MyCassandra: A Cloud Storage Supporting both Read Heavy and Write Heavy Workloads

Shunsuke Nakamura, Kazuyuki Shudo

Department of Mathematical and Computing Sciences
Tokyo Institute of Technology
Tokyo, Japan
{nakamur6, shudo}@is.titech.ac.jp

Abstract

A cloud storage with persistence shows solid performance only with a read heavy or write heavy workload. There is a trade-off between the read-optimized and write-optimized design of a cloud storage. This is dominated by its storage engine, which is a software component for managing data stored on memory and disk. A storage engine can be pluggable with an adequate software design though today's cloud storages are not always modular. We developed a modular cloud storage called MyCassandra to demonstrate that such a cloud storage can be read-optimized and write-optimized with a modular design. Various storage engines can be introduced into MyCassandra and they determine with what workload the cloud storage can perform well. With MyCassandra we proved that such a modular design enables a cloud storage to adapt to workloads.

We propose a method to build a cloud storage that performs well with both read heavy and write heavy workloads. A heterogeneous cluster is built from MyCassandra nodes with different storage engines, read-optimized, write-optimized, and on-memory (read-and-write-optimized). A query is routed to nodes that efficiently process it while the cluster maintains consistency between data replicas with a quorum protocol. The cluster showed comparable performance with the original Cassandra for write heavy workloads, and it showed considerably better performance for read heavy workloads. With read-only workload, read latency was 90.4% lower than and throughput was 11.00 times as high as Cassandra.

Categories and Subject Descriptors C.2.4 [*Computer Systems Organization*]: Distributed databases; H.3.4 [*Software and Software*]: Performance evaluation

General Terms Design, Experimentation, Performance

Keywords Cloud Storage, Distributed Systems, Performance

1. Introduction

Cloud storages such as NoSQL, key-value store (KVS), document-oriented database, and GraphDB, which meet requirements that are difficult for traditional relational database (RDB) to treat, have attracted a great deal of public attention. There are currently over 100 NoSQL projects that have been actively developed and put into practice, and operated. Compared to RDB, in common, they have advantages in scalability to a large number of servers, while they constrain their data models, support only simple queries, and adopt relaxed consistency models.

Each cloud storage has its own characteristics derived from its design decisions. There are many design choices. First, there are various data models in cloud storages: a key-value map, as in Redis [17], a multi-dimensional map, as in Cassandra [10, 20], documents, as in MongoDB [2], and graphs, as in Neo4j [13]. There are also various distribution architectures: a cloud storage can be centralized, such as a master-slave model or a sharded model, and another can be decentralized with consistent hashing. There are other design trade-offs between performance and persistence, i.e., whether to keep data in memory or store them on disk, place replicas or not, and place them synchronously or asynchronously.

Similarly, there is a trade-off between the write-optimized or read-optimized design of a cloud storage with persistence. Table 1 lists properties of common cloud storages with persistence focused on read vs. write performance. A storage engine means a software component for managing data stored on memory and disk. Each storage engine implements an indexing algorithm suitable for its application. B-Trees are a balanced tree commonly used in traditional databases and filesystems. They process write and read queries in logarithmic time, while LSM-Tree is a newer log-based tree optimized for write queries and processes them in a constant time. Apache Cassandra and Apache HBase [1], which adopt LSM-Tree, are write-optimized. Yahoo Sherpa [4] and sharded MySQL [15], which adopt B-Trees, are read-optimized. Our observation shows that the storage engine determines which workload a cloud storage treats efficiently.

On the other hand, the distribution architecture of a cloud storage is independent of the performance characteristics on read and write queries. Queries in a centralized cloud storage take same access path from a client to one or more storage nodes as that in a decentralized one. There is no difference between write and read queries.

We focused on storage engines and developed MyCassandra, a cloud storage which separates storage engine from a distribution architecture. It works with various storage engines, which significantly affect the performance characteristics of the read and write queries. As a result, we confirmed that selection of the storage engine determines if a cloud storage is to be write or read optimized.

Similar to the traditional cloud storages with persistence, a homogeneous cluster of MyCassandra nodes shows solid performance only with a read heavy or write heavy workload. This paper presents a method to build a cloud storage that performs well

SYSTOR'12, June 4–6, 2012, Haifa, Israel.

Table 1. Properties of cloud storages

	Cassandra, HBase	Sherpa, sharded MySQL
Indexing	LSM-Tree	B-Trees
Write	sequential writes	random writes
Read	random reads, merge	random reads
Performance	write-optimized	read-optimized
Storage engine	Bigtable clone	MySQL

with both types of workloads. A heterogeneous cluster is built from MyCassandra nodes with different storage engines. Experiments showed that this method enables the cluster to perform well with both types of workloads.

This paper is organized as follows. Section 2 introduces Apache Cassandra, on which our implementation is based. Section 3 describes MyCassandra, a modular cloud storage we developed, and presents the method to build a heterogeneous cluster of MyCassandra nodes, which is compatible with both the read and write performance. Section 4 shows experimental results and discusses the results. Section 5 discusses the pros and cons of our proposal, specially about the perspectives other from performance. Section 6 examines modular cloud storages and indexing algorithms as related work. Section 7 shows conclusion and future work.

2. Background

It is difficult for a storage to predict what data will be written and read. Therefore, unless all data are on memory, accesses to the data require a number of random I/Os to disk. A log-structured storage [18], which performs only sequential I/Os on disk when writing, achieves much higher throughput than traditional storages involving random I/Os. For a read query, a log-structured storage has to coalesce different parts of logs. Such multiple reads incur a performance penalty. Thus, there is an inherent trade-off between optimizing for read queries and optimizing for a write query in traditional cloud storages.

We propose and give empirical proof of a method for building a cloud storage that shows good performance with both read heavy and write heavy workloads.

We first design and develop a modular cloud storage called MyCassandra. Next, we organize a heterogeneous cluster using MyCassandra nodes.

This section introduces Apache Cassandra, on which our implementation is based.

2.1 Apache Cassandra

Apache Cassandra [10, 20] is an open source cloud storage developed by Facebook and it is currently a top-level project in Apache Software foundation. Its notable properties are scalability up to hundreds of servers operating in multiple data centers and high availability with no single point of failure by adopting a decentralized structure.

