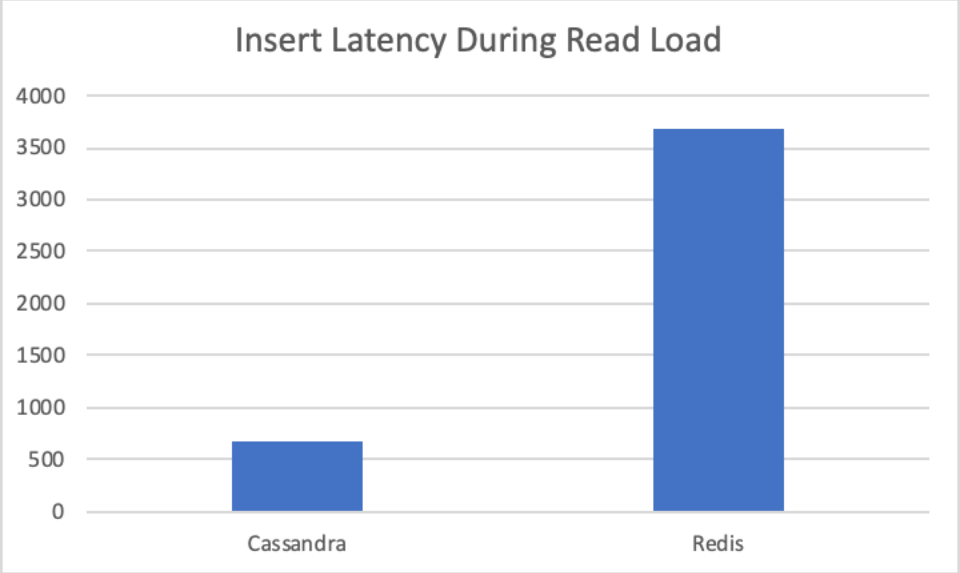
Redis was run with 6 nodes, 3 master and 3 slave. There are additional nodes here because redis requires slave to tolerate any node failure, so if there was any latency introduced by replicating keys to slaves, it was necessary to capture that in the tests. For writes to masters, there was still 3 nodes as in the other configurations, but with the replication to the slaves.



From this figure it's clear that generally speaking, for the workloads that were tested, Cassandra outperforms both redis and hbase, and HBASE is significantly worse than Redis. This average includes both optimal conditions (3 write servers per database) and failure scenario with only 2 write servers per database.



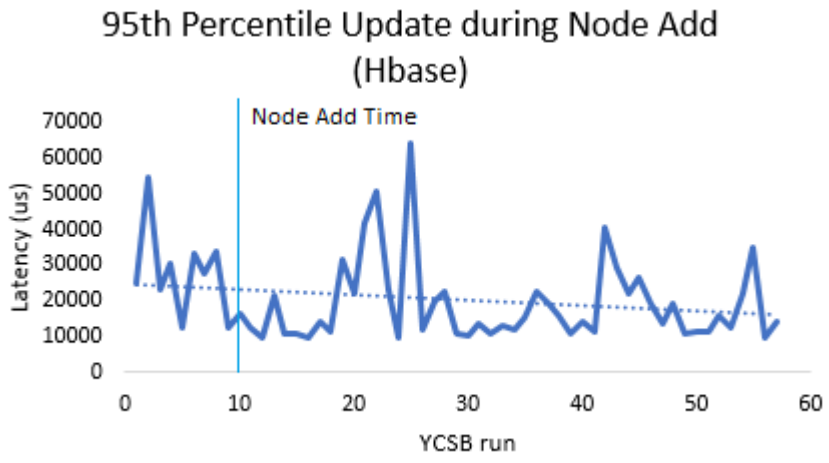
This chart illustrates the performance degradation of the performance when running with one less node. Both systems have less than a 5% performance degradation when running under non optimal conditions, but HBase is more resilient than the Cassandra server.

