2018 the 3rd IEEE International Conference on Cloud Computing and Big Data Analysis

WOKV: A Write-Optimized Key-Value Store

Ling Zhan, Kan Yu
Division of Information Science and Technology
Wenhua University
Wuhan, China
e-mail: zhan_2003@sohu.com, dunyiyu@hust.edu.cn

Chenxi Zhou, Chenlei Tang
Wuhan National Laboratory for Optoelectronics
Huazhong University of Science and Technology
Wuhan, China
e-mail: 445335538@qq.com, gnatseira@gmail.com

Abstract—**In big data era, LevelDB is widely deployed in enterprise server for massive storage. However, with the growing size of stored data, LevelDB will inevidablely frequently call the background compaction thread to compact data between level, resulting in write amplifications and therefore degrading performance. In this paper, we propose WOKV, a Write-Optimized Key-Value store, to mitigate the LevelDB compact write amplification and improve system performance. WOKV adopts a Least-Rewrite Compaction Strategy to mitigate the write amplification induced by compaction operations, and uses Multi-Thread and Multi-Buffer to improve system performance. Evaluation results show that WOKV outperforms LevelDB significantly.**

Keyword-LevelDB, key-value store, LSM-tree

I. Introduction

In big data era, data is growing with an explosive speed. It is urgent to cope with such massive data. Key-value stores have been widely used in enterprise servers to support multiple applications. Most key-value stores, such as LevelDB[2], RocksDB[3], etc., are implemented based on LSM-tree[1]. LSM-tree writes data in a sequential way in memory. Random writes can be converted to sequential writes and write performance can be improved greatly. However, LSM-tree based key-value stores suffer from compaction operations due to LSM-tree structures. A compaction operation means that the key-value store compacts an SSTable from upper level with all overlapping SSTables in key ranges at the next level. As all key-value entries in an SSTable is sorted, LSM-tree needs resort this data to merge SSTables, therefore a compaction operation requires multiple read and write operations, inducing large write amplification.

There are many prior works optimizing key-value stores. [4] improves key-value stores' performance by applying larger memory to buffer more data. SILT[5] leverages different features of three layouts to reduce key-value memory usage. Wisckey[6] separates keys and values to optimize for flash devices. LSM-trie[9] uses a new prefix tree instead of LSM-tree to index data. [7][8] are designed for flash-based storage devices and they focus on how to co-design with device characteristics, but they seldom focus on the write amplification induced by LSM-tree structure.

In this paper, we propose WOKV, a Write-Optimized Key-Value store, and make following contributions in this paper:

- We propose a *Least-Rewrite Compaction Strategy* to mitigate the write amplification for LSM-tree compaction operations.
- We adopt *Multi-Thread and Multi-Buffer* to reduce the possibility of performance degradation or write blocking.
- We implement WOKV based on LevelDB and evaluate it with the benchmark embedded in LevelDB.

The remainder of this paper is organized as follows. Section II describes the background and motivation of WOKV. Section III presents us the design of WOKV. Section IV gives the evaluation of WOKV. Related work is given in Section V and the conclusion is made in Section VI.

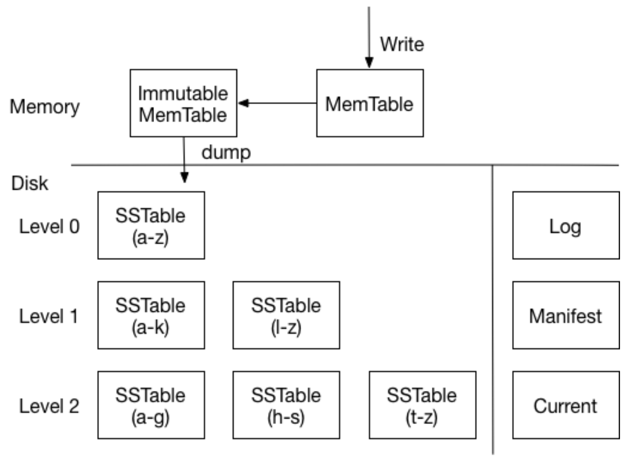

Figure 1. The structure of LevelDB

II. Background and Motivation

A. LevelDB

As illustrated in Figure 1, LevelDB consists of the in memory and on disk data structures. In memory, there are two kinds of table: MemTable and Immutable MemTable. Metadata files and SSTables store metadata and data respectively, and they both are stored on disks. When LevelDB begins to write data, it first writes to the log in case of system crashes or power failures, then it writes to MemTable to buffer data. Once the MemTable is full, data in MemTable will be dumped to Immutable MemTable, which is read-only. Later LevelDB dumps Immutable MemTable to disks if the memory is exhausted. Data on disks can be divided into multiple levels. When the space of a level exhausts or an SSTable is read too many times, LevelDB will do a compaction operation to writes data to the next level. The space of an SSTable in Level *i+1* is several times than the Level *i*'s.

