2019 IEEE 35th International Conference on Data Engineering (ICDE)

DirectLoad: A Fast Web-scale Index System across Large Regional Centers

An Qin[1], Mengbai Xiao[2], Jin Ma[1], Dai Tan[1],
Rubao Lee[3], Xiaodong Zhang[2]
[1]Baidu Inc., {qinan, majin, tandai02}@baidu.com
[2]The Ohio State University, xiao.736@osu.edu, zhang@cse.ohio-state.edu
[3]United Parallel Computing Corporation, lirb@unipacc.com

Abstract*—The freshness of web page indices is the key to improving searching quality of search engines. In Baidu, the major search engine in China, we have developed DirectLoad, an index updating system for efficiently delivering the web-scale indices to nationwide data centers. However, the web-scale index updating suffers from increasingly high data volumes during network transmission and inefficient I/O transactions due to slow disk operations. DirectLoad accelerates the index updating streams from two aspects: 1) DirectLoad effectively cuts down the overwhelmingly high volume of indices in transmission by removing the redundant data across versions, and mutates regular operations in a key-value storage system for successful accesses to the deduplicated datasets. 2) DirectLoad significantly improves the I/O efficiency by replacing the LSM-Tree with a memory-resident table (memtable) and appending-only-files (AOFs) on disk. Specifically, the write amplification stemming from sorting operations on disk is eliminated, and a lazy garbage collection policy further improves the I/O performance at the software level. In addition, DirectLoad directly manipulates the SSD native interfaces to remove the write amplification at the hardware level. In practice, 63% updating bandwidth has been saved due to the deduplication, and the write throughput to SSDs is increased by 3x. The index updating cycle of our production workloads has been compressed from 15 days to 3 days after deploying DirectLoad. In this paper, we show the effectiveness and efficiency of an in-memory index updating system, which is disruptive to the framework in a conventional memory hierarchy. We hope that this work contributes a strong case study in the system research literature.

1. Introduction

Baidu [1] is the major Internet search engine in China, and its index updating system continuously builds the index data from countless crawled web pages and ships them to nationwide data centers, where the index data are placed in a key-value storage system for promptly serving billions of users. As the Internet itself is evolving, the speed of index updating takes a significant role in determining the searching quality.

1.1. Background

An overview of the index building and updating system of Baidu is shown in Figure 1. In this section, we will briefly introduce how the index data are generated and distributed to the regional data centers, related technical issues, and our solution to address the issues.

1.1.1. Index Building. The web crawlers located in a dedicated data center (data center#0 in Figure 1) are crawling the web pages round by round, and in the same data center, all the index datasets are prepared for further distribution to regional data centers. The web crawlers download a document identified by its uniform resource locator (URL) only if it has been modified since last round of crawling. In Baidu, the crawled documents in the amount of petabytes are categorized into VIP level and non-VIP level according to their content quality, the government censorship, and the popularity at the crawling time (e.g., the breaking news). The VIP level data serve more than 80% user queries while consuming only a few TBs of storage space. Building and updating indices generated from both kinds of data are exactly the same so we will not distinguish them in Figure 1.

The crawled pages are fed to the index building engine for generating indices represented as key-value pairs. The *forward indices* are firstly generated in the form of `<URL, terms>`, where the `terms` are a list of words dismantled from the document content. The *summary indices* are generated separately from the forward indices, and the key is also the `URL` but the value is a summary of the document. Once a collection of forward indices have been crafted, the *inverted indices* can be updated. The key of an inverted index is a `term` and its value is a chain of `URLs` representing the documents that contain the `term`. A summary index entry is denoted as `<URL, abstract>` while an inverted index is `<term, URLs>`. In Figure 1, the red arrows represent the summary indices and the blue arrows represent the forward indices combined with inverted indices. A search request to a search engine is at first broken into couples of `terms`. For each `term`, the corresponding `URLs` are retrieved from the inverted indices. These `URLs` are ranked and only the most related ones are returned to the users with their `abstracts` gathered from the summary index.