The followings are Cassandra's features related to the method proposed in this paper.

2.1.1 Consistent Hashing

Cassandra adopts consistent hashing to map data to nodes. Figure 1 represents consistent hashing of Cassandra. In particular, consistent hashing assigns identifiers to both nodes and data on its circular identifier space. For example, if the hash value of the record's key *key1* is *11*, node *14*, which is that closest clockwise to *11*, is appointed as the primary node for *key1* and is responsible for the record. Any node can serve clients as a proxy and relays queries to nodes responsible for the record. Thus, a query with *key1* is relayed to node *14*.

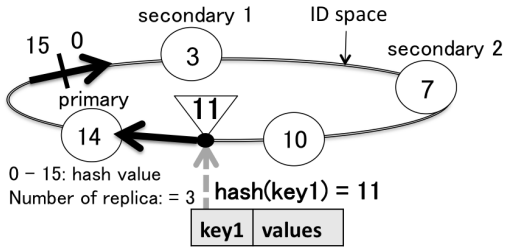

Figure 1. Consistent hashing

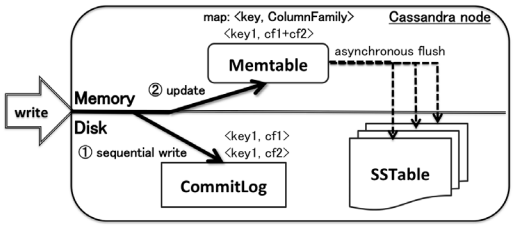

Figure 2. Process flow of a write query in a Cassandra node

An advantage of consistent hashing is fine load balancing by a hash function, and it is able to provide high availability because there are no master that manages responsible range of each node on the identifier space. In most cloud storages including Cassandra, all nodes recognize other nodes' identifiers and IP addresses. Any node in a cluster can take the role of a proxy that responds to queries from a connected client.

2.1.2 Bigtable storage engine

Cassandra's storage engine is a clone of Google Bigtable [3]. It adopts the Log-Structured Merge Tree (LSM-Tree) [14] and is write optimized. The storage engine consists of *CommitLog*, *Memtable*, and *SSTable*. Figure 2 shows how a write query is processed in Cassandra. A node processes write queries in the following procedure:

1. First, the node sequentially appends data to *CommitLog* for persistence.
2. Next, it updates *Memtable*, a map on memory, for quick reading, and acknowledges a client.
3. If data overflow *Memtable* or after data remained for a specified period on *Memtable*, the storage engine asynchronously flushes the data to *SSTable* on disk and deletes them from *CommitLog* and *Memtable*.

This procedure achieves both high performance and consistency by limiting synchronous disk accesses to sequential ones. It also enables a node to respond to concurrent write queries from many clients without locking. On the other hand, these advantages compensate lesser read performance. Cassandra reads changes for a single record from multiple *SSTable*s and merges them to reply to a read query. It requires a number of random I/O to disk. Cassandra always maintains data persistence for any datum by keeping it in the *CommitLog* or a *SSTable* on disk.

2.1.3 Replication

In a cluster of Cassandra nodes, replicas are placed on *N* nodes, a primary node and *N - 1* secondary nodes, which are clockwise nodes from the primary node. There is no distinction between them. For example, Figure 1 shows three replicas on a primary node *14*

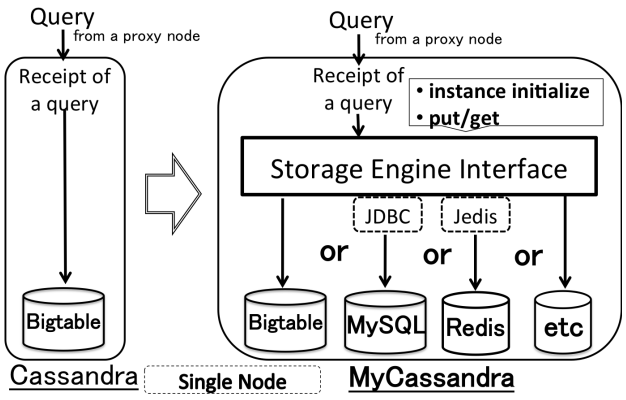

Figure 3. Storage Engine Interface of MyCassandra

and two secondary nodes *3* and *7*. They are responsible for the hash value *11*.

Cassandra can take account of server racks and data centers when choosing secondary nodes. It improves availability in case of network partitions and coincident server failures in the same rack. This way of replication is advantageous in that there are no servers managing meta-data of replicas.

3. System Overview

We propose a method for building a cloud storage that performs well with both read heavy and write heavy workloads in the following steps:

1. **MyCassandra**: a modular cloud storage based on Cassandra, which can be read or write optimized.
2. **A heterogeneous cluster of MyCassandra**: a heterogeneous cloud storage, which consists of MyCassandra nodes with different types of storage engines. It shows good performance with both read and write performance.

3.1 MyCassandra

MyCassandra is a modular cloud storage designed to demonstrate a cloud storage can be read or write optimized with a modular design. Trade-off between the read and write optimized designs of a cloud storage is dominated by storage engine, which is a software component that manages data on memory and disk. A storage engine is an independent component and can be pluggable with a modular software design. Today's cloud storages are not always modular, while MyCassandra is designed to be able to use various storage engines.

Figure 3 shows the components related to reading and writing data. MyCassandra has Storage Engine Interface located between a storage engine and a component for accepting queries. A new storage engine can be added by implementing this interface. The interface prescribes basic functions for data management, i.e., connect, put, and get functions.

This paper targets three different storage engines the authors provide, Bigtable (Cassandra's original one), MySQL, Redis.

Figures 4 and 5 respectively show the write and read latencies of each storage engine (Bigtable, MySQL, and Redis) on a homogeneous cluster of 6 MyCassandra nodes, where read and write are at different rates using the YCSB (Section 4.1). In this experiment, data are not replicated while experiments in Section 4 deploy replicas. The write latency of Bigtable is up to 89.7% lower than that of MySQL in Write Only workload, while the read latency of MySQL is up to 80.3% lower than that of Bigtable in Read Heavy workload.