B. Performance Degradation with Data Growing

Compaction is a key mechanism to merge SSTables from different levels, but frequent background compaction result in significant performance degradation. We randomly insert key-value entries under different stored data volumes in LevelDB, and random inserts incur LevelDB compaction operations. As observed in Figure 2, the throughput of LevelDB drops from 8 MB/s to less than 2 MB/s, and the performance degrades performance significantly. Optimize compaction operation overhead is important to achieve sustainable performance in LevelDB.

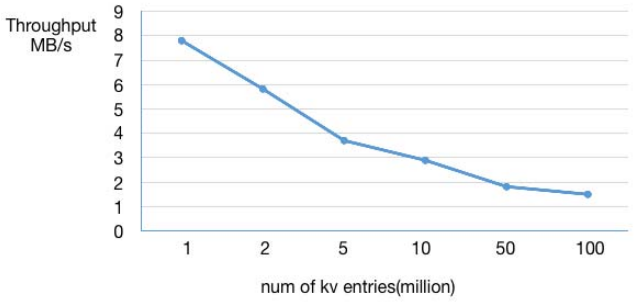

Figure 2. Performance degradation in LevelDB

III. DESIGN OF WOKV

WOKV is designed to be efficient in write-intensive workloads. To achieve this goal, WOKV employs two techniques:

- Least-Rewrite Compaction Strategy to mitigate the write amplification in compaction.
- Multi-Thread and Multi-Buffer to reduce the possibility of performance degradation or write block.

In this section, we first present the architecture of WOKV, then we show the details of these two techniques respectively.

A. System Architecture

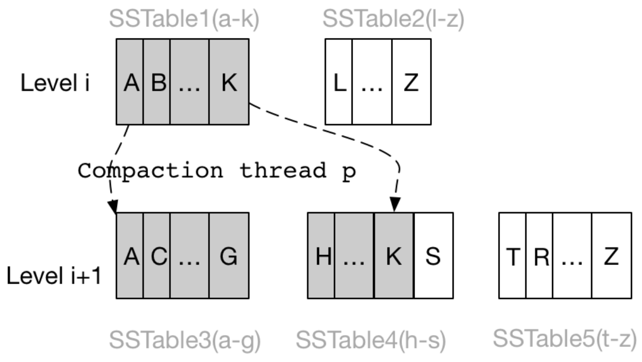

Figure 3. The architecture of WOKV

Figure 3 gives the system architecture of WOKV. LeveDB adopts a simple SSTable-select strategy and uses a single background thread to compact SSTables. Therefore, WOKV adopts Least-Rewrite Compaction Strategy to compact SSTables (details in subsection B). Since LevelDB use a single MemTable to absorb writes issued by applications, WOKV employs multi-thread and multi-buffer to accelerate the absorb to writes (details in Section III-C).

B. Least-Rewrite Compaction Stragety

There are two different kinds of compaction: minor and major. The *minor* compaction means that the key-value store converts the immutable MemTable from memory into an SSTable file and writes it to Level 0. In minor compaction, if there are two entries: a key-value entry and its entry marked as deleted, these two entries will be deleted together. Because key-value entries are write in a sequential way and the delete operation is simply completed by adding a deleted mark. Hence, SSTables converted from minor compaction can be smaller than the size of an ordinary SSTable in disks.

