Improving Response Time for Cassandra with Query Scheduling

Satoshi FUKUDA[*], Ryota KAWASHIMA[*] Shoichi SAITO[*] and Hiroshi MATSUO[*]

[*]Nagoya Institute of Technology
Gokiso, Showa, Nagoya, Japan
Email: fukuda@matlab.nitech.ac.jp

***Abstract*—A management of large-scale data becomes more important, along with the spread of cloud service and the speed-up of networks. Since data management on a single machine can cause performance and scalability problems, data management across multiple machines has been proposed. Distributed Key Value Store(KVS) is a datastore which manages data across multiple machines. Since distributed KVSs manage data which consists of simple key-value pair, they can achieve scalability easily. Distributed KVSs are widely used in many services managing large-scale data, such as Facebook and Twitter. Distributed KVSs provide interfaces to access key-value pair by simply specifying the key. In this paper, we refer to a query which only obtains a value from a key as a single query. Some distributed KVSs support a range query which obtains multiple values from a key range. However, under mixed query workloads that consist of single and range queries, single queries(which can be executed faster) are forced to wait until preceding range queries are finished. And this results in the increase of average response time. We propose an approach to reduce the average response time by query scheduling. We implemented our method on Cassandra, and evaluation results showed a reduction of the average response time.**

***Index Terms*—Cassandra; Range Query; Query Scheduling;**

I. Introduction

A management of large-scale data becomes more important, along with the spread of cloud service and the speed-up of networks. Many services require scalability in order to dynamically adapt to the growing workloads. However, traditional RDBMSs are hard to achieve scalability because of a requirement of strong consistency and availability for data items.

Distributed Key Value Store(KVS) systems are widely used in cloud services because of their characteristics. They can achieve scalability easily by reducing consistency support and by adopting simple key-value pair. On the other hand, distributed KVSs only support simple operations to obtain data. For example, almost all distributed KVSs provide interfaces to get the key-value pair by simply specifying the key. Normally, only a single value can be obtained by a corresponding key in a query (*single query*). Some implementations , e.g. Cassandra and HBase, support more useful operations which obtain multiple values by the corresponding key range (*range query*). As an example of range query usage, range queries are used to obtain latest posts, when using distributed KVSs to manage the user submitted posts.

Pirzadeh et al. evaluated the performance of range queries[1]. It showed that the average response time of range queries got worse by increasing the amount of the results data size. In real services, one user requires a single post while the others require multiple posts. In general, range queries require more resources, e.g. disk IO, than single queries. Thus, under mixed query workloads that consist of single and range queries, single queries(which can be executed faster) are forced to wait by resource contention. And this results in the increase of average response time.

In this paper, we propose a method to reduce the average response time under the mixed query workloads. Our method schedules queries such that single queries are prioritized over range queries. By avoiding single queries being waited by range queries, our approach can reduce the average response time.

Apache Cassandra[2] is one of the most widely used distributed KVS. Cassandra supports not only single queries but also range queries. We implemented our method into Cassandra, and its evaluation result showed a reduction of the average response time.

The rest of this paper is organized as follows. An overview of Cassandra is given in section II. The proposed method is presented in section III and its implementations is illustrated in section IV. Section V shows the evaluation results. Finally, we conclude this paper in section VI.

II. Apache Cassandra

A. Overview

In this paper, we focus on Apache Cassandra. Cassandra is one of the most widely used distributed KVS implementations, and it was initially developed by Facebook and is currently maintained as an Apache Software Foundation product. Cassandra is mainly designed and implemented based on a data model of Google Bigtable[3] and architecture of Amazon Dynamo[4]. Since Cassandra does not have single point of failure. high durability and availability can be achieved. Cassandra uses eventual consistency model, and its consistency level can be determined by the client. Unlike other distributed KVSs, Cassandra also supports range queries which obtain multiple values corresponding to the key range.

B. Data Model

The data model of Cassandra is shown in Figure 1. Generally, KVS has a simple key-value pair data structure.