2375-026X/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDE.2019.00195

IEEE computer society

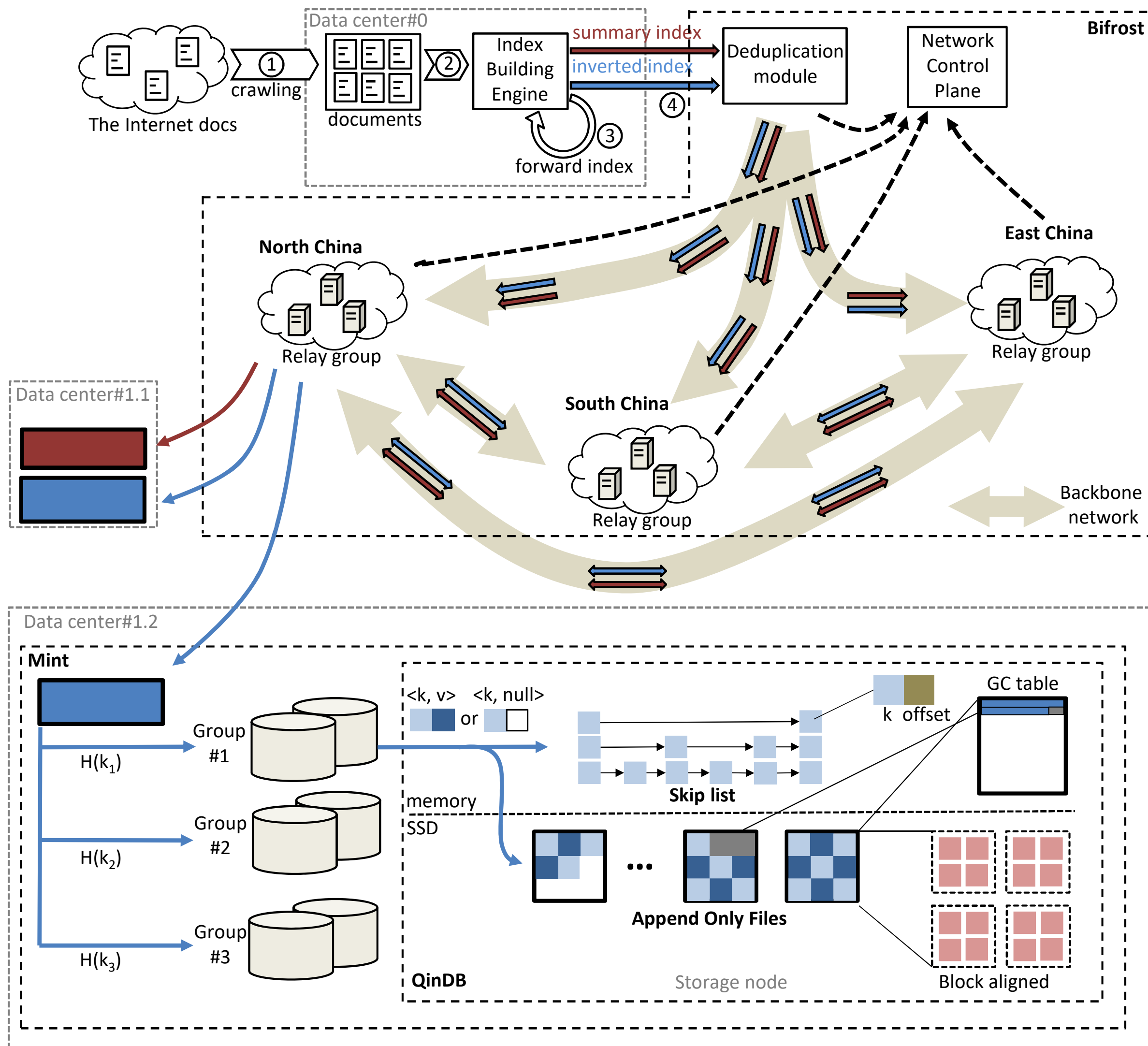


Figure 1: An overview of Baidu's index building and updating system

1.1.2. Index Updating. The data center building the indices (data center#0) keeps sending *slices* of index data in GBs every hour to nationwide data centers. There are in total six data centers in China deployed to accept Baidu's index data and they are evenly distributed in three regions, which are North China, East China, and South China. The inverted indices are stored in all six data centers for low-latency document retrieval while the summary indices can only be found in three ones due to the high storage cost. An example of data center deployment in North China is shown in Figure 1 (data center#1.1 and data center#1.2). For one round of web crawling and selection, the corresponding index data are tagged with an advancing version number. When the index data arrive at a data center, they are arranged to the storage nodes, in which at most four versions of index data persist. In our production environment, two subsystems, *Bifrost* and *Mint*, are designed and implemented to cope with the network transmission and data storage, respectively.

1.2. Problem Statement and Our Solutions

The first obstacle of speeding up the index updating comes from the increasingly high data volumes. In order to react to unpredictable glitches and failures in the production environment, the index data have to be versionized so that rollback to a functional version could be the last resort. But this introduces a tremendous volume of data to be delivered for every update towards a new version, which is likely to overwhelm the backbone networks interconnecting regional data centers. Since around 70% newly generated index data are found to be duplicate compared to its previous version in our production workload, recognizing and removing these redundancy would be a promising solution for bandwidth saving. However, this invalidates the regular data access operations such as `GET(key)` or `DEL(key)` in the key-value

storage system because some keys are preserved without the value fields. Therefore, we need to remold the regular operations in the key-value store system to adapt to the deduplicated key-value pairs.

Another concern is related to poor I/O performance in solid-state storage devices (SSDs). As the state-of-art databases for key-value data (e.g., LevelDB [2] and RocksDB [3]) adopt LSM-Tree [4], [5] for high write throughput, we have observed that in our system the inherent write amplification of this data structure can not be effectively compensated by sequential writes to the SSD any more. Wisckey [6] attempts to alleviate the write amplification by separating the storage of keys and values, but the LSM-Tree is retained for keeping keys sorted. Sorting data on the disk has to read and write data repeatedly so that the write amplification is unavoidable. On the other hand, the write amplification also exists at the hardware level because the SSD can only erase data as a whole block (e.g., 256 KB) while writing data is in a smaller granularity (e.g., 4 KB). As a result, the inefficient I/O transactions have to be well addressed before we can implement a system achieving fast index updating.

In this paper, we present *DirectLoad*, a web-scale index updating system, which consists of two components, *Bifrost* (Data center#0 in Figure 1) and *Mint* (Data center#1.2 in Figure 1). Bifrost is responsible for delivering index data to regional centers, and Mint is a distributed key-value store system for index data in each regional data center. DirectLoad effectively and systematically addresses the above-mentioned two challenges in the following ways. First, Bitfrost is able to reduce the data volume to be delivered in regional centers by deduplications. This effort may leave some key-value pairs incomplete, which is addressed by Mint for additional operations to make key-value pairs complete. Second, instead of building an LSM tree in the memory and disk storage, we have implemented a novel and effective data structure of Mint to achieve the goals of fast data accesses and best utilization of SSD storage. Mint consists of a memory resident table, termed as memtable and appending-only-files (AOFs) in SSD storage. The memtable can be a sorted data structure to store keys, such as a tree structure or a list, where data reads are fast because accesses are in-memory in sequence. Since coming index entries are appended in SSD disks in a block-aligned way, the number of writes is minimized. In addition, we have used a lazy garbage collection (GC) policy to make usable space. The efforts of AOFs and lazy GC have greatly reduced write amplifications at both the software- and hardware-level. The improvement of data access performance by Mint may lead a longer recovery time after the main memory is in a failure mode because memtable has to be rebuilt based on the entire AOFs in SSD. In addition, a lazy GC may also needs more SSD space. However, these two side effects are manageable, and trivial compared with what we have accomplished. The experiments conducted with our production workloads show that Bifrost can reduce up to 63% network traffic by deduplications, and Mint raises the write throughput by 3x compared with a conventional LSM-tree structure. With the support of Bifrost and Mint, DirectLoad has successfully reduced the index updating cycle from 15 days to 3 days.