In general, the *major* compaction indicates that LevelDB compacts an SSTable from Level *i* with all overlapping SSTables within the key range in Level *i+1*. All overlapping SSTables will be compacted, sorted and deduped in memory in the *major* compaction. After that, all SSTables will be write back to Level *i+1*. Typically, when doing the compaction operation in Level 0, all SSTables in this level will get involved in the compaction with Level 1, because all key-value entries are not sorted in Level 0. There are two conditions to trigger a *major* compaction:

- *Size Compaction*: The first condition is that the total size of all SSTables in Level *i* reach the corresponding threshold T*i*. It indicates that there is no free space for upcoming data. The key-value store needs to compact SSTables to the next level Level *i+1*. Otherwise, it will make the balance of LSM-tree worse
- *Seek Compaction*: The second condition occurs that the number of an SSTable read exceeds a threshold. If the key-value store reads more than two SSTables in a query, the counter of the first SSTable read will decrease by 1. When the counter equals to 0, the key-value store begins to compact this SSTable.

The *major* compaction is triggered frequently in write-intensive workloads. The compaction can significantly influence the system performance. In consequence, it is vital to determine how and when to trigger a compaction

To address above problems, WOKV uses a Least-Rewrite Compaction Strategy to mitigate the write amplification in compaction, aiming to mitigate the write amplification in compaction. In Least-Rewrite Compaction Strategy, we define a ratio *R* to indicate the size ratio of two levels:

$$R = S_a / S_b. \quad (1)$$

In Formula (1), S_a is the size of SSTable to be compacted in the upper level, and S_b represents the size of all overlapping SSTables in the next level. The larger the *R* is, the lower the write amplification will be in this compaction. And we define *best_ratio* to indicate the best choice to compact， which is initialized to be 0 at the beginning. In Figure 3, *R* equals the size of SSTable 1 divided by the size of SSTable3 and SSTable4. The Least-Rewrite Compaction Strategy is illustrated as follows:

First, WOKV selects a level L_i to compact by size compaction or seek compaction. Second, WOKV iterates L_i and gets the size of current SSTable S_a in each iteration as well as the size of all overlapping SSTables S_b in level L_{i+1}. Third, WOKV gets the $R = S_a / S_b$ in current iteration. If *R* is larger than the *best_ratio*, WOKV updates the *best_ratio* with *R*. Fourth, WOKV can get the *best_ratio* after iterating all SSTables in L_i and choose the best SSTable to be compacted. Fifth, WOKV read the best SSTable and all overlapping SSTables to the memory. Then WOKV sorts in a Merge-Sort manner, deduplicates all repeated key-value entries and writes them back to the next level L_{i+1}.

C. Multi-Thread and Multi-Buffer

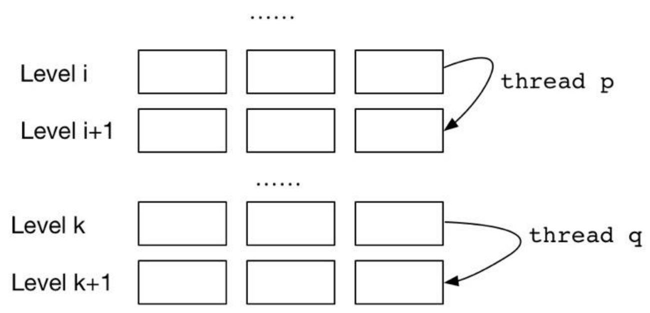

Figure 4. Multiple Thread in WOKV

LevelDB uses a single thread to write data to the buffer zones. There are two different buffer zones in LevelDB. The first one is MemTable, and the other one is the read-only Immutable MemTable. LevelDB first writes to MemTable. When MemTable becomes full, LevelDB dumps data from MemTable to Immutable MemTable. However, if the dump operation is not complete, LevelDB will block all write operations until MemTable is available again, resulting in degrading system performance greatly. Thus, LevelDB doesn't perform well in write-intensive workloads.

A naive approach is to increase the size of MemTable. However, with the increasing capacity of MemTable, it occurs that the performance of *major* compaction and *Get* operation becomes worse. Because the size of MemTable is aligned with an SSTable.

To solve above problems, WOKV employs *Multi-Thread and Multi-Buffer* to further boost LevelDB performance, for heavy write traffic in particular. Adding more buffer zones in memory may result in more resource usage, but it indeed reduces the possibility of performance degradation and write blocking. Multiple buffer zones need extra buffer zone management. WOKV uses a simple but efficient method to manage these buffer zones. When there are multiple buffer zones become immutable due to capacity shortage, WOKV adds these zones to a queue to be compacted. And the background compaction thread will dump data in these zones to persistent devices in an FIFO order. To ease the burden of compaction in Level 0, WOKV checks that whether there are any overlapping SSTables in Level 0. If not, data in buffer zones will be flushed to higher level.