978-1-4799-2795-1/13 $31.00 © 2013 IEEE
DOI 10.1109/CANDAR.2013.25

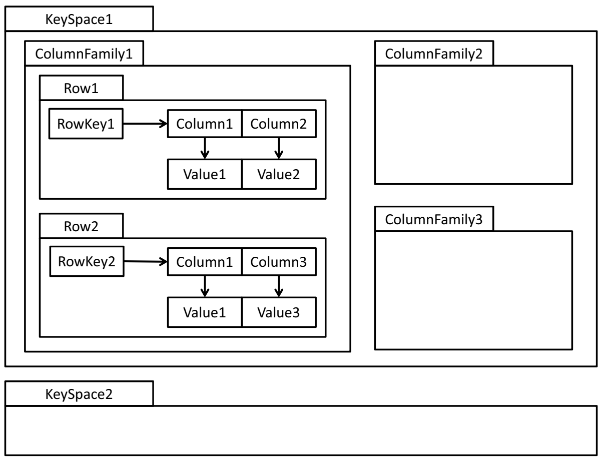

Fig. 1. Data model of Cassandra

While, Cassandra provides more complex data structures such as KeySpace, ColumnFamily, RowKey and Column. The KeySpace is the outermost container for data that is similar to a *database* in RDBMS. The ColumnFamily is a collection of Rows and the Row is a collection of Columns. The Column is the most fundamental data unit in Cassandra and it consists of name, value, and timestamp. These data structures enable us to manage data more sophisticatedly than other distributed KVSs.

C. Data Placement

In distributed KVSs, whole data is managed across multiple machines (*nodes*) and each data is assigned to a *node*. Generally, distributed KVSs assign data to a node based on ConsistentHashing[5] method. ConsistentHashing method uses hashed key to determine the node to place the value. This method works fine even if further nodes are added or removed to/from the cluster. However, the key order is not preserved due to the hashed keys. This results in broadcast request to all of the cluster nodes when executing some range queries under ConsistentHashing. This search method is inefficient and does not achieve scalability. Even worse, range queries return data in an essentially random order and the returned values need to be sorted later. For these reasons, data placement based on ConsistentHashing method is not suitable for range queries.

In order to execute range queries efficiently, Cassandra supports order preserved data placement which treats the key as in byte order (no hashing). In this data placement, range queries can be executed efficiently because values corresponding to the specified key range are placed on a restricted nodes. In addition, range queries return order preserved data, and therefore, the sorting is not needed. Hence, this data placement is suitable if clients require range queries. However, this data placement has some disadvantages in that heavily-lopsided data placement, because real-world data is not written evenly. In this paper, we assume that different types of queries like single and range queries are mixed in workloads and use the order preserved data placement rather than ConsistentHashing method.

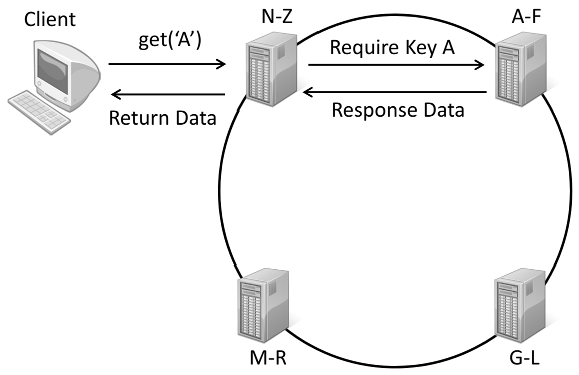

Fig. 2. A flow of the single query

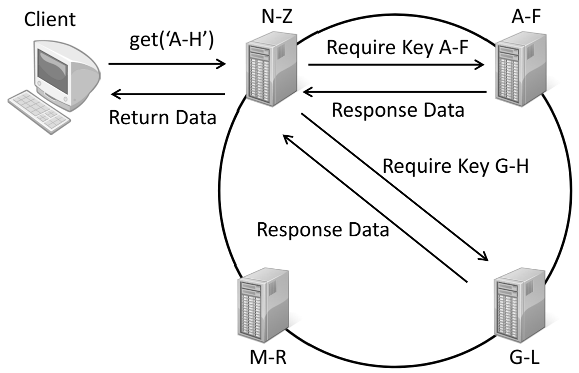

Fig. 3. A flow of the range query

D. Flow of reading Data

Clients can connect to any node in the cluster to read data. If the node does not have the required data, it seeks the data from other nodes as a coordinator. After that, the node returns the data to the client. Next, the details of both single and range queries are described.

