Benchmarking Replication and Consistency Strategies in Cloud Serving Databases: HBase and Cassandra

Huajin Wang[1,2], Jianhui Li[1](✉), Haiming Zhang[1], and Yuanchun Zhou[1]

[1] Computer Network Information Center, Chinese Academy of Sciences, Beijing, China
{wanghj,lijh,hai,zyc}@cnic.cn
[2] University of the Chinese Academy of Sciences, Beijing, China

Abstract. Databases serving OLTP operations generated by cloud applications have been widely researched and deployed nowadays. Such cloud serving databases like BigTable, HBase, Cassandra, Azure and many others are designed to handle a large number of concurrent requests performed on the cloud end. Such systems can elastically scale out to thousands of commodity hardware by using a shared nothing distributed architecture. This implies a strong need of data replication to guarantee service availability and data access performance. Data replication can improve system availability by redirecting operations against failed data blocks to their replicas and improve performance by rebalancing load across multiple replicas. However, according to the PA-CELC model, as soon as a distributed database replicates data, another tradeoff between consistency and latency arises. This tradeoff motivates us to figure out how the latency changes when we adjust the replication factor and the consistency level. The replication factor determines how many replicas a data block should maintain, and the consistency level specifies how to deal with read and write requests performed on replicas. We use YCSB to conduct several benchmarking efforts to do this job. We report benchmark results for two widely used systems: HBase and Cassandra.

Keywords: Database · Replication · Consistency · Benchmark · Hbase · Cassandra · YCSB

1 Introduction

In recent years, it has become a trend to adopt cloud computing in the IT industry. This trend is driven by the rapid development of internet-based services such as social network, online shopping and web search engines. These cloud based systems need to deal with terabytes and even larger amounts of data, as well as keep the cloud service high reliable and available for millions of users. Such scenarios require cloud serving databases to be able to handle huge number of concurrent transaction timely (availability) and to increase their computing capacity during running time (scalability and elasticity). It is difficult for traditional databases to do such jobs, which motivates the development of new cloud serving databases to satisfy the above requirements. BigTable [1], developed by Google to support cloud data serving in the context of Big Data era, is the first of such databases, and it has inspired a variety of similar systems such as HBase [2] and Cassandra [3]. These systems are usually based on key-value stores, adopt a shared nothing distributed framework and therefore can scale out to thousands of commodity hardware at running time.

However, these new type of databases encounter the CAP [4] tradeoff: According to the CAP theorem, it is impossible for a distributed computer system to simultaneously provide all the three following guarantees: availability, consistency and partition tolerance. In practice, cloud serving databases are usually deployed on clusters consisting of thousands of commodity machines. In such clusters, network partitions are unavoidable: Failures of commodity hardware are very common and the update operations can barely be simultaneously performed on different nodes too, which mimics partitions. For such reasons, cloud serving databases must make tradeoff between availability and consistency. In order to keep high availability, most cloud serving databases choose a weaker consistency mechanism than the ACID transactions in traditional databases.

Also due to the partition problem, cloud serving databases usually use data replication to guarantee service availability and data access performance. Data replication can improve the system availability by redirecting operations against failed data blocks to their replicas, and improve data access performance by rebalancing load across multiple replicas. However, according to the PACELC [5] model, as soon as a distributed database replicates data, another tradeoff between consistency and latency arises. This tradeoff motivates us to figure out how the latency changes when we adjust the replication factor and the consistency level:

- *Replication factor:* The replication factor determine how many replicas a data block should maintain in a specific scenario? Replicas can be used for failover and to spread the load to them, which may imply that the higher replication factor, the better load balancing and the shorter request latency. However, such an assumption is questionable before we do some performance comparisons by changing the replication factor of the same cluster. Besides, the storage capacity are not unlimited. When we use a replication factor of n, the actual space occupied by the database is n times the size of the records it originally intends to store. So, we should carefully make decisions on the replication factor.
- *Consistency level:* How to process read and write requests performed on replicas? For example, writes are synchronously written to all of the replicas in HBase to keep all replicas up to date, which may lead to high write latency. Asynchronously writing brings lower latency, however, in which replicas may be outdate.