The rest of the paper is organized as follows. Section 2 presents the design of Bifrost and Mint in DirectLoad. After discussing implementation details of DirectLoad in Section 3, we evaluate the performance of our system in Sections 4 and 5. Section 6 introduces the related work and Section 7 concludes the work.

2. System Design

To elaborate the design choices in DirectLoad, we will outline our system first. Then we describe how index data are delivered to the regional data centers by Bifrost, and how Mint stores the arriving data in a data center. On each storage node, a key-value storage engine named *QinDB* (Quick-Indexing Database) is installed to optimize the write throughput to SSDs (See Figure 1).

2.1. In-memory data processing with SSD storage

Having looked into the data structures of LSM-tree [4], LSbM-tree [5] and Wisckey [6], we have determined they may not be suitable in our production system. All of these designs build LSM-tree in hard disks, which is not efficient in our SSD storage since range query accesses in SSD are not very cost-effective and frequent compactions in LSM-tree are not affordable for SSD. A compaction buffer is built in LSbM-tree to minimize the LSM-tree compaction induced buffer cache invalidations. Since we have built a sorted data structure in memory for fast data accesses, buffer cache is not very critical in our system. The incoming data stream is written into our SSD storage in an appending format at a minimal cost, which is then placed in our memtable in a sorted format in the main memory. Since only keys are stored in memory, there are sufficient memory space to build a fully sorted data structure in memory. The data accesses can be very fast in memory to satisfy high quality search requirement.

Furthermore, in a conventional KV-store with a hashing mechanism, frequent indexing operations can cause a high number of random accesses in memory, reducing KV throughput [7]. In DirectLoad, key-value store is implemented by the sorted keys in memtable and fast accesses to their values in SSD without a hashing table, significantly improving the access efficiency.

One disadvantage of this design is that the memtable recovering can be relatively slow after an electricity outrage compared with the data structure with an LSM-tree in SSD. However, our design is highly preferred in practice for three reasons. First, the chance to rebuild the memtable is very rare according to our daily operation experiences, though it is checkpointed periodically. Second, the cost to build an LSM-tree on SSD is quite expensive and not suitable due to its life span based on limited write cycles. Finally, our simple appending format in SSD not only reduces the write amplification but also supports random accesses for values, which is best suitable for SSD.

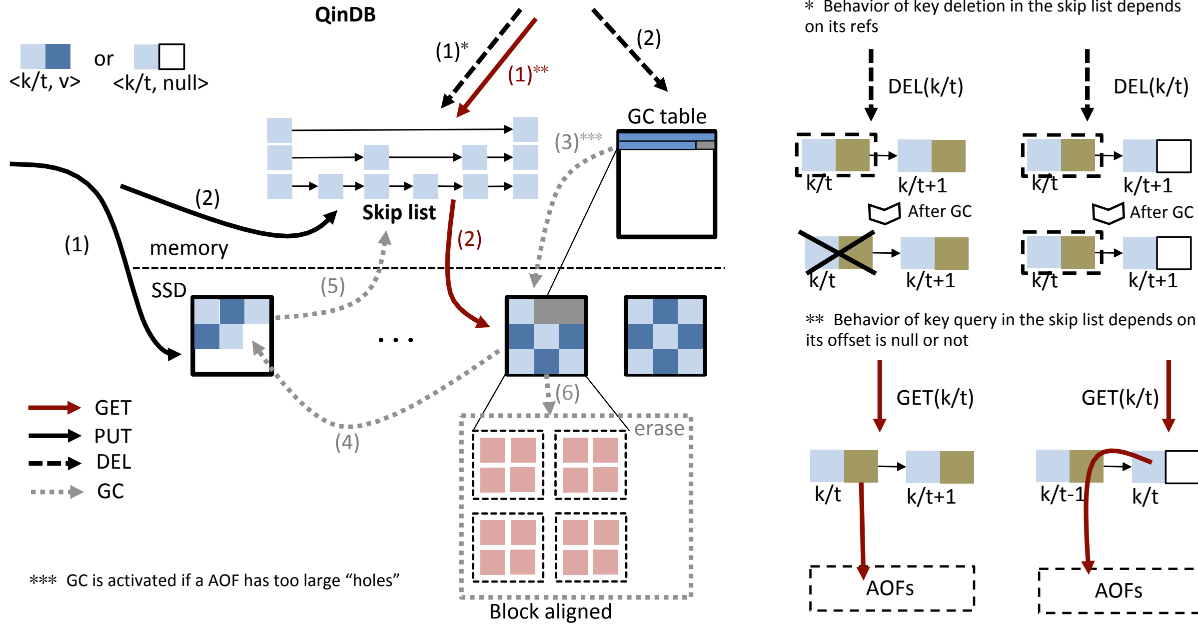


Figure 2: The regular operations (GET, PUT and DEL) in QinDB. All of these operations have to mutate to adapt the new paradigm of key-value pair.

2.2. Bifrost

Modification to a web page in a short period rarely lead to semantic changes of this document. As a result, the index data generated from a new round of crawling usually stay the same. Our data show that on average 70% index data are exactly the same between two consecutive versions. Bifrost removes the redundancy by comparing the signatures of index data between consecutive versions. Only if the signature differs, a key-value pair is forwarded to the network transmission, otherwise the value field will be removed before delivery.