1) Single Query: A flow of the single queries is shown in Figure 2. A ring in the figure represents a Cassandra cluster, and the cluster consists of multiple nodes. The letters A-F, G-L, M-R, and N-Z represent the key ranges which are managed by each node. The left most node represents a client node.

When a client performs single queries, it connects to a node of the cluster. In this example, the client connects to a node which manages key range N-Z with key 'A' and the node requests the data to another node which has key range A-F. Then, the node returns the requested data to the first node, and finally the client acquires the data.

If a client connects to a node which manages required key, the node searches data and send it to the client.

2) Range Query: A flow of the range queries is shown in Figure 3. When a client executes range queries, the client connects to a node of the cluster as in the case of single queries. The difference is that clients have to specify two keys in range queries. In this example, the client connects to a node which manages key range N-Z, and it connects to a node which manages key range A-F, and acquires the data corresponding to the key range A-F. Next, the node to which the client is connected connects to a node which manages key range G-L, and requests the remaining data to the node and it returns the requested data to the original node. As a result, the client can get entire values corresponding to the key range A-H.

E. Increasing Response Time of Queries

Single queries which obtain a single value from a single key are usually finished faster than range queries which obtain multiple values from a key range. Since Cassandra executes requested queries in order of arrival, single queries are forced to wait until preceding range queries are finished. And this results in the increase of average response time.

In addition, the sequential nature of the range search in Cassandra also leads to increasing response time of the system. That is, the increase of the cluster nodes can cause further execution time of the system.

In this paper, we propose a method which prevents the increase of average response time of the system.

III. Improving Response Time of Search Queries

In this section, we describe the details of the proposed method. In practice, three improvement methods are proposed to reduce average response time of search queries.

First, average response time of single queries is reduced by scheduling the order of single and range queries. In this scheduling, single queries are prioritized over range queries because they are expected to finish faster than the later queries, and the average response time of single queries can be reduced by preventing the queries from being waited by range queries.

Next, average response time of range queries is also reduced by parallel execution of them. In Cassandra, the coordinator node sequentially sends the requests to all nodes within the range. In our method, the coordinator sends the requests to all the nodes at one time, and each request is handled by them in parallel.

As a results, average response time of range queries can be reduced.

Finally, the scheduling of range queries can also be improved to reduce the average response time. In this scheduling, the priority of range queries is varied depending on the range size. In particular, range queries with narrower key range are prioritized over the other range queries. By this priority settings, range queries which involves fewer nodes are preferentially executed. The range size is determined by how many requests are sent. In addition, their priorities are dynamically changed as the progress of query execution on other nodes. For example, when two range queries have same range size, the average response time can be improved by prioritizing a query that requires fewer search results. That is, the priority of the range query is dynamically changed as the progress of its execution.

In this scheduling, range queries which involve large number of nodes have low priority. Thus, response time of these queries are to be increased. However applying these three methods appropriately, the average response time of search queries can be decreased in Cassandra.

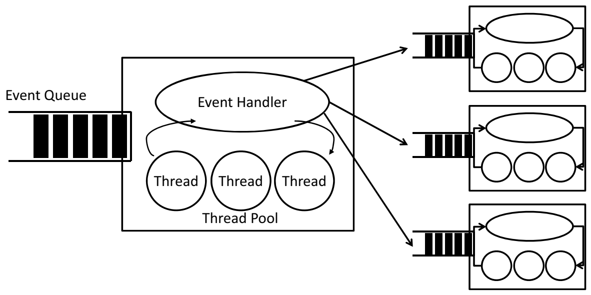

Fig. 4. Each stages of SEDA

IV. Implementation

A. Scheduling Single Queries and Range Queries

Cassandra supports a Staged Event Driven Architecture (SEDA)[6]. In SEDA, it divides an application into *stages*. As shown in Figure 4, the *stage* consists of an *event queue*, an *event handler*, and an *associated thread pool*. The stage is a basic unit of work, and a single operation internally causes state-transition from one stage to another. For instance, a search operation command is enqueued in the *event queue*. *Thread pool* dequeues the operation command from the *event queue* and executes it with the associated *event handler*. In this way, SEDA separates event executions from receipt of requests, and the throughput degradation caused by the growth of the thread can be avoided.