We need to benchmark these tradeoffs to give answers to the above questions. In this work, we elaborate the benchmark methodology and show some results of this benchmarking effort. We report the performance results for two databases: HBase and Cassandra. We focus on the changes in request latency and throughput when the strategy of replication and consistency changes.

The paper is organized as follows. Section 2 provides a brief introduction to the strategy designs on replication and consistency in HBase and Cassandra. Section 3 discusses the methodology behind this benchmarking effort. Details about the benchmark and the testbed are also included in this section. Benchmark results are illustrated with corresponding analyses in Sect. 4. Section 5 reviews several related work on benchmarking efforts in this filed. We look to the future work in Sect. 6 and make conclusions in Sect. 7.

2 Database Design

This section investigates how each of HBase and Cassandra has been designed on data replication and how they try to keep consistency between replicas.

HBase provides strong consistency for both read and write. The clients cannot read inconsistent records until the inconsistency is fixed [6]. HBase doesn't write updates to disk instantly, instead, it saves updates in a write-ahead-log (WAL) stored in hard drive and then does in-memory data replication across different nodes, which increases the write throughput. In-memory files are flushed into HDFS when the size of them reaches the upper limit. HBase uses HDFS to configure the replication factor and save replicas.[1] Apparently, HBase prefer consistency to availability when it makes the CAP tradeoff.

Unlike HBase, Cassandra supports a tunable consistency level. There are three well known consistency levels in Cassandra: level ONE, level ALL and level QUORUM. Literally, the names represent the number of replicas on which the read/write must succeed before response to the client. The consistency level for read and write can be set separately in Cassandra. Reasonable choices on consistency are listed below:

- *ONE*: This level is the default setting in Cassandra. It returns a response from one replica for both read and write. For read, the replicas may not always have the most recent write, which means that you have to tolerate reading stale data. For write, the operation should be successfully performed on at least one replica. This strategy provides high level of availability and low level of consistency. Apparently, Cassandra prefers availability to consistency by default.
- *Write ALL*: This level writes to all replicas and read from one of them. A write will fail if a replica doesn't make a response. This level provides high level of consistency and read availability, but low level of write availability.
- *QUORUM*: For write, the operation must be successfully performed on more than half of replicas. For read, this level returns the record with the most recent timestamp after more than half of replicas have responded. This strategy provides high level of consistency and strong availability.

Similar to HBase, Cassandra updates are first written to a commit log stored on hard drive and then to an in-memory table called *memtable*. *Memtables* are periodically written to replicas stored on the disk. Cassandra determines the responsible replicas in fixed order when handling requests. The first replica, also called main replica, is always performed, no matter which consistency level is used.

[1] HBase also supports replicating data across datacenter for disaster recovery purpose only.

3 Benchmarking Replication and Consistency

We conduct several benchmarking efforts to give performance results for different replication/consistency strategies. These results are obtained through different workloads under specified system stress level. This work is based on YCSB [7]: Yahoo! Cloud Serving Benchmark. YCSB has become the de facto industry standard benchmark for cloud serving databases since its release. The core workloads in YCSB are sets of read/insert/update/scan operations mixed in different proportions. The operated records are selected use some distributions. Such workloads are apt to test the tradeoff strategies in different scenarios. What's more, YCSB is good at extensibility: Researchers can implement its interfaces to put new databases into this benchmark. To date, NoSQL databases like HBase, Cassandra, PNUTS [8], Voldemort [9] and many others have been benchmarked using YCSB.