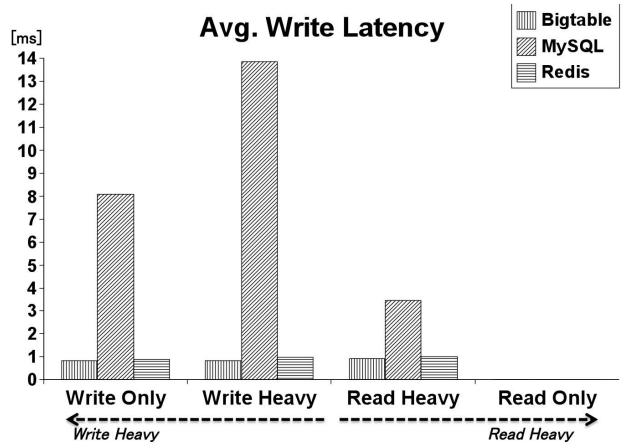

Figure 4. Write latency for each storage engine

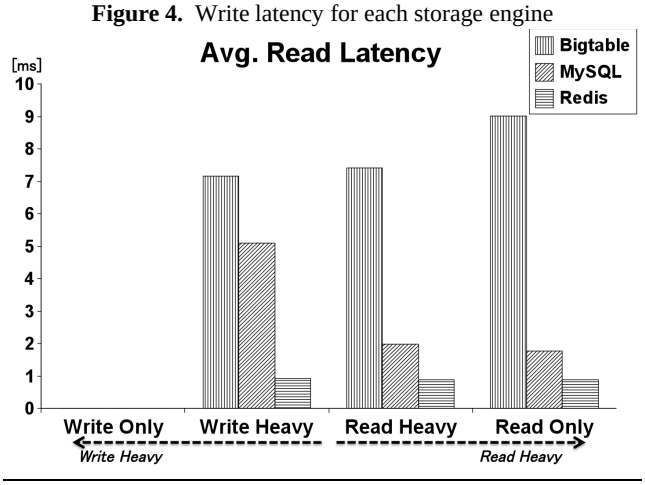

Figure 5. Read latency for each storage engine

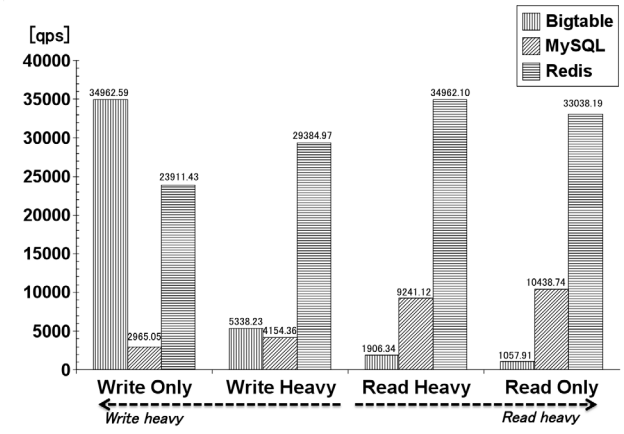

Figure 6. Throughput for each storage engine

Figure 6 also shows throughput for each storage engine. Bigtable achieved up to 11.79 times as high throughput as that of MySQL in Write Only workload, while MySQL achieved up to 9.87 times as high throughput as that of Bigtable in Read Only workload. Redis processes both read and write queries fast by placing data on memory.

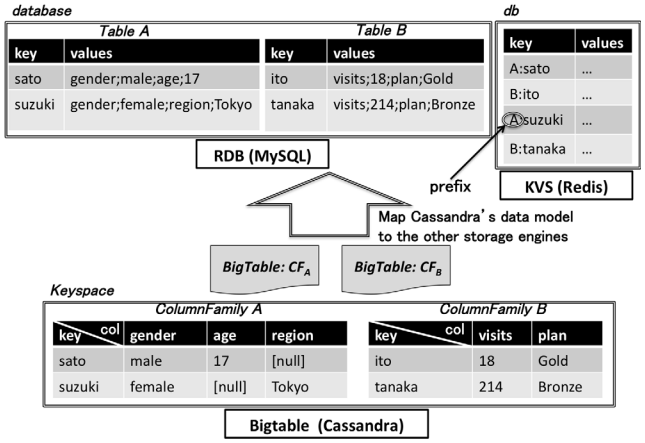

Figure 7. Mapping Cassandra's data model to that of RDB and KVS.

3.1.1 Data Model Mapping

Data for Cassandra cannot be stored directly in RDB or KVS because of their data models are mismatched. Cassandra adopts multi-map model and it is schema-less. Therefore, MyCassandra must convert data format from Cassandra's one to each storage engine's one.

Figure 7 shows Cassandra's data model mapped to MySQL, which is a RDB, and Redis, which is a KVS. Cassandra stores data in a bucket called *ColumnFamily*, where a set of columns is stored. A *ColumnFamily* can include arbitrary types of columns without a pre-specified schema. A *Table* of MySQL is appropriated for the *ColumnFamily*. MyCassandra stores schema-less data of Cassandra into *Tables* of MySQL as follows. MySQL's *Table* requires a predefined schema and MyCassandra defines the schema as having a single key and a single value. MySQL stores multiple columns as a single value by converting them to a byte sequence.

Redis also stores the key-value pair the same as MySQL. In case of Redis, a ColumnFamily name is appended to the head of a key as a prefix because Redis does not support multiple buckets like MySQL's *Tables*. MyCassandra recognizes which ColumnFamily has the datum by the prefix. *Keyspace* of Cassandra is a container for multiple *ColumnFamilies* and it is simply mapped to *database* in MySQL and *db* in Redis.

3.1.2 Storage Engine

This section describes storage engines MyCassandra provides.

Bigtable storage engine is Cassandra's original based on the log-structured model described in Section 2.1.2.

MySQL storage engine is MySQL, a relational DBMS. MyCassandra accesses it via Java Database Connectivity (JDBC). MySQL also has its own various storage engines such as InnoDB and MyISAM. In this work, we choose InnoDB, a default storage engine of MySQL 5.5. In addition, we use stored procedure. It improves performance by eliminating SQL parsing. MyCassandra issues limited SQL statements and stored procedure eliminates all SQL parsing.