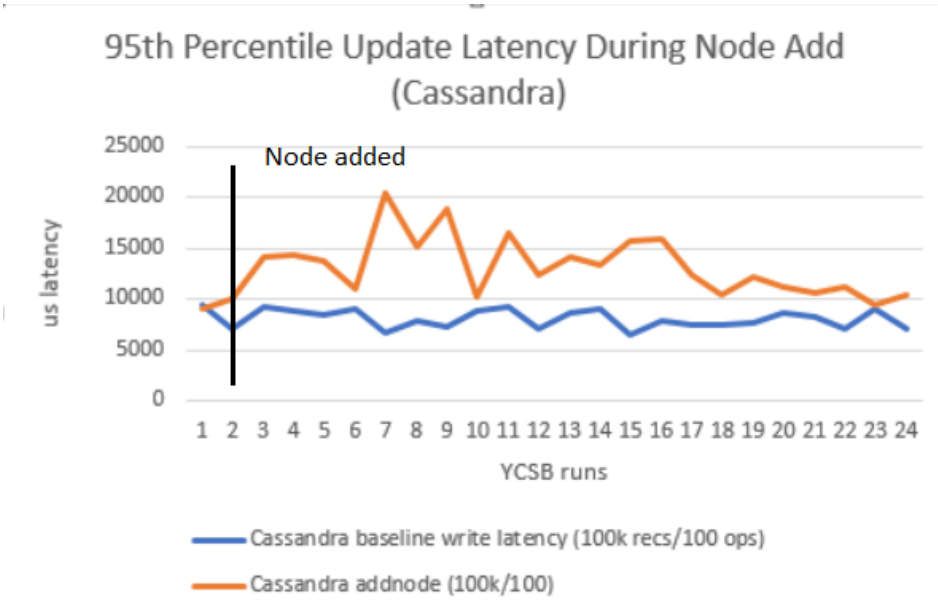


For the use case presented, where write latency needs to always be low as possible even during read operations, It's clear that Cassandra outperforms Redis by this metric.

Node Add Operation

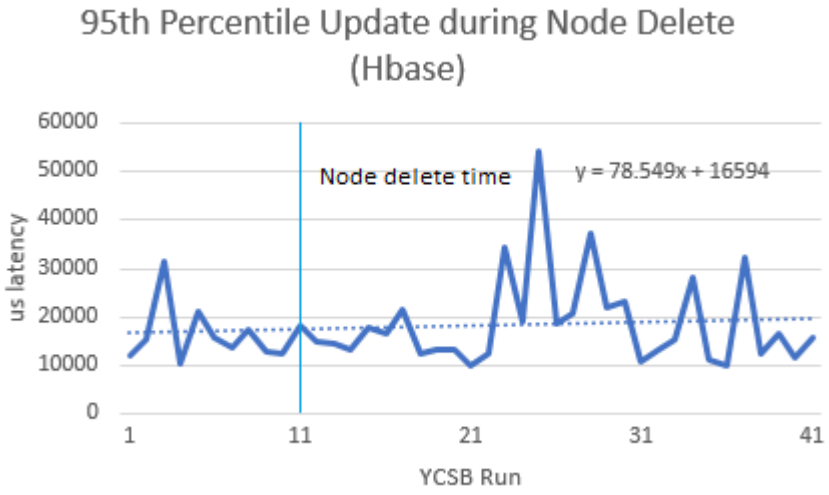
For the Node Add Operation we creeated a long running YCSB jobs (approximately 100k operations) to create server load, then a short running ycsb jobs (approx 100 ops) was run in the background to measure the performance. This was due to the fact that there was little measurable change with just running a medium size YCSB job, the servers needed to already be under load for the node add to have any measurable impact, even with that the reduction in latency was not that signficant for HBase. For Cassandra, there was an increase in latency of roughly 50% when a node was added, and the system took about a minute to recover. 95th Percentile seemed to have the most measurable impact by node join time. The following graphs show the measured performance from consecutive ycsb runs over the course of 2 minutes.