3.1 Benchmark Methodology

In this work, all the tests are conducted on the same testbed to obtain comparative performance of databases using different replication and consistency strategies. In order to get credible and reasonable test results, several additional considerations should be carefully taken:

- *The number of test threads:* The number of client test threads of YCSB should be carefully chosen to prevent side effects of latencies caused by clients. If we use a heavy benchmark workload but a small number of test threads, each thread will be burdened with too many requests, as a result of which, the request latency rises for non-database-related reasons. The right number of client test threads can be set by analyzing the average system load of the client.
- *The number of records:* The number of records should be large enough to avoid a *local trap*. The local trap means that most of the operations are handled by only a few cluster nodes, as a result of which, we cannot obtain the overall performance of the cluster. This issue usually happens when there are not enough test data. The small number of records will also cause the *fit-in-memory* problem: The majority of the test data is cached in memory, as a result of which, benchmarking read performance become meaningless. The proper number of records should ensure that the test operations can be performed with disk access on the whole database cluster evenly.
- *The number of operations:* The number of operations should be large enough to generate substantial and stable load across all nodes evenly and make sure that the test can run for a long time to overcome side effects of *cold start* and *memory garbage collect*.

3.2 Benchmark Types

What make a benchmarking effort reasonable, reliable and meaningful? The reasonability and reliability of a benchmarking effort can be guaranteed by including some micro tests, because univariate results from such tests can be used to predict and explain results of comprehensive tests. The meaningfulness of a benchmarking effort can be archived by doing some stress tests, for stress tests can give an overview of the performance of the database system and are more similar to reality. Therefore, we introduce the following benchmarks into this effort:

- *Micro benchmark for replication:* This benchmark uses different replication strategies to compare the throughput and latency of databases. In order to get the most basic aspects of the performance, this benchmark uses workloads consisting of atomic operations. In order to reduce the latency variance introduced by the various size of transaction data, the test data consists of records of tiny size.
- *Stress benchmark for replication:* This benchmark uses different replication strategies to compares the throughput and latency of databases running at full speed. This benchmark can present an overview of the system performance and is more similar to reality for its scenario-based workloads.
- *Stress benchmark for consistency:* This benchmark uses different consistency levels to check out the changes in database throughputs. The workload of this benchmark is the same as that of the stress benchmark for replication.

3.3 Benchmark Workloads

As we mentioned above, the workloads in micro tests and stress tests are different. In micro tests, the test data consists of 1 billion records of 1 byte, and the workloads are basic insert/read/update/scan operations. In stress tests, the test data consists of 100 million records of 1000 bytes, the target records are chosen using some distributions, and the workloads are borrowed from YCSB with adjustments on the insert/read/update/scan ratio to simulate the real life workloads of different scenarios:

- *Read mostly*: This workload consists of reads mixed with a small portion of writes. This workload represents real life applications like reading on social website mixed with remarking actions.
- *Read latest:* This workload reads the newest updated records. The typical usage scenario is reading feeds on Twitter, Google plus etc.
- *Read & update:* This workload concerns reads and updates equally. The typical real life representative is the online shopping cart: People review the cart and change their choices.
- *Read-modify-write:* This workload reads some records to modify, then write them back. A typical real life usage is that people often review and change their profiles on websites.
- *Scan short range:* This workload retrieves records satisfying certain conditions. A typical application scenario is that people view news retrieved by recommended trends or topics on the social media websites.

We conclude the benchmark workloads used in the stress benchmark in Table 1.

3.4 Testbed

We use 16 server-class machines in the same rack as the testbed, which can reduce inferences from the partition problem (*a.k.a.* the P in the CAP theorem). Each machine

Table 1. Workloads of the stress benchmarks for replication and consistency

Workload	Typical usage	Operations	Records distribution
Read mostly	Online tagging	Read/update ratio: 95/5	*Zipfian*
Read latest	Feeds reading	Read/insert ratio: 80/20	*Latest*
Read & update	Online shopping cart	Read/update ratio: 50/50	*Zipfian*
Read-modify-write	User profile	Read/read-modify-write ratio: 50/50	*Zipfian*
Scan short ranges	Topic retrieving	Scan/insert ratio: 95/5	*Zipfian*

owns two Xeon L5640 64 bit processers (each processer owns 6 cores and each core owns 2 threads), 32 GB of RAM, one hard drive and gigabit ethernet connection. In HBase tests, we configure 15 nodes as HRegion servers, leaving the last node serving as both HMaster and the YCSB client. In Cassandra tests, we also use 15 nodes to do the server job, leaving one machine to emit the test requests. We run Cassandra 2.0.8 and HBase 0.96.1.1 with recommended configurations in these tests. In order to make YCSB compatible with Cassandra 2.0.8 and HBase 0.96.1.1, we compile YCSB from the latest source code with modifications on library dependencies.