The index data with or without the value field are transmitted to three regional relay groups continuously. For each relay group, there are 20∼30 relay nodes caching and relaying the data to the two data centers located in the same region. Since the backbone networks interconnects every pair of the three relay groups, we have opportunities to optimize the data transmission by flexibly arranging data streams to circumvent the channels sustaining high traffic. In order to timely understand the inter-regional network traffic, a centralized network monitoring platform keeps collecting the real-time network statistics from the relay groups, predicts the available bandwidth resources of the network channels, and directs how the index data should be delivered to the relay groups.

More importantly, Bifrost must ensure that individual data streams, i.e., summary indices and inverted indices, arrive at all data centers simultaneously due to two reasons: 1) the index data of different types are generated continuously but there are no sufficient space in the intermediate nodes to cache them; 2) while the relay nodes also serve other applications, the general-purpose resource management module is unaware of index traffic pattern so that it will revoke the allocated bandwidth if one data stream has stopped. Reallocation of network resources is unpredictable and could be time consuming. In practice, we empirically reserve 40% bandwidth for summary indices and 60% bandwidth for inverted indices.

2.3. Mint

When the index data have arrived in a regional data center, Mint directs how the key-value pairs persist on the disks of storage nodes. The key-value pairs are dispatched to storage nodes according to $H(k)$, where $H(\cdot)$ is a hash function and k denotes the key, for load balance (See Data center#1.2 in Figure 1). However, Mint avoids to map $H(k)$ directly to the storage node due to the poor scalability. The storage nodes are organized in groups and the $H(k)$ is mapped to a group. In this way, Mint can add/remove storage nodes in the groups without redistributing the stored key-value pairs. Furthermore, three replicates of a key-value pair are distributed to different storage nodes in the same group for reliability. Requests to a specific key can be sent

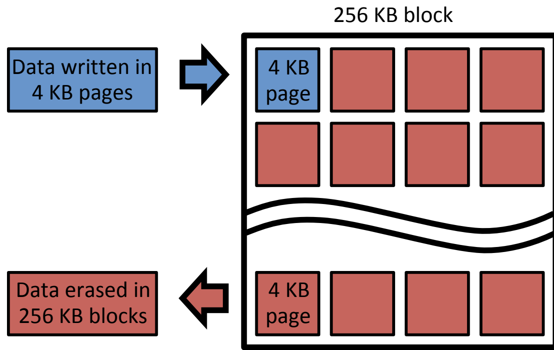


Figure 3: The asymmetric operations of write and delete on SSDs

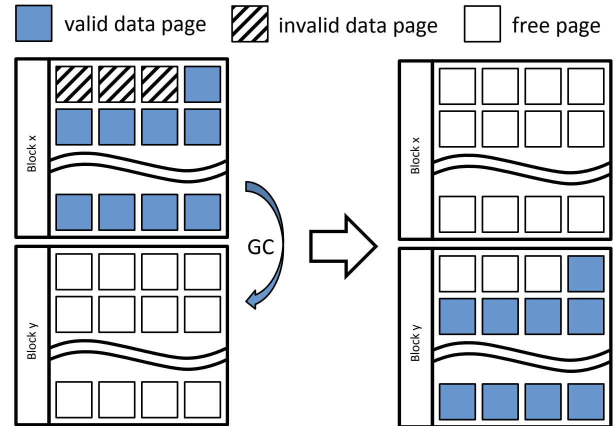


Figure 4: The GC mechanism on the SSD incurs both read and write amplification.

to the relevant nodes in parallel without incurring additional query latency.

For each storage node, we have developed a novel key-value storage engine, namely QinDB, for effectively and efficiently accessing the index data. In order to correctly serve index data that have been deuduplicated, QinDB mutates the regular data access operations. Besides, QinDB adopts a memtable in conjunction with AOFs in SSD instead of LSM-Tree for minimizing write amplification.

QinDB: The key-value pairs are appended to the AOFs and the keys are sorted in a memory-resident *skip list* [8]. The AOFs are append-only files with a fixed size of 64 MB and it includes a GC mechanism for revoking the space occupied by the deleted data. In this way, sorting operations are only conducted in the main memory and we only need to address the write amplification in the GC. In the skip list, an item is composed of a key and the offset referring to the actual position of the value in the AOFs. For a storage node, there is only one SSD of 2TB installed so that the skip list is able to fully reside in the main memory without a huge amount of values.

As the datum of indices have changed to a key carrying a value or `NULL`, we need to tweak the regular data access queries in the key-value storage system. The modified operations are shown in Figure 2.

`PUT`(<k/t, v>): The `PUT` is invoked when a key-value pair, <k/t, v>, has arrived, where k represents the key, t is a version number and v is the value. In Figure 2, the black arrows represent the steps of the `PUT` operation. QinDB (1) appends the arriving key-value pair to the end of the AOFs. Then (2) k/t and $offset$ are crafted as an item and inserted into the skip list, no matter if there is a valid value field or not. In addition to these two fields, we also have an individual flag r marking if this item has been deduplicated and a flag of d representing if this key-value pair has been deleted. It worth noting that items are sorted in the skip list so that the same keys are naturally aggregated in the order of increasing version numbers.

`GET`(k/t): When a read query to k/t arrives, QinDB at first searches the k/t in the skip list. The `GET` operation follows the red arrows in Figure 2. If (1) we can find a corresponding item in the skip list and the flag r indicates that the value field exists, (2) the key-value pair can then be successfully positioned in the AOF and returned to who sent the query. If the r flag shows the value field of k/t has been deduplicated, QinDB is expected to find the value field as `NULL` in the AOF. In order to discover a valid value for the query, QinDB tracebacks to older versions until an item containing the value field has been found. The traceback is illustrated at the right bottom of Figure 2.