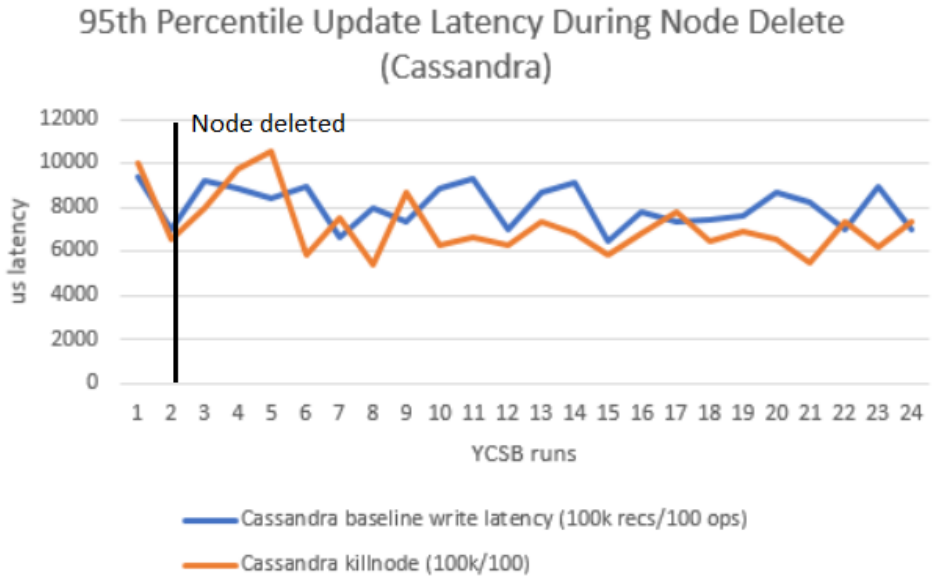




Node Deletion Operation

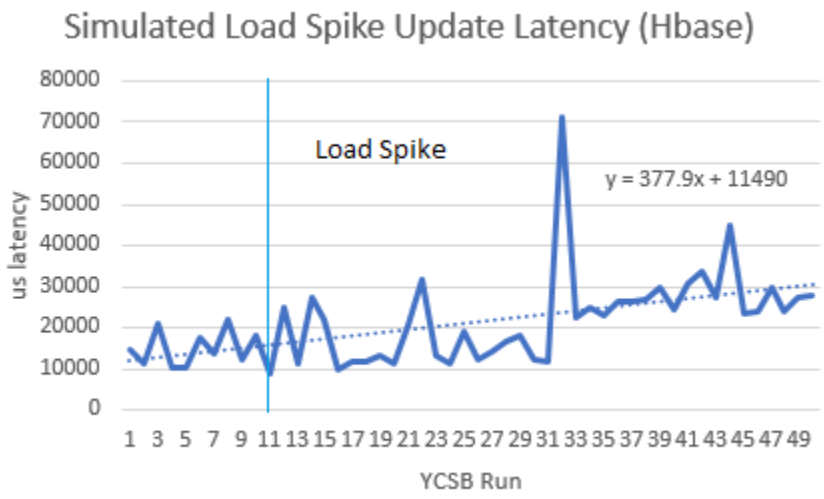
The procedure is the same for this metric, but instead of adding a node, a virtual machine was shut down to simulate a node failure. For Cassandra this had very little affect on latency, because the system was able to immediately redirect to the other node on which the data was replicated.