4 Experimental Results

The experimental results are shown and analyzed separately for different benchmarks.

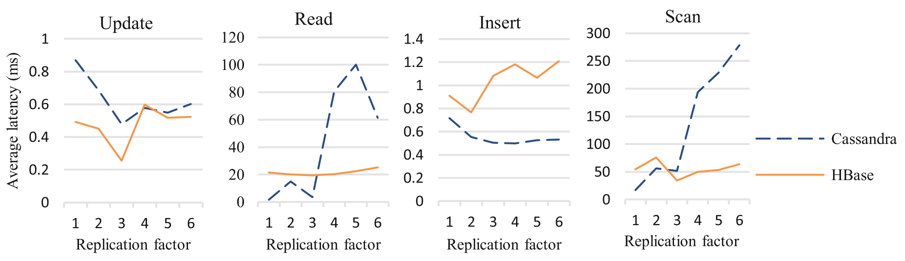

Fig. 1. Results of the micro benchmark for replication in HBase and Cassandra

4.1 Micro Benchmark for Replication

In this benchmark, we keep the load of the testbed in unsaturated state by limiting the number of concurrence requests, and conduct six rounds of testing. In each round, the replication factor is increased by one, and the update/read/insert/scan test is run one after another. Our expectations on changes in latency when the replication factor increases are:

- The read/scan latency changes slightly in both HBase and Cassandra tests. This is because no matter how many replicas exist, either HBase or Cassandra can only read from the main replica when using the default consistency strategy.
- The insert/update latency becomes higher in HBase tests. This is because HBase need to guarantee writing to all replicas successfully, and the write overhead become heavier when the replication factor increases.
- The insert/update latency changes slightly in Cassandra tests. This is because no matter how many replicas exist, Cassandra only need to guarantee writing to one replica successfully when using the default consistency strategy.

The experimental results are illustrated in Fig. 1, from which we can learn:

- Both the curve of read/scan latency in HBase tests and the curve of insert/update latency in Cassandra tests fluctuate smoothly, which is in line with the expectations.
- There is no significant change in the insert/update latency in HBase tests, which do not meet the expectation. The possible reason for such results is that when HBase writes, it do in-memory data replication instead of writing to replicas on hard drive instantly, which significantly reduce the write overhead.
- The read/scan latency in Cassandra tests increases rapidly as the replication factor is higher than 3, which is beyond the expectation. Such a result may be due to the extra burden introduced by the *read repair* process which issues writes to the out-of-date replicas in background to ensure that frequently-read data remains