WOKV uses multiple threads to do *Write* and *Compaction* operations in parallel, as illustrated in Figure 4. Compared to the single thread in LevelDB, WOKV can improve system performance greatly. However, there exists some issues in applying multi-thread directly. A *Write* operation involves an append to log in disk and an insertion to MemTable. The former one is slow to write to disk, and the latter one is very fast to complete in memory. Thus, WOKV modifies the log structure and MemTable to support multi-thread. MemTable is implemented based on skip-list. Read of skip-list is thread-safe, but write is exclusive to each other. WOKV adds lock support to write of skip-list and log, in consequence, all operations in log and MemTable is thread-safe.

Compaction consists of *minor* compaction and *major* compaction. For minor compaction, the number of compaction thread depends on the number of buffer zones. By default, WOKV uses 2 threads for *minor* compaction to read MemTables from the queue. It is more complex to apply multi-thread in *major* compaction. LSM-tree is divided into many levels and keys in SSTables of Level 0 can be overlapped with each other. Thus, compaction in Level 0 has higher overhead. For major compaction, WOKV uses 3 threads by default. One thread is used to compact Level 0, one is for Level 1 and Level 2 and the rest one is used for other levels. Note that a level can only be accessed by one thread, because it may conflict data consistency when a level involves in compactions from two threads. In Figure 4, Level i and i+1 are accessed by `thread p`, so other LevelDB compaction threads are not allowed to access these two levels.

IV. EVALUATION

In this section, we evaluate WOKV to answer the following questions:

- What's the overall performance improvement?
- How does WOKV perform compared with LevelDB as the stored data growing?
- How does WOKV improve performance under multi-thread?

A. Experimental Setup

The experiments are conducted on an X86 server with Intel Duo CPU T6600@2.2GHz, and 6G memory of 1333MHz. The Linux release version is Ubuntu-15.10-desktop-amd64 with a kernel version Linux 4.2.0-16-generic. And we use a 120GB Kinston SSD as the storage device. And we use the embedded benchmark *db_bench* of LevelDB to evaluate the system performance.

For simplicity, we set the size of each key-value entry with a particular value. The size of key and value is 16Bytes and 100Bytes respectively. The number of levels is 7. The size of MemTable and SSTable in Level 0 is 4MB, and the capacity of Level 0 and Level 1 is 20MB and 100MB respectively. The size of the other levels will increase with a factor of 10.

B. *Overall Performance Evaluation*

In this section, we will evaluate the overall performance of LevelDB and WOKV in two cases: sequential read/write and random read/write. Note that random/sequential in key-value stores depends on whether key-value entries are in sequential/random dictionary order or not. We measure the performance by the throughput exported by *db_bench*.

Sequential Read/Write: We compare the sequential performance of LevelDB and WOKV by reading 10 million key-value entries in sequential after appending them. And Figure 5 shows the comparison of LevelDB and WOKV in sequential workload. WOKV outperforms LevelDB in both sequential read and write. In particular, the throughput of sequential write of WOKV is 1.3 times of LevelDB. It is due to the application of multi-thread and multi-buffer, and we will discuss it in detail in subsection D of this section. LevelDB only has two buffer zones, and Immutable MemTable is read-only. In consequence, when the write traffic is heavy, LevelDB will block all write operations until the MemTable is available. By leveraging multi-thread and multi-buffer, WOKV can provide larger buffer space and convert MemTable into SSTable vastly.