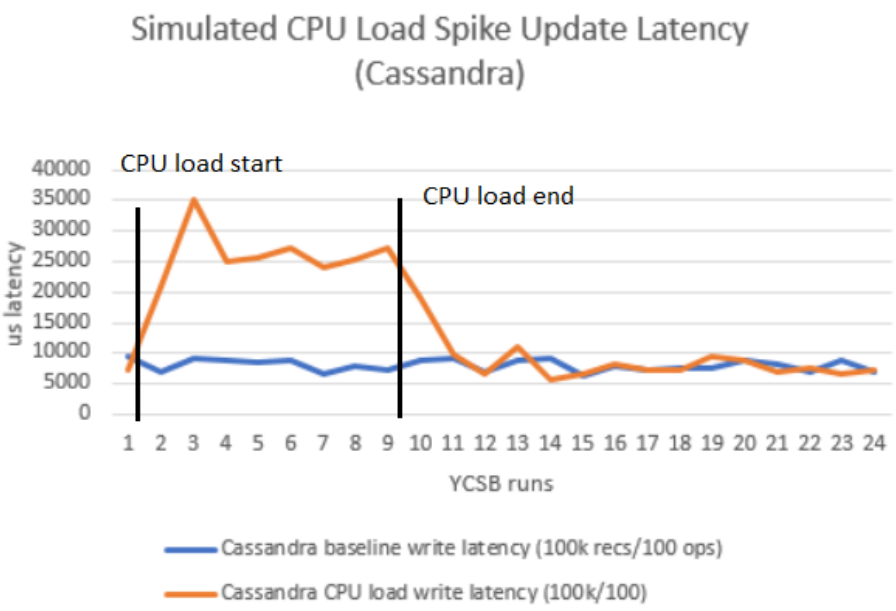




Simulated Load Spike On Single Node

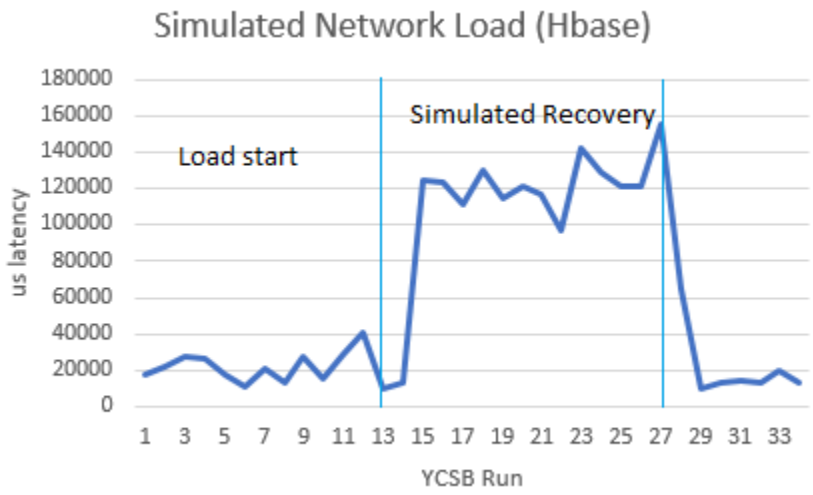
To measure how the database performs during poor load balancing that resulted in high CPU usage, one of the Virtual Machines in the cluster had a command run on it ($ yes > /dev/null &) that consumed nearly all the CPU. The result for Hbase was pretty significant latency increase (even more so than node failure) this implies that during a node failure the system recovers more quickly than during a node "partial" failure or node performance degradation. Cassandra had a similar result, with latency increasing by around 200% while the CPU was being used, and returning to normal very quickly as soon as the load was reduced.

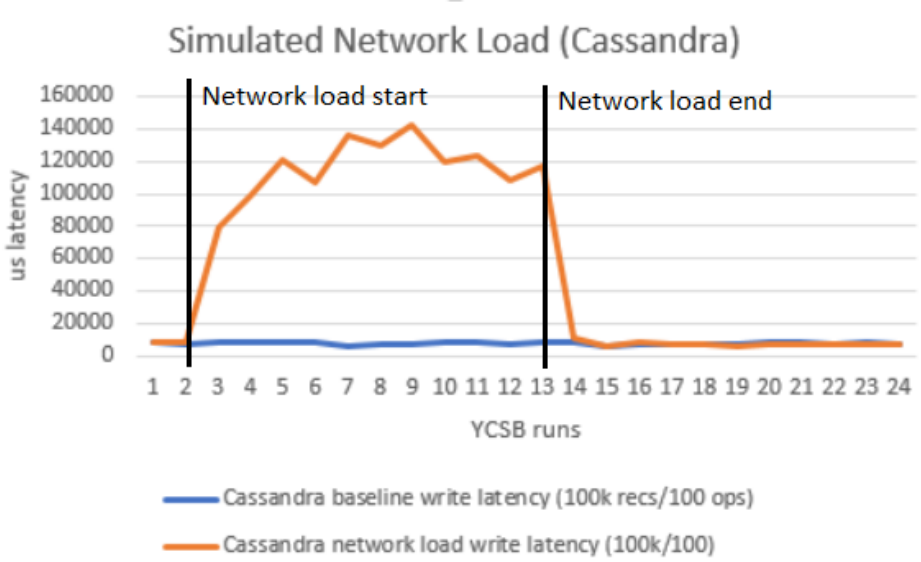




Simulated Network Load

To test the performance of the database under simulated network load, iperf3 was used. The performance degradation here is probably not quite accurate to a real world situation, since all the traffic is going over the same bridge network and same physical switch, regardless this was the highest jump in latency for any simulated event.