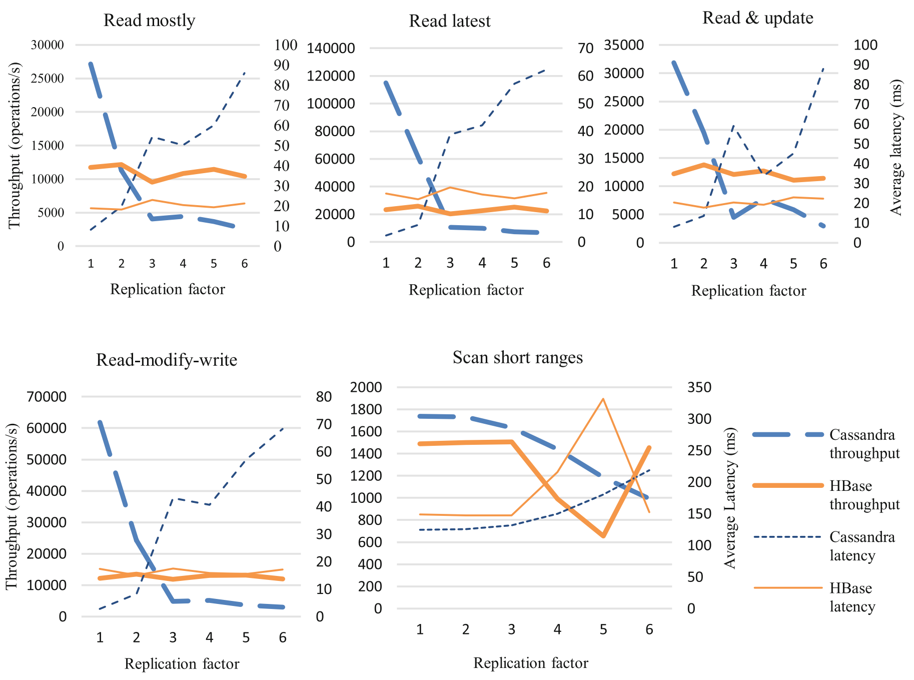

Fig. 2. Results of the stress benchmark for replication in HBase and Cassandra

consistent. Cassandra enables this feature by default [10], and the read-after-write test pipeline can trigger the *read repair* processes. When the replication factor increases, the burden of *read repair* continue to increase, which results in higher read latencies.

4.2 Stress Benchmark for Replication

In this benchmark, we use a constant number of test threads and a variety of target throughputs to detect the peak runtime throughput and the corresponding latency of databases. We conduct six rounds of testing. In each round, the replication factor is increased by one, and the read latest/scan short ranges/read mostly/read-modify-write/read & update test is run one after another. Our expectations on changes in latency and throughput when the replication factor increases are:

- The runtime throughput is inverted-related with the latency in all tests. This is because when we use stress workloads to exam the upper limit of processing capacity, the latency depends on the capacity of the database cluster. The YCSB client will not emits a new request until it receives a response for the prior request — higher latencies will slow down the request emitting rate and then lead to lower runtime throughputs.