Redis [17] is a KVS that stores key-value pairs on memory, similar to memcached [6]. Unlike memcached, Redis supports virtual memory, that extends its capacity beyond equipped memory by swapping out part of data to disk. However, MyCassandra does not use this feature and use Redis just as an on-memory KVS because of its higher performance. To access the Redis storage engine, MyCassandra uses Redis' original APIs, Jedis.

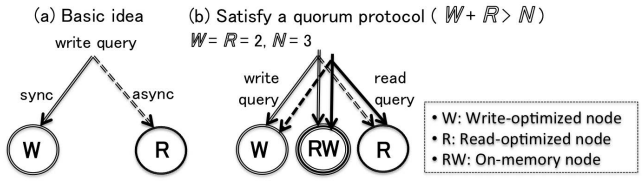

Figure 8. Basic concept of a heterogeneous cluster of MyCassandra

3.2 A heterogeneous cluster of MyCassandra

As the next step, we propose a method to build a cloud storage that supports both read heavy and write heavy workloads. A heterogeneous cluster is built from MyCassandra nodes equipped with different storage engines, write-optimized, read-optimized and on-memory.

The heterogeneous cluster replicates data on different storage engine nodes. A proxy in the cluster receives a query from a client and then routes it to nodes that are responsible to the key specified in the query. While a query is synchronously routed to nodes processing it fast, it is asynchronously routed to nodes processing it slower. In Figure 8-(a), a write query is synchronously routed to a node equipped with a write-optimized storage engine. A synchronous routing returns an acknowledgment to the proxy. An asynchronous routing does not respond to the proxy and the query can be processed later. This approach achieves good performance for both read and write queries in a single cloud storage.

The challenge this approach involves is maintenance of consistency between replicas as much as possible with the original Cassandra. We solve this issue by a quorum protocol. A quorum protocol is a consistency management policy Cassandra supports. This protocol for N replicas is defined as:

$$W + R > N \tag{1}$$

W and R refer to the number of replicas in agreement for writing and reading. They are one or larger. For example, if the number of replicas N is three, $W = R = 2$ satisfies the requirement as $2 + 2 > 3$. That is, there has to be at least one node processing both write and read queries synchronously to maintain a quorum protocol on the heterogeneous cluster. The node must process fast both write and read queries otherwise the cluster performs bad with either write or read queries. Therefore, we give on-memory storage engine nodes the role of the node. Described in section 2.2.1, an on-memory storage engine node can process both write and read queries fast. For example, a write query is synchronously routed to write-optimized and on-memory storage engine nodes and is asynchronously routed to a read-optimized storage engine node, as shown in Figure 8-(b). Also, a read query is synchronously routed to read-optimized and on-memory storage engine nodes and is asynchronously routed to a write-optimized storage engine. Of course, there are trade-offs in using an on-memory storage engine. We discuss these trade-offs in Section 5.

3.2.1 Node placement and load balancing

As stated in the previous section, Cassandra and MyCassandra partition data and route queries by consistent hashing. A proxy places N replicas on a primary node and *N-1* closest clockwise secondary nodes on the circular identifier space. On a heterogeneous cluster, a proxy takes account of storage engine type in replica placement. Specifically, a proxy choose *N-1* secondary nodes as N nodes satisfy the constraints on W and R in the formula (1). The number of write-optimized nodes is W or larger and the number of read-optimized nodes is R or larger. The proxy choose such *N-1* nodes as closest clockwise from the primary node as possible.

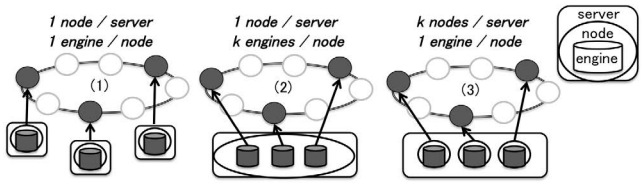

Figure 9. Server, node and storage placement on circular identifier space

There are three approaches at least in node placement on servers, and we investigated the trade-offs between them. Figure 9 shows the three node placement approaches.

We concluded the approach (3) is adequate to our implementation because of fault-tolerance and load balancing, in which a server has at least three nodes, write-optimized, read-optimized and on-memory node. In this approach, a proxy chooses secondary nodes as a server does not have multiple replicas of the same data. If not, replicas cannot improve fault-tolerance and availability. Specifically, when choosing secondary nodes, a proxy skips over a node located on a server having a node that has been chosen as a secondary node.

Approach (1) is naive, in which a server has a single node and each node has a single storage engine. This approach has a problem of load imbalance. For example, if a workload is read heavy, the load is concentrating to read-optimized and on-memory servers. Another problem is to be unable to fully utilize the resources of a server because on-memory nodes do not require any disk space.

Approach (2) is space-efficient, in which a server has a single node and each node has different multiple storage engines. This approach is like a virtual node [7] because a node has multiple identifiers on the circular identifier space. However, we doubt that it is difficult for the approach to add or drop storage engines while a cluster is working and has no advantages in terms of fault tolerance. Approach (3) enables storage engine management by starting up and stopping nodes, that requires no additional mechanism.

3.2.2 Membership management

Any node in a heterogeneous cluster has to recognize the storage engine type of all other nodes, because any node serves a client as a proxy and routes a query to a appropriate nodes according to the query type and the storage engine type.

Therefore, we extend Cassandra's gossip protocol, which is used for membership management. The gossip protocol is responsible for ensuring that all of members in a cluster are aware of the state of the other nodes (i.e. live, dead, join). In our implementation, a storage engine type is attached to a gossip message each node exchanges. By that, all nodes recognize others' storage engine types.

3.2.3 Read and write queries

Figure 10 shows the process flow of write and read queries. To process fast both write and read queries, a proxy routes synchronously a write query to write-optimized nodes and a read query to read-optimized nodes. Also, a heterogeneous cluster uses a quorum protocol to ensure data consistency between replicas. In this paper, N is the number of replicas, W is the number of agreements for writing, and R is the number of agreements for reading.