Summary

From the performance data, it's obvious that HBASE is not a contender for this use case, as the insert speed is quite slow during operations (when reads are occurring). Interestingly the HBASE bulk insert is marginally faster compared to Cassandra, but an order of magnitude slower when compared to redis.

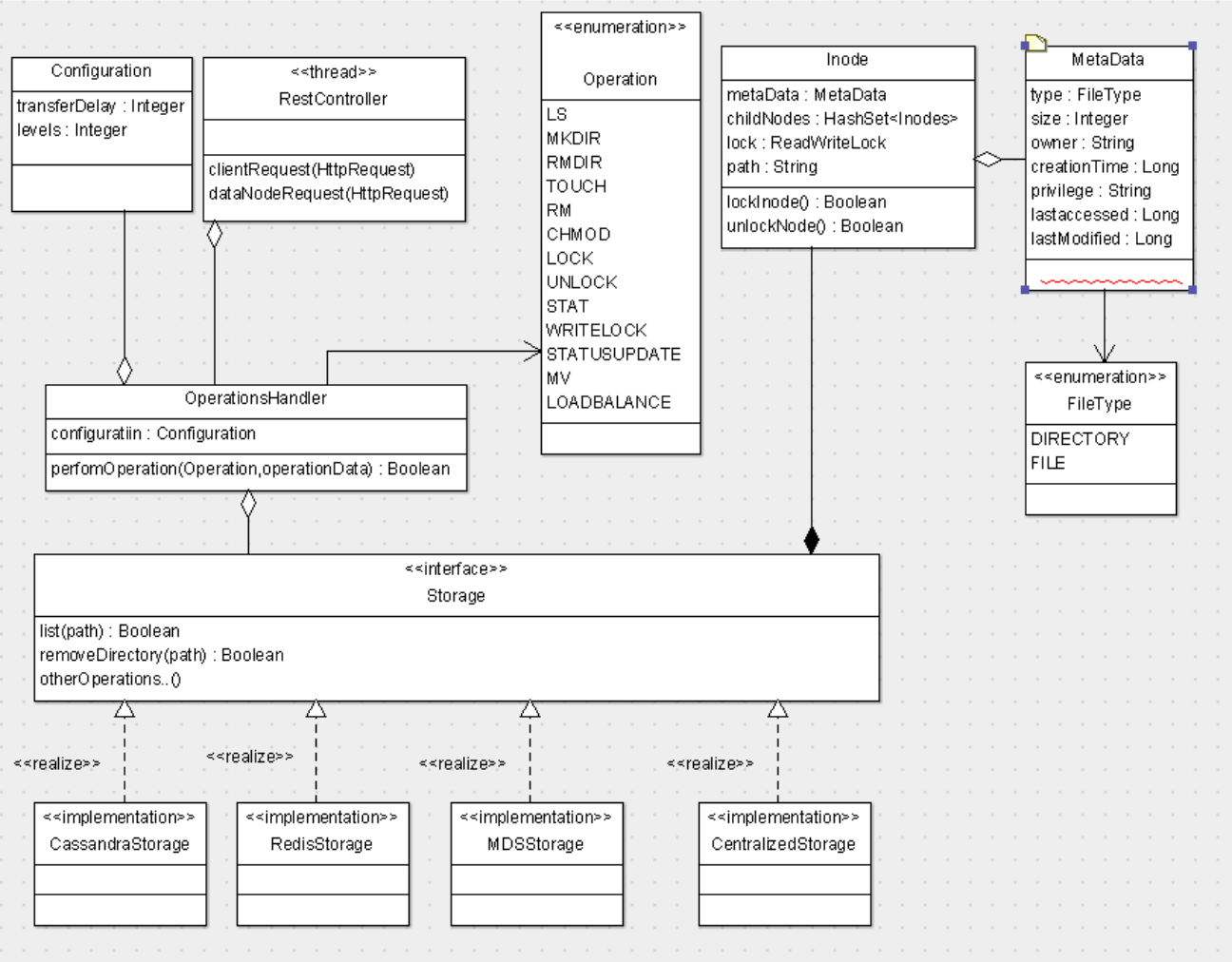
Redis does have good latency when bulk inserting (the fastest of all the systems we tested), but does not have as good of insert performance when writes are also occurring. The read latency is about an order of magnitude higher than that of cassandra.