`DEL`(k/t): The `DEL` operation is sophisticated because a value may be referred by multiple keys and the GC is involved. In Figure 2, the dashed black arrows mean the `DEL` operation and the dashed grey arrows indicate the GC steps. In most cases, (1) the `DEL`(k/t) operation simply set the delete flag d in the main memory. When the manipulation against the skip list has completed, (2) QinDB updates the occupancy ratio of the corresponding file containing the deleted key and value, which are maintained in a GC table in the memory. The deletions actually happen in the GC process. (3) If the occupancy ratio of a file has reached a predefined threshold, QinDB invokes the GC process to recycle this file. (4) The GC process appends all valid key-value pairs and invalid key-value pairs that are referred by later version keys to current end of the AOFs, (5) updates the $offset$ fields in the skip list for corresponding items, and (6) eventually erase the target file on the disk. For the invalid key-value pairs with no referent, QinDB also removes their matching items in the skip list, which has the deletion flag set already. The difference between deleting items with and without referents are described at right top in Figure 2.

QinDB achieves a higher throughput at a cost of longer recovery time and larger storage space. When a storage node has recovered from a failure, we have to scan all AOFs for reconstruction of the memtable and the GC table. And QinDB revokes the disk space occupied by the deleted data

less often compared with the periodical GC policy. However, both costs are acceptable because: 1) The long recovery time can hardly impair the system availability as there are three replicas of the index data in Mint. The parallel requests to the replicas will hide the node recovery from front-end users. 2) In Baidu's search service, the I/O throughput, which directly impacts the user experience, is much more important than the storage space.

Block-aligned files: In addition to the write amplification at the software level, there is also write amplification caused by the SSDs. Currently the SSD can only erase the data at the block scale while supporting writes at the page scale. Figure 3 illustrates the asymmetric write and delete operations on SSDs, in which a block is 256 KB composed of 64 pages in 4 KB. This means that the valid data in a block have to be moved if we want to free other pages containing invalid data in this block. The recycling operations are completed by the GC processing on the SSD. An example of SSD GC is shown in Figure 4. In this example, when we want to free three pages containing invalid data in the block x, we have to first migrate all valid pages to another block y. Then we can free the storage space of block x entirely.

In order to circumvent the SSD GC mechanism that incurs both read and write amplification, QinDB directly invokes the native SSD programming interfaces to store and erase the AOFs in the block-aligned manner. In other words, QinDB stores files on SSD in a unit of block. In the best case, all data pages in a block can be either valid or invalid. GC only targets invalid blocks, eliminating write amplification.

3. Implementation

We have implemented two mechanisms for promptly locating bugs, errors, and failures in DirectLoad. We verify the data integrity based on checksums in Bifrost and build *gray release* for tracing potential malfunctions in a new version.

Failures in Transmission: The arrival of data at a data center does not mean the success of index delivery. The integrity of delivered data may be impaired during the network transmission due to failures of switches or other relay nodes. In order to discover the transmission errors as early as possible, every intermediate nodes in Bifrost will recalculate and compare the checksum of received index data slices. The slice checksum will be removed after all new version data are ready across the regional data centers. Another reason for failing the data transmission is that the network can not deliver the indices in time. In the case that an out-of-date data slice is found, a warning message will be generated and this may lead to a repair process.

Gray Release: A *gray release* that allows version advance at only one out of the six data centers is necessary. The gray release accepts realistic user queries and it is an opportunity of exposing unexpected malfunctions, glitches, and errors before activating the new version in all data centers. The malfunctions/glitches/errors stopping the releasing of a new version includes data inconsistency, module failures, long-tail query latency and etc. The period that the gray release lasts depends on the data type. The VIP index data are updated more frequently compared to the non-VIP data. Rolling back to the last version is the last resort if the malfunctions/glitches/errors can not be fixed in time.

The gray release occasionally leads to inconsistent searching results for users traveling across regions. A user may access the different versions of inverted index and summary index. In DirectLoad, the inconsistency of searching result is measured under 0.1% and this inconsistency rarely confuses users because of the highly overlapped content between consecutive versions.

4. Evaluation

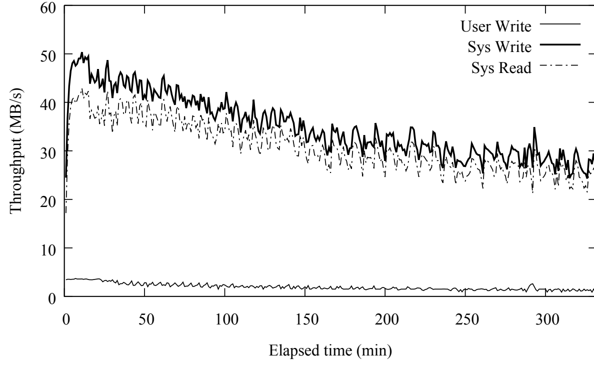
DirectLoad has been served Baidu's index update more than two years, and contributes most on index freshness enhancement. Inconsistent rate between DirectLoad indices and the expected results has been decreased from 5% to 1.2% statistically after our update system is online. The decrease of index inconsistent rate has benefited document retrieval accuracy for Baidu users.

In the evaluation process, we first prepare some micro-benchmarks and compare the performance of QinDB to the traditional LSM-Tree based engine. Then, we evaluate the updating throughput of DirectLoad over our production workload. Finally, we collect experimental data to inspect the performance variation in terms of data availability raised by our system.