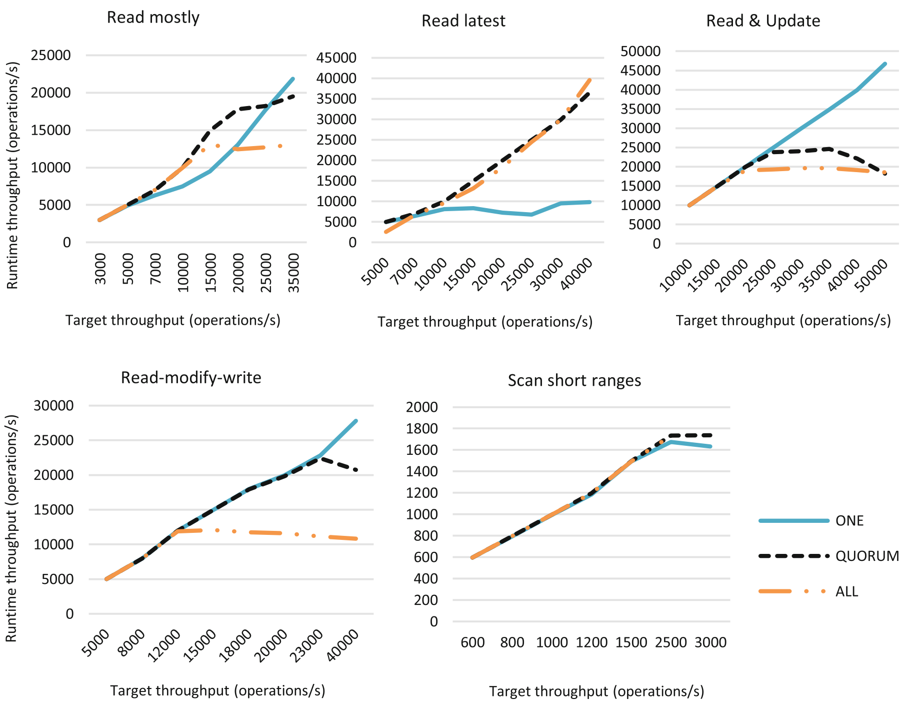

Fig. 3. Results of the stress benchmark for consistency in Cassandra

- The latency in all HBase tests changes insignificantly, which can be derived from the results of the HBase micro tests: All the five stress workloads simply consist of basic read/write operations. If the read/write latency in micro tests changes slightly, there is no reason for dramatic changes in latency in the stress tests.
- The latency in all Cassandra tests increases significantly, which can be derived from the changes of the read/scan latency in the Cassandra micro tests, because in all the stress workloads, at least 50 percent of operations are reads.

The throughput/latency versus replication factor results for HBase and Cassandra for the five stress workloads are illustrated in Fig. 2, from which we can learn that all the test results are in line with the expectations.

4.3 Stress Benchmark for Consistency

In this benchmark, we use a replication factor of 3, a constant number of test threads and a variety of target throughputs to detect the runtime throughput of Cassandra. There is no convenient method to adjust the default consistency strategy of HBase, hence we can only use Cassandra to do this job. Cassandra allows specifying the consistency level in request time, which makes the tests feasible. We conduct three rounds of testing, the consistency levels of which are respectively ONE, write ALL and QUORUM, and the read latest/scan short ranges/read mostly/read-modify-write/read & update test is run one after another in each round. Our expectations on changes in runtime throughput when the consistency level changes are:

- In the read latest test, level ONE performs worst, and level QUORUM and level ALL perform closely better. This is because the *read repair* process is frequently triggered in the read latest test. In this test, reads are intensively performed on new written records. Each write produces two inconsistent replicas in level ONE and almost zero inconsistent replica in level QUORUM/ALL. Apparently, more inconsistent replicas lead to heavier overhead of the *read repair* process.
- In the scan short ranges test, all the three levels perform closely. This is because overhead of the *read repair* process dramatically declined in this test, for we run this test after the read latest test which has repaired the majority of inconsistency. Moreover, there are only reads in this test, and in the perspective of YCSB client, reading from which of the replicas in the same rack is indifferently, as a result of which, the read latency can hardly be affected by the number of replicas too.
- In other tests, level ONE, level QUORUM and level ALL perform best, almost worst and worst respectively. This is because the write overhead becomes heavier when using a higher consistency level.

Figure 3 presents the runtime throughput versus target throughput with different consistency levels. We observe that all the expectations are confirmed by the experiment results with narrow biases. What's more, the bigger write proportion, the more obvious performance difference in these tests.

5 Related Work

It is complicated to conduct benchmarking efforts in the cloud serving database field: There are many tradeoffs need to think about when we evaluate a specific aspect of cloud serving databases. However, the YCSB framework brought a relatively easy way to do *apple-to-apple* comparisons between cloud serving databases, and has inspired some other benchmark tools in this field. The BigDataBench [11, 12], for example, has adopted YCSB as one of its components with extensions like the new metric on energy consumption.

With the help of YCSB, several meaningful efforts have been done in this filed. Pokluda et al. [13] benchmarked the availability of the failover characteristics of two representative systems: Cassandra and Voldemort. They used YCSB to monitor the throughput and latency of individual requests during a node failed and came back online, and found that transaction latency increased slightly while the node was down and that recovery. Bermbach et al. [14] evaluated the effects of geo-distributed replicas on the consistency of two cloud serving database: Cassandra and MongoDB. They used YCSB to generate workloads and Amazon EC2 to deploy these two databases. Replicas were distributed in same/different regions (Western Europe, northern California and Singapore) to compare the degree of inconsistency.

However, there are few efforts like this work to benchmark the replication factor.