This subsection describes the process flow of queries.

A write query is processed as follows:

1. A proxy node receives a write query from a client. The proxy routes the query to nodes responsible for the record specified by the query.

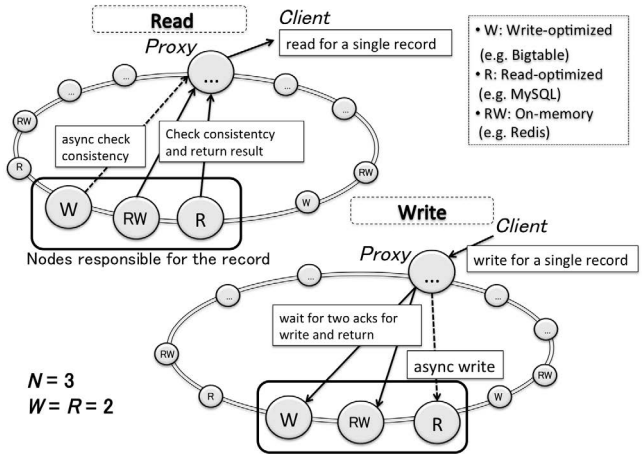

Figure 10. Process flow of read and write queries (in case N = 3)

2. The proxy waits W acknowledgments. Write-optimized and on-memory nodes usually reply fast and they return W or more acknowledgments.

3-a. If writing to them succeeds and the proxy receives W acknowledgments, it returns a success message for the write query to the client.

3-b. If a write-optimized or on-memory node fails in writing, the proxy waits for W acknowledgments including that from read-optimized nodes and returns a success message.

4. After returning, the proxy asynchronously waits for acknowledgments from the remaining N-W nodes.

Write latency is limited to the largest latency in those of the write-optimized and on-memory nodes as long as they succeed. The read-optimized nodes do not affect write performance because they are not counted in W nodes.

A read query is processed as follows:

1. A proxy node receives a read query from a client. The proxy sends a request for the specified record to a read-optimized or on-memory node, and sends a request for digest (hash) of the specified record to other replicas.

2. The proxy waits for R replies including the specified record. Read-optimized and on-memory nodes usually reply fast and they return R or more replies.

3-a. If succeeded in reading and the record and *R-1* digests are consistent, the proxy returns the record to the client.

3-b. If failed to read or the record and digests are not consistent, the proxy tries to read and collect digests until they satisfy the quorum R.

4. The proxy waits for digests from the remaining N-R nodes after replying to the client. If there is an old record or inconsistency among N replicas, the proxy asynchronously updates the old record to the newest version. A Cassandra's feature *ReadRepair* does it.

Read latency is limited to the largest latency in those of the read-optimized and on-memory nodes as long as they succeed. The write-optimized nodes do not affect read performance because they are not counted in R nodes.

The most common cause of the read-time inconsistency is due to the process flow of write queries. On-memory nodes tend to have newer data than read-optimized nodes because an update is asyn-

chronously reflected to read-optimized nodes. It is possible for a read-optimized node not to reflect the latest update. When a proxy notices such an inconsistency between read-optimized nodes and on-memory nodes, it waits for replies from write-optimized nodes, satisfies the quorum and returns consistent record to a client. Even in such a case, write-optimized nodes and on-memory nodes are consistent due to an agreement taken when writing. After that, the proxy resolves the inconsistency by the *ReadRepair*, a Cassandra's feature to resolve such an inconsistency in the background. Following read accesses to the data succeed due to the background repair and take the 3-a step.

By these write and read protocols, write and read latencies are limited to those of the nodes that process the query fast while keeping data consistency. Even in the worst case, the latency is the same as the original Cassandra.

4. Performance Evaluation

This section demonstrates that a heterogeneous cluster of MyCassandra nodes shows good performance with both read heavy and write heavy workloads. We evaluate its performance with the following storage engines:

- Write-optimized: Bigtable (Cassandra 0.7.5)
- Read-optimized: MySQL (6.0.10)
- On-memory: Redis (2.2.8)

4.1 Benchmark

We use Yahoo! Cloud Serving Benchmark (YCSB) [5], that is a cloud benchmark framework developed by Yahoo! Research for evaluation of cloud storages. YCSB provides core workloads close to real-world applications.

In YCSB, an user can specify the ratio of the number of read queries to write queries. YCSB executes these queries on the target cloud storage and measures throughput of the entire workload and the time required for each query, i.e., latency.

Benchmarking process consists of the following phases:

1. Load phase: load data.
2. Warm up phase: warm up the target cloud storage with the same workload as the following measurement.
3. Transaction phase: measure response times.

Table 2 lists the four workloads used in this measurement.

Write Only and Write Heavy are workloads with a high write ratio, while Read Only and Read Heavy are workloads with a high read ratio. This table also lists examples of actual applications with the write and read ratios of each workload.

Accessed data are chosen along Zipfian distribution. The Zipfian distribution is a probability distribution, which means that access frequency of each datum is determined by its popularity, not by freshness.

Table 3 lists experimental parameters, and Table 4 lists server configurations. As described in the previous section, the replication number N is three, and we prepare six servers and boot three nodes on each server, i.e., $3 \times 6 = 18$ nodes for a heterogeneous cluster