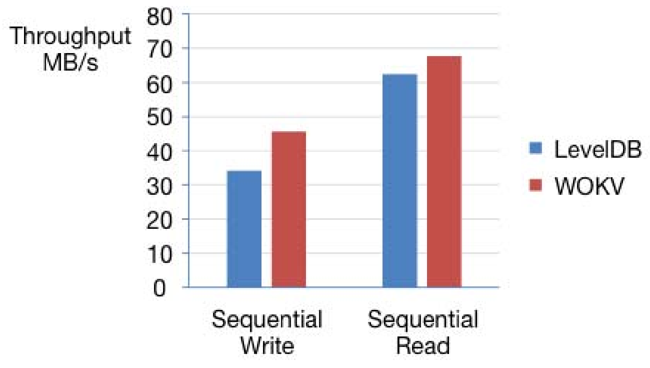

Figure 5. Evaluation of sequential workload

Random Read/Write: We compare the random performance of WOKV and LevelDB by inserting and reading 100 million key-value entries in a random order. Considering the compaction mechanism of LSM-tree based key-value stores, we will read data in a minute after all compaction operations are completed. Figure 6 gives us the evaluation results of random workload. The random write performance of WOKV is higher than LevelDB about 42%, and the random read performance of WOKV is higher than LevelDB about 47%. The write performance improvement is due to the same reason of sequential workload, because multi-thread can alleviate write performance degradation or block significantly. And the random read performance improvement is because there are multiple buffer zones in memory. Read requests will hit the buffer zones in memory, in consequence, read performance can be improved.

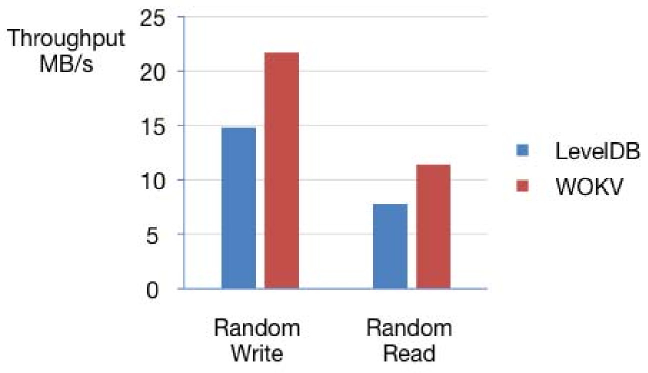

Figure 6. Evaluation of random workload

C. *Evaluation with Heavy Traffic*

WOKV is designed to be efficient in write-intensive workloads. Thus, we compare WOKV with LevelDB under heavy write traffic. Figure 7 gives the comparison between WOKV and LevelDB under 1 million insertions to 200 million insertions. In Figure 2, we can see that with the growing size of insertion, write performance of the two key-value stores degrades. It is because LSM-tree needs to compact data to ensure data balance. It is likely to compact more overlapping SSTables if data to be compacted in current level is large due to multiple levels structure. Write amplification induced by compaction can be very large. The worst case is that a compaction in Level 0 can induce compactions to the bottom level. The more data it compacts, the worse system performance become.

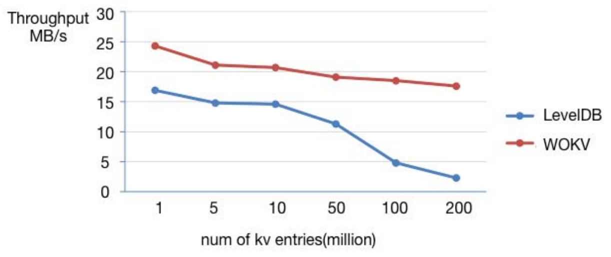

Figure 7. Evaluation with heavy traffic

D. *Evaluation with Multi-Thread*

LevelDB uses a single thread to compact SSTables, and WOKV adopts multi-thread to improve system performance. Hence, we evaluate the performance of multi-thread under 1 million, 10million and 50 million key-value entries insertion. As illustrated in Figure 8, by increasing the number of compaction threads, write performance can be improved. However, when insertion data is small(1 million), write performance improved is limited. When insertion data is large(50 million), we can see that write performance can be improved greatly. Because under write-intensive workloads, especially in heavy write traffic, frequent background compaction operations become the bottleneck of whole system. It is beneficial to add more threads to do the compaction operations. By leveraging multi-CPUs in hardware, it is easy to ease the burden of compaction so that write performance degradation or block can be reduced.

When insertion data is small, background compaction operations are not performed frequently. Therefore, it can improve limited performance when insertion data is small. Only when insertion data is large can improve system performance greatly by adding multiple threads. Hence, it is efficient for WOKV to perform under write-intensive workloads, especially in write heavy traffic.