Cassandra's executions are divided into 11 *stages*, such as READ Stage for search operation and MUTATION Stage for data insertion.

In our method, the *event queue* in READ Stage is replaced with the priority queue and the proposed scheduling functions are implemented in this queue. In our implementation, the priority queue is implemented with java.util.concurrent.PriorityBlockingQueue . This priority queue is based on priority heap, thus the cost of enqueue and dequeue is O(1) and O(log(n)). It is considered that the scheduling overhead is negligible. Priority is determined in the following order: single query request commands from the same node, single query request commands from other nodes, and range query request commands.

B. Parallel Execution of Range Queries

For range queries in Cassandra, clients specify start key, end key, and *count* value that indicates how many data items are required by the client. The range search operation can be terminated in mid-flow of the range. Thus, in Cassandra, the coordinator node sends the requests to all nodes within the range sequentially until the number of the results reaches to the *count* value. In this method, the more nodes increase within the range, the more response time of range queries increases.

While, in our method, the coordinator sends the requests to all nodes within the range at one time, and the requests of each node are handled in parallel. In Cassandra, the amount of data items in a node is collected as statistical information. By sharing this information among the nodes, the coordinator can predict the number of nodes to send the requests. In our approach, the coordinator sends the requests to the limited nodes to meet the *count*. Note that the coordinator does not count the amount of data items in the node that corresponds to the start key. The amount is calculated as how many data items are stored in the node. Since the start key of the query and the beginning of the range can be different, the amount of the data items in this node is assumed as zero in our method.

C. Scheduling Range Queries

In the proposed method, priority of range queries is determined depending on the range size. The range size is based on how many requests are sent at one time. If the requests are sent to only a few nodes, the range query request commands have higher priority. While the requests are sent to a lot of nodes, the commands have lower priority. If two queries have same range size, their priorities are determined based on *count*. If the *count* is low, the range query request command has higher priority, and likewise, lower priority is given when the value is large. We implemented this scheduling function to the *event queue* at READ Stage.

Our scheduling function also supports dynamic changes of the priority by observing the progress of other nodes. When a node returns the result of the range query request, the coordinator notifies the nodes which are not finished their commands yet and they know the number of remaining nodes. The notified nodes dequeue the range query request command corresponding to the range query and increase its priority such that the priority is higher than that of other range queries that require more nodes. That is, the range query request command that involves less nodes is executed preferentially.

V. Performance Evaluation

We implemented our method in Cassandra 1.1.2 and evaluated its performance. The performance was evaluated by sending queries which consist of both single and range queries. In this evaluation, YCSB(Yahoo! Cloud Serving Benchmark)[7] was used as a client program.

A. Evaluation Environment

In this evaluation, we constructed a Cassandra cluster consisting of 12 nodes and also installed YCSB 0.1.4 on a client node to send queries. The cluster nodes and the client node were connected via the same hub. The specification of Cassandra cluster nodes are shown in Table I, and the specification of the client node is shown in Table II.

TABLE I. The specification of Cassandra cluster nodes

OS	Linux 2.6.38-8-amd64 Ubuntu 11.04 Server
CPU	Intel(R) Core(TM) i5 CPU 750 @ 2.67GHz
Memory	8GB
Network	1000BASE-T

TABLE II. The specification of a client node

OS	Linux 3.5.0-23-amd64 Ubuntu 12.04 Server
CPU	Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
Memory	32GB
Network	1000BASE-T

B. Evaluation Description

First, 10,000,000 data items were inserted into the Cassandra cluster and each data size was 1KB. Each data item was stored separately to the 1,000 ColumnFamilies and data items were searched from each ColumnFamily. By separating the data items into ColumnFamilies, the number of data items acquired by range queries is to be small. That is, at most 10MB of data items are acquired by a range query and this prevents insufficient memory and increasing data transfer time. Otherwise, all of the data items can reside in memory. Generally, Cassandra handles large-scaled data which cannot reside in memory, and therefore, we separated the data items into ColumnFamilies in this performance evaluation. In addition, in order to clear the data items from memory, we freed pagecache on each node before executing the benchmark.