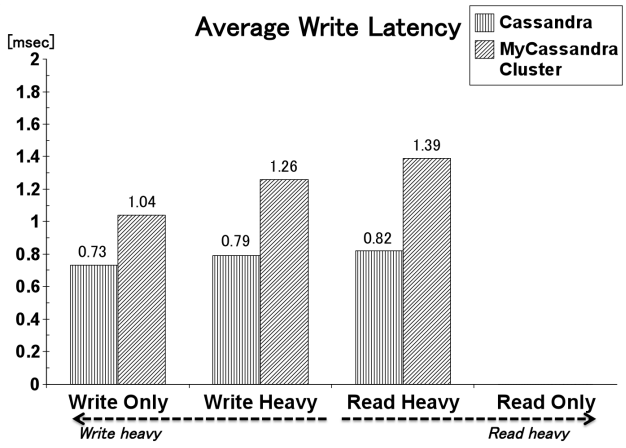

Figure 11. Write latencies for each workload

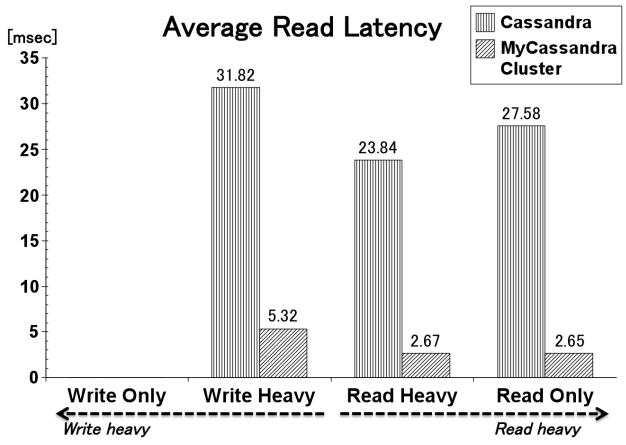

Figure 12. Read latencies for each workload

of MyCassandra, while we boot one Cassandra node on each six physical servers, i.e., 6 nodes. For the number of replicas $N = 3$, the number of replicas in agreement W and R are set both 2 to satisfy a quorum protocol, i.e., $2 + 2 > 3$. We allocated 6 GB of memory to each server as Java Virtual Machine heap for Cassandra, MySQL buffer pool and Redis. Three nodes on a single server share the memory and use 2 GB each. Each server loads 6 millions entries (12 millions $\times$ 3 replicas / 6 servers). Each server seems to have 6 GB of data (1 KB value $\times$ 6 million), but it has more because a data storage requires an index and meta data for each record. For example, a Cassandra node takes 8.8 GB for the loaded data. In this data capacity, each server allocates 6 GB memory resources to data storage (6 GB). MyCassandra assigns three nodes each to 2 GB on a server.

Table 2. YCSB workloads

Workload	Application example	Read	Update
Write Only	*Log*	0%	100%
Write Heavy	*Session*	50%	50%
Read Heavy	*Photo Tagging*	95%	5%
Read Only	*Cache*	100%	0%

Table 3. Experimental parameters

Num. of server machines	6
Num. of client machines	1
Num. of loaded records	12 million
Quorum	(N, W, R) = (3, 2, 2)
Size of a key	up to 10 byte
Size of a value	100 byte $\times$ 10 columns = 1,000 byte

Table 4. Server configuration	
OS	Linux 2.6.35.6 (x86_64)
CPU	2.40 GHz Xeon E5620 × 2
Memory	8 GB RAM
Disk	128 GB SSD × 2
	Crucial Real SSD C300 128 GB
Java Virtual Machine	Java SE 6 Update 21

4.2 Experimental results and discussion

Figures 11 and 12 show average write and read latencies. Throughput was fixed to 5,000 queries per second, that is moderate in YCSB experiments [5]. Write latency of the heterogeneous cluster was higher than that of Cassandra, but the differences were small as they are between 0.31 ms and 0.57 ms. In contrast to them, read latency of MyCassandra was much lower than that of Cassandra as 83.3% lower with Write Heavy workload and 90.4% lower with Read Only workload.

Figure 13 shows distribution of write latencies. in Write Heavy workload, The 99th percentiles of the write latencies are relatively close to the averages in Figure 11. Other workloads show the similar results. But read latencies are very different. Figure 14 and 15 show distribution of read latencies in Write Heavy and Read Only workloads. The 99th percentiles of the read latencies are much larger than the averages in Figure 12. For example, in case of Read Only workload, with Cassandra and the heterogeneous cluster, the averages of each were 27.58 ms and 2.65 ms, and the 99th percentiles were 486 ms and 104 ms. These results mean that the worst read latencies are hundreds milliseconds though the averages are much lower. The 99th percentile of the heterogeneous cluster was 78.6% lower than Cassandra in Read Only workload and 77.7% lower in Write Heavy workload. Such an improvement of hundreds milliseconds is enough large to be noticed directly by an user.

Figure 16 shows throughput for all the workloads. We set the number of client threads as a moderate number 40 because an excessive number suppresses throughput. Scalability in the number of clients is an important property but part of future work. Throughput of MyCassandra was 13 % lower than that of Cassandra with Write Only workload. In contrast to it, MyCassandra achieved much higher throughput with other three workloads as 11.0 times as high as Cassandra in Read Only workload.

Write performance of MyCassandra was comparable with Cassandra though Cassandra employs only the write-optimized storage engine. Our future work includes investigation of specific causes of the lesser write performance in part of benchmark. We are sure that load balance was involved. Cassandra scatters synchronously all queries to N nodes, which are write-optimized. MyCassandra routes a write query to W nodes, which are write-optimized and on-memory but the number W is less than N. N is three and W is two in the paper. However, the difference in write performance is relatively small and MyCassandra could outperform Cassandra much.