6 Future Work

In this work, we specify the stress level using different target throughputs, which is inaccurate and lack of versatility for the comparison between different clusters. Another way to specify the stress level is using the service level agreement, SLA. An SLA is commonly specified like this: At least p percentage of requests get response within l latency during a period of time t. Using the SLA, We can keep user experiences at same level to compare throughputs of different systems. However, it is hard to specify an SLA using YCSB. We need to extend it.

There is *cold start* problem when we run benchmark workloads using YCSB, which leads to inaccurate results in latency tests. We have to run the tests for a long time, and repeat the tests several times to overcome this flaw. Consequently, the whole running time become long, and the benchmarking effort become inefficient in energy consumption. We need to optimize the method of test and result measurement used in YCSB.

Furthermore, this work has shown that a single rack of nodes cannot form a convincing testbed for more complicated tests such as geo-read latency test, partition test and availability test. We need to build a geo-distributed testbed to conduct such tests.

7 Conclusion

In this paper, we present our benchmarking effort on the replication and consistency strategies used in two cloud serving databases: HBase and Cassandra. This work is motivated by the tradeoff between latency, consistency and data replication. This benchmarking effort consists of three parts: Firstly, we use the atomic read/insert/update/scan operations to do micro tests to fetch the basic performance aspects of the target database. This part makes the foundation of further comprehensive benchmarks. Secondly, we change the replication factor to compare the performance of cloud serving databases. This part sketches an overview of the whole system performance. Finally, we use different consistency levels to compare the runtime throughputs. This part figures out the proper consistency strategy for some scenarios. We have observed some interesting benchmark results from a single-rack testbed:

- More replicas can hardly accelerate the read/write operation (in HBase), and even harm the read performance (In Cassandra using low level of consistency).
- The write latency dramatically increases when using higher level of consistency in Cassandra.
- High consistency level is not suitable for read latest and write heavy scenarios in Cassandra.

These results can give answers to the questions we mentioned in the introduction part within a SINGLE-RACK scope of validity.

In this effort, we also find the testbed is not suitable for read latency tests, and the benchmark tool has some efficiency affecting flaws. We will optimize the testbed and benchmark tools to conduct more rational and stable tests in the future.

References

1. Chang, F., et al.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. (TOCS) **26**(2), 4 (2008)
2. Apache HBase. http://hbase.apache.org/
3. Apache Cassandra. http://cassandra.apache.org/
4. Brewer, E.: CAP twelve years later: How the “rules” have changed. Computer **45**(2), 23–29 (2012)
5. Abadi, D.J.: Consistency tradeoffs in modern distributed database system design. IEEE Comput. Mag. **45**(2), 37 (2012)
6. HBase read high-availability using timeline-consistent region replicas. http://issues.apache.org/jira/browse/HBASE-10070
7. Cooper, B.F., et al.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing. ACM (2010)
8. Cooper, B.F., et al.: PNUTS: Yahoo!’s hosted data serving platform. Proc. VLDB Endow. **1**(2), 1277–1288 (2008)
9. Project Voldemort: A distributed database. http://www.project-voldemort.com/voldemort/
10. About Cassandra’s Built-in Consistency Repair Features. http://www.datastax.com/docs/1.1/dml/data_consistency#builtin-consistency
11. Gao, W., et al.: Bigdatabench: a big data benchmark suite from web search engines (2013). arXiv preprint arXiv:1307.0320
12. Wang, L., et al.: Bigdatabench: A big data benchmark suite from internet services (2014). arXiv preprint arXiv:1401.1406

13. Pokluda, A., Sun, W.: Benchmarking Failover Characteristics of Large-Scale Data Storage Applications: Cassandra and Voldemort. http://www.alexanderpokluda.ca/coursework/cs848/CS848%20Project%20Report%20-%20Alexander%20Pokluda%20and%20Wei%20Sun.pdf
14. Bermbach, D., Zhao, L., Sakr, S.: Towards comprehensive measurement of consistency guarantees for cloud-hosted data storage services. In: Nambiar, R., Poess, M. (eds.) TPCTC 2013. LNCS, vol. 8391, pp. 32–47. Springer, Heidelberg (2014)