In this evaluation, clients sent 2000 queries in total and single and range queries were sent at a ratio of 5:5 and 9:1. In range queries, the start key was selected in random order from keys within the Cassandra cluster. *Count* has the uniform distribution between 1 and 10,000. The number of clients which send queries was 125, 250, 500 or 1000. We measured average response time of single and range queries.

We evaluated following six implementations.

(O)	Original
(ReS)	Prioritized Single Queries
(P)	Parallel Execution of Range Queries
(ReS+P)	Prioritized Single Queries, and Parallel Execution of Range Queries
(P+RaS)	Parallel Execution of Range Queries, and Scheduling Range Queries
(A)	Prioritized Single Queries, Parallel Execution of Range Queries, and Scheduling Range Queries

We evaluated each implementation for 10 times and adopted the best result for comparison. In this evaluation, we did not have any replication.

C. Evaluation Result

Figure 5 shows the average response time of single queries when the ratio of single and range queries was 9:1, and Figure 6 shows the average response time of range queries. Likewise, Figure 7 represents the result of single queries at a ratio of 5:5, and the result of range queries at the same ratio is given in 8.

As shown in Figure 5, the the prioritized single queries did not effect in 125, 250 and 500 clients workload. On the other hand, in 1000 clients, the average response time of single queries was reduced in (ReS), (ReS+P) and (A). In (O), the number of the range queries was increased because of the increase of clients. If the number of range queries increases, single queries can be forced to wait until preceding range queries are finished. Hence, the average response time of single queries will increase. In (ReS), (ReS+P) and (A), single queries were executed preferentially than range queries, thus, the influence of the increased range queries was suppressed. In (P), (P+RaS), the average response time of single queries was significantly increased as the number of clients increased. Especially in 1000 clients, the average response time of single queries increased to 2.4 times compared with that of (O).

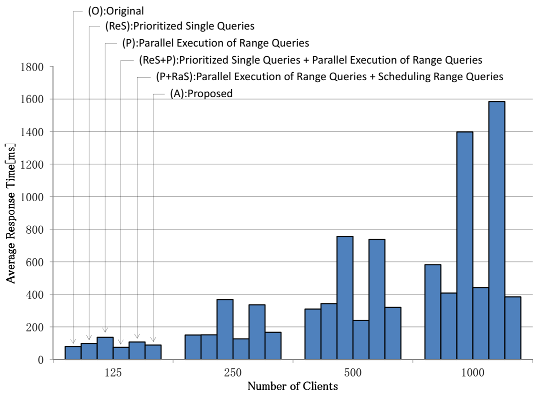

Fig. 5. Average response time of single queries when sending single and range queries at a ratio of 9:1

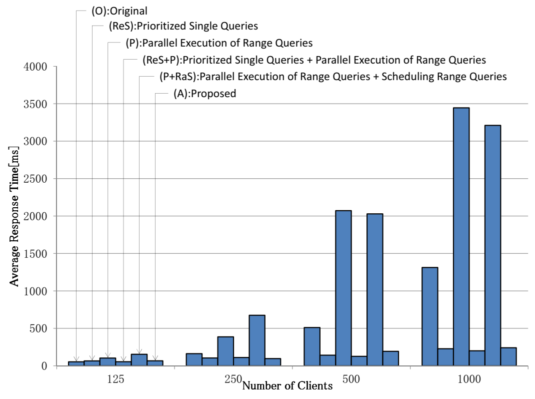

Fig. 7. Average response time of single queries when sending single and range queries at a ratio of 5:5

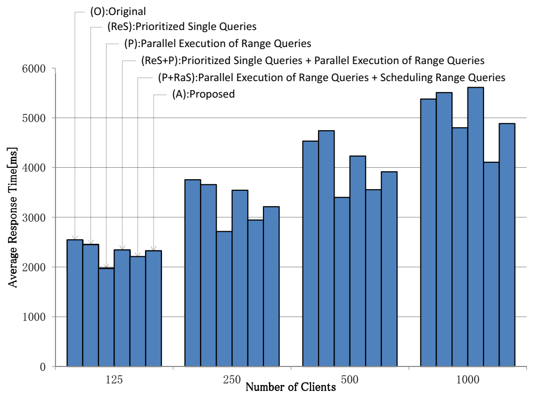

Fig. 6. Average response time of range queries when sending single and range queries at a ratio of 9:1