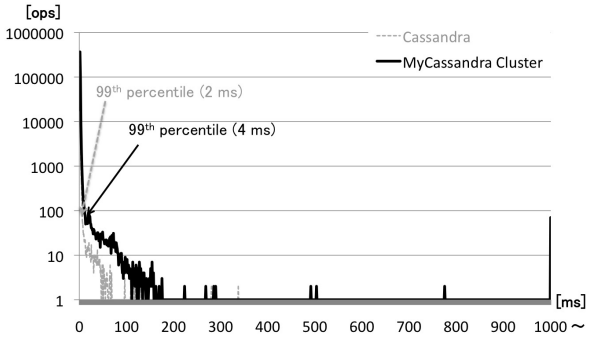

Figure 13. Distribution of write latencies in Write Heavy workload

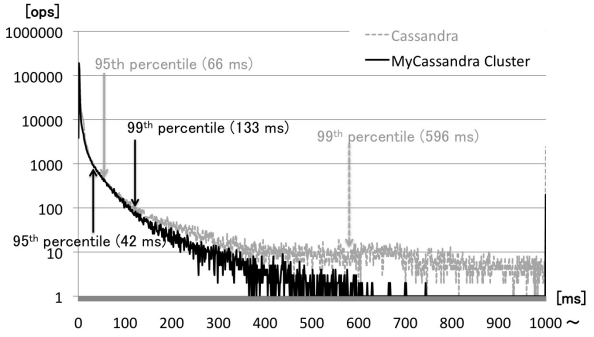

Figure 14. Distribution of read latencies in Write Heavy workload

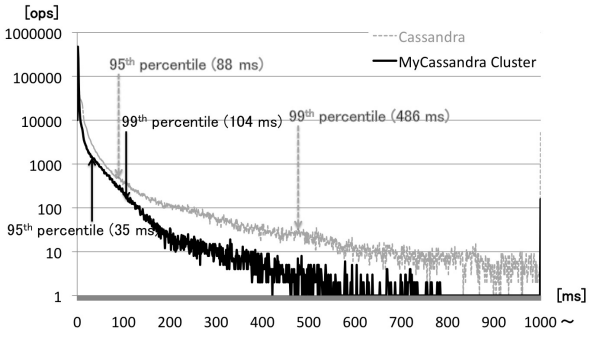

Figure 15. Distribution of read latencies in Read Only workload

5. Other Considerations

This section discusses the pros and cons of the heterogeneous cluster of MyCassandra nodes.

5.1 Memory overflow

The amount of memory is generally smaller and scarcer than persistent storage. There are two means with MyCassandra to deal with a fulfilled on-memory node. The first one is to use a persistence function equipped with an on-memory storage engine, that stores part of data on a persistent storage. Redis provides such a function named on-disk snapshots. The another is to sweep out overflowed data with least-recently-used or a similar policy. The swept data are loaded again to an on-memory node by *ReadRepair*, described below.

In the latter case, an on-memory storage engine node can be regarded as a cache, that provides fast access and less persistence, but can be filled at any time.

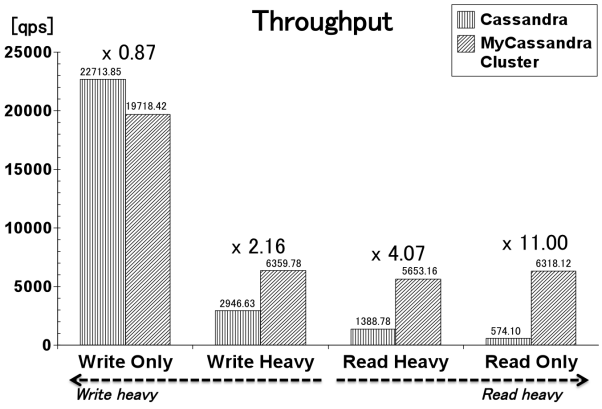

Figure 16. Throughput for each workload

5.2 Fault tolerance and Consistency

An on-memory storage engine node lose its data in case the node fails. Even in the case, the heterogeneous cluster tries to keep *N* replicas and maintain consistency between them. Cassandra's repair functions *ReadRepair* and *HintedHandoff* do them.

HintedHandoff stores changes for a record which should be stored in a failed node to an alternative node. This function works as same as for a persistent storage engine node, and prevents decrease of replicas. *ReadRepair* asynchronously resolves inconsistency between replicas. When a proxy node notices an old replica or a replica has been lost, it fills up consistent *N* replicas. It restores replicas on an on-memory node after it restarts.

At least, the heterogeneous cluster achieves better performance while keeping original Cassandra's fault tolerance and consistency. A heterogeneous cluster with a quorum $(N, W, R) = (4, 3, 2)$ having 2 write-optimized nodes, 1 read-optimized node and 1 on-memory node achieves the same persistence as a Cassandra cluster with a quorum $(N, W, R) = (3, 2, 2)$ having 3 write-optimized nodes because both have 3 persistent nodes. They show similar performance for a write query because it is routed to 2 write-optimized nodes in both cases. But the heterogeneous cluster performs better for a read query. A read query is routed to 2 write-optimized nodes in Cassandra, and 1 read-optimized node and 1 on-memory node in the heterogeneous cluster.

If we regard the on-memory node as a cache, this heterogeneous cluster has 3 persistent nodes as same as the Cassandra cluster and an additional on-memory cache. We discuss such an on-memory cache later.

In Section 4, the heterogeneous cluster places 3 replicas on 2 persistent nodes and 1 on-memory node. The cluster trades persistence that the on-memory node do not provide for its performance. But its fault tolerance is supported not only by persistence but also by replicas and replica maintenance functions mentioned above. Our future work includes quantitative evaluation of fault tolerance. It requires fault models including failure probability and correlated failures [8].

5.3 Fast but persistent devices

Non Volatile RAM (NVRAM) such as Magnetic RAM (MRAM) is a possible storage device to build a read-and-write-optimized storage engine. It provides persistence in addition to read and write performance.

There are various storage devices including PCI-attached SSDs and traditional HDDs in addition to standard SATA-connected SSDs used in this paper. They have their own properties including capacity, performance and cost. NVRAM seems to be a deliverer but it is relatively expensive today. Available devices and the properties are changing with the times. The heterogeneous cluster enables performance coordination by combining them.

5.4 Memory usage

In the heterogeneous cluster, an on-memory node works like a cache. There are other approaches to allocate memory resources. For example, it has become common to place on-memory cache such as memcached in front of data storages. We discuss the approaches here though detailed performance comparison between them is part of our future work.

The front-end cache approach will achieve similar performance to the heterogeneous cluster as long as the both approaches have the same number of persistent nodes and on-memory nodes. The front-end cache works if we accept data reliability in case that a datum is from a single cache. For example, ECC memory will enable it. The heterogeneous cluster enables reading from multiple replicas and comparing them, and it can configure the number of the replicas by nature.

At least, contribution of the heterogeneous cluster is providing a novel method to utilize a read-and-write-optimized device including memory. In addition, its flexibility of cluster configuration enables new applications. For example, it allows simultaneous OLAP workloads while processing OLTP queries. A heterogeneous cluster stores written data to read-optimized nodes though it is asynchronous. OLAP workloads requires sustainable read performance that read-optimized nodes provide. Because of it, an existing write-optimized cluster for OLTP cannot support it. A column-oriented database [19], that is well-suited for OLAP workloads, is a promising candidate to be part of the heterogeneous cluster.