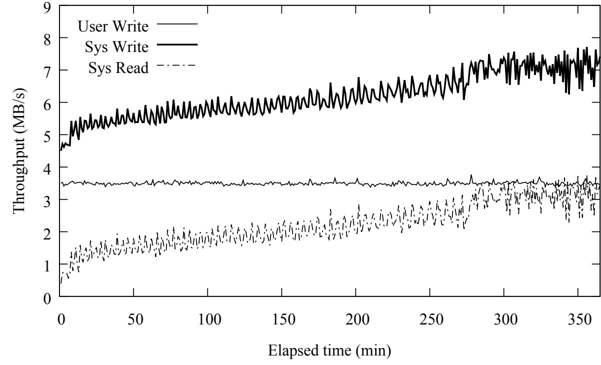
All of our experiments are carried out in dockers equipped with a 4-core 2.4 GHz Xeon processor, 4 GB of memory, and one 500G SSD disk. The dockers are mutually networked with 1 Gbps full-duplex Ethernet. The experiments measuring performance of QinDB are conducted in a single docker and we extend experiments to 200 nodes when evaluating the overall updating performance of DirectLoad. In all experiments, LevelDB 1.9.0 [2] running with the default configurations is used as the baseline.

4.1. Performance of QinDB

1) *Write Amplification*: in the first experiment we evaluate how much write amplification can be removed in QinDB. We rerun a 6-hour workload of summary index on both QinDB and LevelDB, in which 11 versions of data are updated onto the SSDs. The workload is composed of key-value pairs with 20-byte keys, and the value field is 20 KB on average. For QinDB, there are 8 write threads including 1 deletion thread and 7 insertion threads. If there are four versions of data on the disks already, the deletion thread removes the oldest version when the new version of data are inserted. LevelDB is configured with the exactly same threads. The results of this experiment are shown in Figure 5, in which the x-axes are elapsed time in minutes and the y-axes are throughput in MB/s. The *Sys Read* and *Sys Write* used in experiments are measured by the SSD firmware, while the *User Write* is from user/application

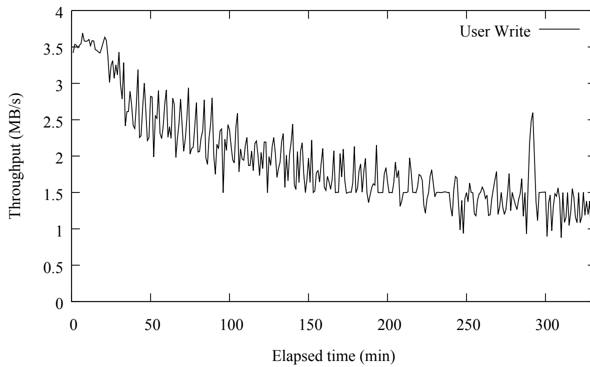


(a) LevelDB

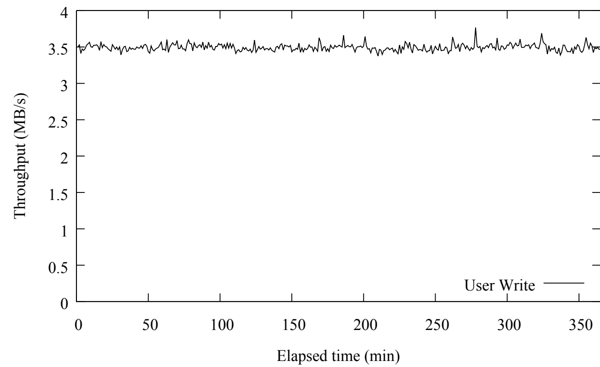


(b) QinDB

Figure 5: The write amplification comparison of LevelDB *vs.* QinDB



(a) LevelDB



(b) QinDB

Figure 6: The write throughput dynamics at the application level

view. Figure 5a shows the throughput variation of *User Write*, *Sys Write* and *Sys Read* of LevelDB while Figure 5b presents the same performance measurement of QinDB. Figure 5a shows that the User Write throughput is only 1.5 MB/s on average while the underlying Sys Write throughput ranges from 30 MB/s to 50 MB/s. The throughput difference between Sys Write and User Write reflects the level of write amplification. In this case, such 20x to 25x write amplification comes from the compaction operations of the LSM-Tree. In other words, over 90% I/O bandwidth are consumed by the LSM-tree compaction operations. On the other hand, Figure 5b presents the three throughputs of QinDB, where the User Write throughput is 3.5 MB/s on average and the Sys Write throughput is only 7.5 MB/s on average. The write amplification is much lower compared to LevelDB due to the migration of sorting operations to the main memory. There is still up to 2.5x write amplification as QinDB has to re-append valid data of deleted files in the GC process.

The detailed throughput of User write of two storage engines are plotted in Figure 6. Figure 6a and Figure 6b show that the User Write throughput dynamics of LevelDB is much higher than that of QinDB due to the frequent LSM-Tree compaction, in which the standard deviation of User Write throughput in LevelDB is 0.6616 MB/s and the same metric in QinDB is 0.0501 MB/s. The write throughput dynamics in QinDB is effectively smoothed out by the migration of sorting operations and the lazy GC policy. The high fluctuation of write throughput in an index updating system is likely to defer the arrival of index data and further impair the system availability.