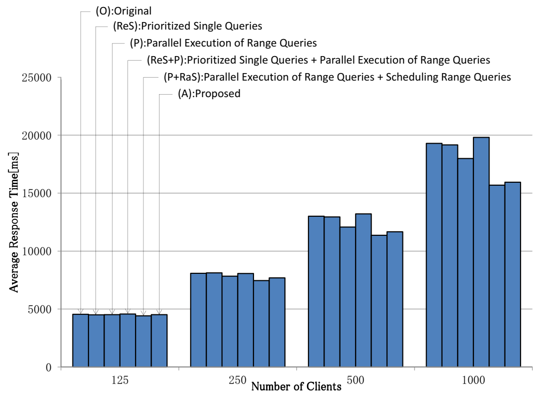

Fig. 8. Average response time of range queries when sending single and range queries at a ratio of 5:5

Because of parallel execution of range queries, requests from each node were sent at one time. As a result, the number of the request commands was increased in a certain time, and delayed single queries were also increased.

Figure 6 shows that the average response time of range queries in (P) and (P+RaS) was reduced compare with (O). Especially in 1000 clients, it was reduced by 10% compared with (O). The waiting time for collecting the results was reduced because of parallel execution of range queries. In 1000 clients, the average response time of (P+RaS) was reduced compare with (P). This is because that scheduling range queries worked effectively. In (ReS+P), the average response time was not reduced. Although range queries were executed in parallel, single queries were eventually executed preferentially. Thus, parallel execution of range queries did not work effectively. However, scheduling range queries (A) worked effectively.

Figure 7 shows that the average response time of (O) was significantly worse than that of the case where the ratio was 9:1. Due to increase in the ratio of range queries, a lot of single queries were forced to wait until preceding range queries were finished. On the other hand, the average response time of (ReS), (ReS+P) and (A) was reduced. Especially in 1000 clients, the average response time was reduced by 80%. As well as sending single and range queries at a ratio of 5:5, the average response time of (P) and (P+RaS) was increased to 2.4 times compared with that of (O). This was caused by the increase of range queries due to the parallel execution of range queries.

As shown in Figure 8, the average response time of (P), (P+RaS) and (A) was reduced. Especially in 1000 clients, the average response time of (A) was reduced by 17% compared with (O). In this evaluation, parallel execution of the range queries and scheduling range queries were worked effectively, because the ratio of range queries was large.

From the above results, the following conclusions can be drawn.

- Executing single queries preferentially is effective to reduce the response time of the single queries, but does not influence the response time of range queries.
- Parallel execution of range queries influences negatively the response time of single queries, but is effective to reduce the response time of range queries
- Scheduling range queries does not influence the response time of single queries, but is effective to reduce the response time of range queries

In our proposed method, these three methods were worked effectively for reducing response time.

VI. Conclusion

In this paper, we proposed scheduling search queries and parallel execution of range queries to improve average response time of search queries. We realized the scheduling methods by setting appropriate priority to each query under the situation that both single and range queries are mixed.

We implemented the proposed method into Cassandra and evaluated it. As a result, the average response time of single queries was reduced in the proposed method compared with original Cassandra. In addition, the average response time of range queries was also reduced by the effect of scheduling and parallel execution of range queries.

Acknowledgment

This work was supported in part by MEXT KAKENHI Grant Number 25400113.

References

[1] P. Pirzadeh, J. Tatemura, and H. Hacigumus. Performance evaluation of range queries in key value stores. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 1092–1101, 2011.

[2] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, Vol. 44, No. 2, April 2010.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, Vol. 26, No. 2, November 2006.

[4] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, Vol. 41, No. 6, pp. 205–220, October 2007.

[5] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *In ACM Symposium on Theory of Computing*, pp. 654–663, 1997.

[6] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services, 2001.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb, 2010. http://github.com/brianfrankcooper/YCSB.