Appendix Raw YCSB Data

	Bulk INSERT latency	READ latency	UPDATE latency	INSERT During Load Test	Read Modify Write	Throughput (ops/sec)
workloada (3 RegionServers)	12071	3981	8239	-		264.5502646
workloada (2 RegionServers)	10071	7019	6947	-		239.6931927
workloadb (3 RegionServers)	10975	4839	5751	-		264.7603918
workloadb (2 RegionServers)	8839	7927	16447	-		282.2466836
workloadc (3 RegionServers)	8359	5099	-	-		336.3605785
workloadc (2 RegionServers)	9031	9031	-	-		327.2251309
workloadd (3 RegionServers)	7503	5327	-	5011		303.030303
workloadd (2 RegionServers)	10903	4207	-	9679		322.4766204
workloade (3 RegionServers)	6835	-	-	11607		89.92805755
workloade (2 RegionServers)	6783	-	-	12895		80.93889114
workloadf (3 RegionServers)	9247	4055	4647	-	8399	205.7189879
workloadf (2 RegionServers)	9519	7983	19023	-	20143	231.9109462
Cassandra-workloadA-2nodes	11791	26	62	-	-	9259
Cassandra-workloadA-3nodes	11575	30	60	-	-	8771
Cassandra-workloadB-2nodes	12407	29	59	-	-	10869
Cassandra-workloadB-3nodes	13207	30	99	-	-	9345
Cassandra-workloadC-2nodes	11015	29	-	-	-	12195
Cassandra-workloadC-3nodes	12735	29	-	-	-	11235
Cassandra-workloadD-2nodes	11319	28	-	360	-	9708
Cassandra-workloadD-3nodes	13023	29	-	995	-	9433
Cassandra-workloadE-2nodes	12815	-	-	377	-	8475
Cassandra-workloadE-3nodes	13015	-	-	1001	-	8849
Cassandra-workloadF-2nodes	13607	26	60	-	116	7092
Cassandra-workloadF-3nodes	12743	27	58	-	116	7692
Redis - workloadA	1531	714	739			1937.9844
Redis - workloadB	1531	545	1791	1161		2145.922
Redis - workloadC	1131	498				2247.191
Redis - workloadD	1171	606		4679		1930.501
Redis - workloadE	1197			2667		281.214
Redis - workloadF	1196	567	410		1402	1779.359

Metadata Management Project Implementation

Class Diagram



USAGE

We are using Http post requests to make requests to the metadata server. Any restclient can be used to communicate with our server. Any errors or output is returned in the body of the response

For example

Ls Post to localhost:8080 ls /a/b

Mkdir Post to localhost:8080 mkdir /a/b

Directory Structure

The Directory structure was implemented using a tree data structure. Each node in the tree represents a directory. A child of the node represents the subdirectory of that

directory. For ease of finding the parent directory the nodes also have a pointer to its parent. We may have two types of nodes in the tree. Virtual nodes represent those directories that are not located on the same server for example in the case of ceph like server whereas physical nodes will be on the same server. We foresee that implementation of the virtual nodes may include functionality such as locking across servers.

Implementation of the centralized metadata server

We have successfully implemented mkdir and ls with locking. Other operations will have similar flow.

Locking

For operations such as “ls”, the locking starts from the root the directory to the target directory. All locks in this case would be read locks. This would help prevent any operations that would delete any ancestors of the target directory.

For operations such as “mkdir”, all ancestors of the target directory(that is the parent of the new directory) will have a read lock whereas the target directory will have a write lock. This would prevent any deletion of the ancestors just like in the case of “ls” and the write lock on the target directory would prevent any read operations such as “ls” from getting the wrong list of children.

For operations such as “rmdir”, the flow will exactly be the same except the target directory(that is the parent of the directory to be deleted) and the directory to be deleted will have write locks. The deleted directory is write locked in order to wait for any operations to finish on it and the target directory is write locked to remove the directory to be deleted from its list of children.