5.5 Disk/memory allocation to nodes

We assume that a server organizing a heterogeneous cluster equips two disks to perform well. A node assigns a dedicated disk to a Bigtable storage engine to process a write query fast with only sequential I/Os.

Nodes on the same server in a heterogeneous cluster share memory equipped with the server. Even distribution to those nodes is the best if each storage engine consumes the same amount of memory per record. But it is not the case though the difference is usually less than an order of magnitude. There should be the optimal distribution.

6. Related Work

This section introduces other modular cloud storages and efforts in achieving both read and write performances.

Anvil [12] is a modular data storage which is composed of components *dTable*. Anvil composes a data storage according to access patterns of an application. Anvil tried to adapt to applications in a single node. Our proposal also adapts to read heavy and write heavy workloads by its modularity but the heterogeneous cluster supports both workloads by combining different nodes as a distributed system.

Cloudy [9] is a proposal for designing a modular cloud storage. It will enables selection of routing scheme and load balance while our proposal focuses storage engines and balance between write and read performance.

Amazon Dynamo [7] adopted Berkeley DB, MySQL and in-memory buffer as its storage engines. An user chooses an engine according to the size of data.

There have been studies on indexing algorithms whose goals include achieving both write and read performance. FD-Tree [11] is an indexing algorithm that works efficiently on SSD. Its perfor-

mance is close to B+-tree in searching and LSM-Tree in updating. Also, it is expected to be adopted for cloud storages.

Fractal-Tree, which is an indexing algorithm implemented in the recent MySQL's storage engine TokuDB [21], are a drop-in replacement for B-Trees. B-Trees are slow for high-entropy inserts because most nodes of the tree are not in memory and most insertions require a random disk I/O, while Fractal-Tree effectively replaces a random I/O with a sequential I/O, which is faster on spinning disks. A modular data store can select an indexing algorithm by selecting a storage engine implementing it. Our future work includes establishment of a method to select an appropriate storage engine according to read/write ratio in an application.

Fractured mirrors [16] is a heterogeneous data store focusing on data partitioning. This approach combines row-oriented N-ary Storage Model (NSM) and column-oriented Decomposition Storage Model (DSM) to operate queries efficiently. Our proposal combines different types of nodes into a single distributed system. It is possible to implement fractured mirrors based on our proposal.

7. Conclusion and Future Work

We proposed a method to build a cloud storage that supports both read heavy and write heavy workloads. A heterogeneous cluster is built from different types of nodes that are write-optimized, read-optimized and read-and-write-optimized, that is, on-memory. We provided those types of nodes by implementing a modular cloud storage MyCassandra, that accepts various storage engines such as MySQL and Redis. Benchmark results demonstrate that the heterogeneous cluster achieved better throughput than the original Cassandra on read heavy workloads. Read latency was up to 90.4% lower and throughput was up to 11.00 times as high as those of Cassandra.

One of our next steps is performance and scalability evaluation of the heterogeneous cluster on servers offered by real cloud services such as the Amazon Elastic Compute Cloud (EC2). We expect that proposed method scales to a large number of nodes because the query processing path does not depend on the the number of nodes, i.e., from a single proxy to replicas. Therefore, our method is independent of the number of nodes.

Next steps include consideration of network proximity. A network connecting a cluster can be far from homogeneous due to rack and data center boundaries. The proposed method places replicas according to node characteristic on read and write performance, but it is not obvious how to incorporate network proximity into the method.

If an application developer knows the read/write ratio of his or her application, a heterogeneous cluster has an opportunity to adjust storage engine ratio to achieve the highest performance. We are investigating a mechanism by which a cloud storage adjusts itself to workloads. The concept of elasticity of a cloud will be continuously extended to such autonomy.

Acknowledgments

This work was supported by MEXT KAKENHI (22680005), and "New generation network R&D program for innovative network virtualization platform and its applications", the Commissioned Research of National Institute of Information and Communications Technology (NICT).

References

[1] The Apache Software Foundation. Apache HBase. http://hadoop.apache.org/hbase/.

[2] 10gen. MongoDB. http://www.mongodb.org/.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *Proc. OSDI'06*, 7:205–218, 2006.

[4] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. In *Proc. VLDB'08*, 2008.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. SOCC 2010*, 2010.

[6] Danga Interactive. memcached. http://www.memcached.org/.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. SOSP 2007*, 2007.

[8] D. Ford, F. Labelle, F. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. OSDI'10*, 2010.

[9] D. Kossmann, T. Kraska, S. Loesing, S. Merkli, R. Mittal, and F. Pfaffhauser. Cloudy: A Modular Cloud Storage System. In *Proc. VLDB'10*, 2010.

[10] A. Lakshman and P. Malik. Cassandra - A Decentralized Structured Storage System. In *Proc. LADIS'09*, 2009.

[11] Y. Li, B. He, R. J. Yang, L. Qiong, and Y. Ke. Tree Indexing on Solid State Drives. In *VLDB'10*, 2010.

[12] M. Mammarella, S. Hovsepian, and E. Kohler. Modular Data Storage with Anvil. In *Proc. SOSP 2009*, 2009.

[13] Neo Technology. Neo4j. http://neo4j.org/.

[14] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-Structureed Merge-Tree (LSM-Tree). In *Acta Informatica*, volume 33 (2), pages 351–385, 1996.

[15] Oracle Corporation. MySQL. http://www.mysql.com/.

[16] R. Ramamurthy, D. David J., and Q. Su. A Case for Fractured Mirrors. In *Proc. VLDB'02*, 2002.

[17] Salvatore Sanfilippo. Redis. http://redis.io/.

[18] R. Sears, M. Callaghan, and E. Brewer. Rose: compressed, log-structured replication. In *Proc. of the VLDB Endowment*, volume 1 (1), pages 526–537, 2008.

[19] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. A Column-oriented DBMS. In *Proc. VLDB'05*, 2005.

[20] The Apache Software Foundation. Apache Cassandra. http://cassandra.apache.org/.

[21] Tokutek. TokuDB. http://tokutek.com/.