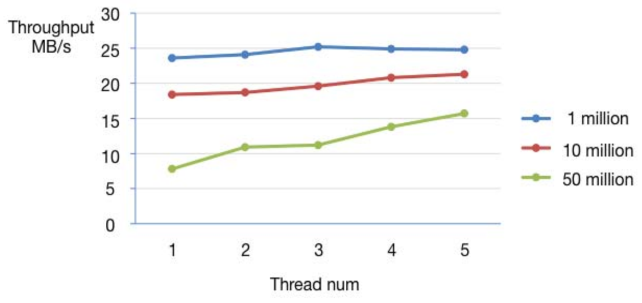

Figure 8. Evaluation with Multi-Thread

V. REALTED WORK

BerkeleyDB [11] is an originator product of modern key-value stores. Now it is one of the members of Oracle databases. It is a light-weighted embedded database. It is simple, stable and high-performance. It buffers data in memory and flushes it later together to disks. Memcached [10] is a memory key-value store. It is designed to be the high-performance object cache. By buffering data and object data structure, it reduces the numbers of read/write to the back-end system. Cassandra [13] is a hybrid key-value store developed by Facebook. It aims to store simple data formats, such as mails at the beginning. Inspired by Google Big Table [12] and Amazon Dynamo [14], it becomes a loose-structured distributed database. Compared with other databases, Cassandra is a distributed network service consisted of multiple data nodes. Redis [15] is an open-source key-value store developed by VMWare. It can be employed as memory storage or persistent storage. It supports multiple data structures, such as String, List, Set, Hash, etc. LevelDB [2] is developed by Google. It uses LSM-tree to store data on disk and converts random writes to sequential writes to improve performance.

VI. CONCLUSION

We propose WOKV, a Write-Optimized Key-Value store, to improve LevelDB performance for write-intensive workloads. By employing *Least-Rewrite Compaction Strategy* and *Multi-Thread and Multi-Buffer*, WOKV significantly improves performance under heavy write traffic. Evaluation results show that WOKV outperforms LevelDB a lot. And we can further explore read optimizations as the future work, such as bloom filter optimization.

ACKNOWLEDGMENT

We thank Peng Xu from WNLO for his help of writing of this paper. This work was sponsored by Research Project of Hubei Provincial Department of Education (B2017341).

REFERENCES

[1] O'Neil P, Cheng E, Gawlick D, et al. The log-structured merge-tree (LSM-tree)[J]. Acta Informatica, 1996, 33(4): 351-385.

[2] Ghemawat S, Dean J. LevelDB[J]. URL: https://github. com/google/leveldb,% 20http://leveldb. org, 2011.

[3] Facebook, "Rocksdb, a persistent key-value store for fast storage enviroments." http://rocksdb.org/, 2016

[4] Yue Y, He B, Li Y, et al. Building an Efficient Put-Intensive Key-Value Store with Skip-Tree[J]. IEEE Transactions on Parallel and Distributed Systems, 2017, 28(4): 961-973.

[5] Lim H, Fan B, Andersen D G, et al. SILT: A memory-efficient, high-performance key-value store[C] Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. ACM, 2011

[6] Lu L, Pillai T S, Arpaci-Dusseau A C, et al. WiscKey: separating keys from values in SSD-conscious storage[C]//Proceedings of the 14th Usenix Conference on File and Storage Technologies. USENIX Association, 2016: 133-148.

[7] L. Marmol, S. Sundararaman, N. Talagala, R. Rangaswami, S. Devendrappa, B. Ramsundar, and S. Ganesan, "Nvmkv: A scalable and lightweight flash aware key-value store," in 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14), 2014.

[8] B. Debnath, S. Sengupta, and J. Li, "Skimpystash: Ram space skimpy key-value store on flash-based storage," in Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, 2011, pp. 25–36.

[9] Wu X, Xu Y, Shao Z, et al. LSM-trie: an LSM-tree-based ultra-large key-value store for small data[C]//Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference. USENIX Association, 2015: 71-82.

[10] Fitzpatrick B, Vorobey A. Memcached: a distributed memory object caching system[J]. 2011.

[11] Olson M A, Bostic K, Seltzer M I. Berkeley DB[C]//USENIX Annual Technical Conference, FREENIX Track. 1999: 183-191.

[12] Chang F, Dean J, Ghemawat S, et al. Bigtable: A Distributed Storage System for Structured Data[J]. To appear in OSDI, 2006: 1.

[13] Lakshman A, Malik P. Cassandra: a decentralized structured storage system[J]. ACM SIGOPS Operating Systems Review, 2010, 44(2): 35-40.

[14] DeCandia G, Hastorun D, Jampani M, et al. Dynamo: amazon's highly available key-value store[J]. ACM SIGOPS operating systems review, 2007, 41(6): 205-220.

[15] Redis, URL: http://redis.io/