2) *Storage Occupation*: We also measure the storage space usage of QinDB in the same experiment as a lazy GC policy is used in the system. For QinDB, an AOF is recycled if its occupancy ratio has lowered to 25%. Furthermore, in QinDB, the GC will be deferred if there are ongoing reads and free disk space. The experimental results are presented in Figure 7, in which the y-axis is storage space usage in GB and the x-axis is elapsed time in minutes. We observe that with the same workload, LevelDB consumes less storage space due to its more frequent compaction operations. In the first 180 minutes, QinDB uses disk space much more quickly without revoking the deleted data, but this trend slows down at the 185th minutes since the GC starts to work. At the end of the experiment, ∼80 GB disk space is used by QinDB while only ∼40 GB space is consumed by

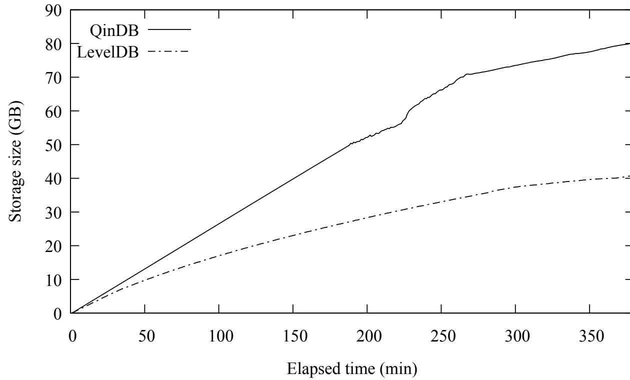


Figure 7: Storage occupation during data processing

LevelDB.

3) *Read Latency*: Next we want to understand if our design impact the read performance. We measure the read latency in our production environment that has Mint and QinDB installed. The data are collected with or without updating data streams. Both experiments last 5 hours, and when there are updating data streams, 11 versions of indices are inserted. In the experiments, the query rate of user reads are 8 million/s and the updating throughput is 5 MB/s. Both of the results are shown in Figure 8. The y-axes are read latency in microseconds and the x-axes represent three statistical points: average latency, 99th percentile latency, and 99.9th percentile latency. Different bars in the figures are the results measured in LevelDB or QinDB. In Figure 8a, QinDB has almost the same average and 99th percentile read latency as LevelDB, which are 1803 us (1846 us for LevelDB) and 3558 us (3909 us for LevelDB), repectively. QinDB experiences a much lower 99.9th percentile read latency (6574 us) than LevelDB (15081 us) because in the worse case, LevelDB has to open multiple files, i.e., searching along layers in the LSM-Tree, to locate values. In Figure 8b, LevelDB has the read latency of 2668 us on average, 12789 us at the 99th percentile, and 26458 us at the 99.9th percentile. QinDB has a similar average read latency of 2104 us, but reaches much lower 99th percentile and 99.9th percentile read latency, which are 4397 us and 13663 us, respectively. Since the long latency of a read would easily violate the Service Level Object (SLO) that all search queries should be completed in 500 ms, So QinDB is highly preferred as the storage engine in Baidu.

4.2. Online Cross-region Update

Next, we examine how much redundancy in the index updating streams can be removed and if the deduplication will impact the reliability of our index updating system.

1) *Deduplication and Upate Time*: We evaluate the overall updating performance in our production system with DirectLoad installed. The results are analyzed from a one-month long system log containing 10 versions of index data. We expect to understand the amount of redundant data can be removed and how fast the index updating can achieve. The deduplication ratio is measured as the proportion of data removed by the deduplication module before network transmission and the update time is measured as the period starting at the generation of the very first index data in a new version and ending at all data of this version are prepared in the six data centers. Since the update time is affected by a lot of factors like network congestion or unexpected glitches other than the deduplication ratio, we want to confirm that the deduplication of network transmission can contribute to the major improvement of update efficiency.

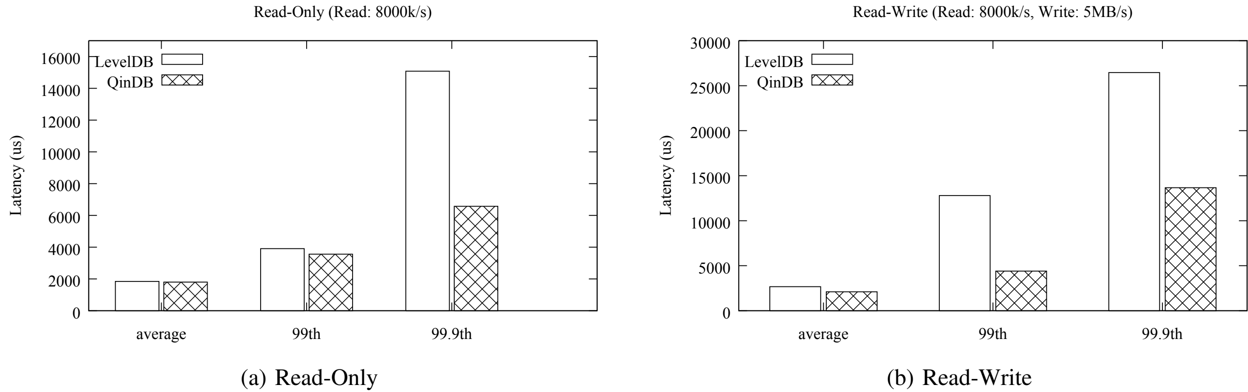
The experimental results are presented in Figure 9, in which the x-axis is the elapsed days within one-month scope, the left y-axis is the update time in minutes and the right one represents deduplication ratio in percent. We observe that in DirectLoad, the update time is mainly determined by the deduplication ratio while the fluctuations reflect the impacts from other factors. For example, in an early day of the month, as the ratio goes down to 23%, the update time increases to 130 minutes. The update time decreases to $\sim$30 minutes when the deduplication ratio has reached $\sim$80% in the middle of the month. Overall, the update time of index data is closely correlated to the decuplication ratio, and it can be reduced to as low as 30 minutes.

2) *Availability*: The *miss ratio* is denoted as the proportion of a slice of data that takes more than one hour to arrive at the destination data center, and it reflects the availability of one version of updating data. In this part, we want to check if DirectLoad will degrade the data availability.

The results are retrieved from the same one-month dataset. Figure 10a compares the updating throughput of systems with and without DirectLoad. The x-axis is elasped days within one month and the y-axis is the updating throughput in 10^4 keys/sec (kps). The results show that the updating throughput can be improved by up to 5x times in DirectLoad, which is contributed by both the deduplication and the new storage engine. Figure 10b shows the data availability of DirectLoad, where the right x-axis is the miss ratio in percentage. The results show that the miss ratio of DirectLoad is only 0.24% while the corresponding SLO in Baidu is 0.6%.

5. An Evaluation by the RUM Conjecture

The RUM Conjecture [9] considers the design of a storage system with three critical parameters: read latency (R), Update overhead, or write performance (U), in conjunction with memory and storage cost (M). Optimization in each area means to minimize the read latency, to maximize the write throughput, and to minimize the memory/storage space. The conjecture states that in a system design, optimizing any two parameters would be at a cost of the third one. For example, the designers of LSbM-tree aim to improve the read performance by minimize the LSM-tree induced buffer cache validations to gain high hit rates, and to retain the high write throughput of LSM-tree. The read



(a) Read-Only

(b) Read-Write

Figure 8: Latency under two scenarios

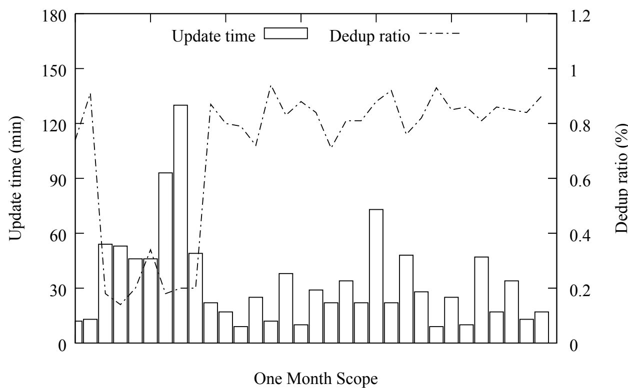


Figure 9: Dedup ratio and update time within one month

and write performance optimizations are achieved at the cost of additional space of a compaction buffer in the disk.

The storage system in DirectLoad gains its read and write performance under the RUM framework in the following way. During normal operations, each parameter is optimized or used to help others. The read performance is high with the in-memory index and fast accesses in SSD. The write throughput is also high with simple appending operations, lazy GC and block-aligned write, minimizing write amplification. The space in memory is efficiently used, but storage space is increased due to lazy GC that improves the user write and read throughput. Furthermore, once the system is crashed, we need to rebuild the index in the memory from the appended data items in the SSD storage. Both read and write activities have to be stalled during the recovering time. In practice, the recovery operations have been rarely triggered.

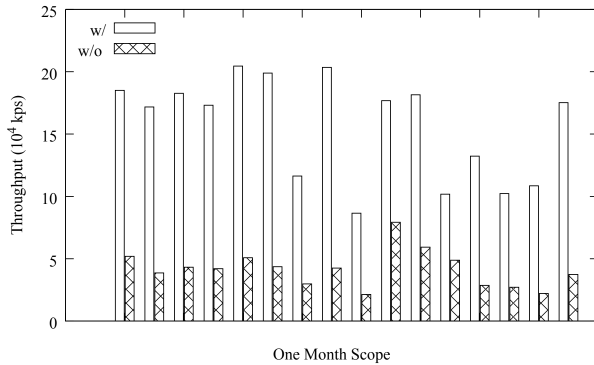
6. Related Work

We first survey various key-value stores on the SSD and the related techniques adopted for this emerging hardware. We also investigate the key-value stores that choose the LSM-Tree to improve their write performance, where lots of efforts have been made to overcome the inefficiency from the compaction operation. Building a large scale of key-value store is challenging and we will discuss work related to this topic.

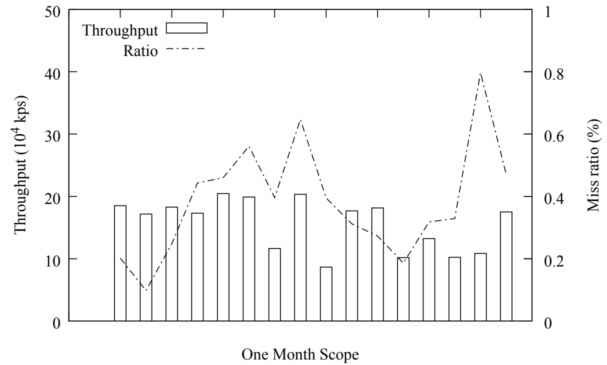
6.1. Key-value Stores on the SSD

The SSD is an emerging storage hardware and its details are introduced in [10], [11], [12]. Chen et. al [13] identifies the I/O parallelism as the top priority of performance improvement on SSDs and most schemes desgined for HDDs may not be effective any more. LOCS [14] exposes the multi-channel capacity in SSD hardware to LSM-Tree applications and achieves high I/O throughput. NVMKV [15] builds the key-value store based on the advanced capabilities of modern Flash Translation Layers (FTLs), and the performance of a `GET/PUT` operation approaches that of raw devices. Such a FTL-aware solution has been also adopted in other systems [16], [17], [18].

To improve write performance to SSD, most key-value stores [19], [20], [21], [22] append their data to a log structure while having a hash table in the main memory for indexing. FlashStore [19] uses the flash device as the cache between the main memory and the disk, in which key signatures are kept for space saving. In the main memory, a variety of cuckoo hashing is used to resolve key-collisions. SkimpyStash [20] offloads most memory pointers to the flash so that the extremely low memory footprint can be achieved, and two advanced techniques further improve the performance. FAWN [21] features the capability of fast and energy efficient data processing. SILT [22] modifies the storage layout on the flash according to data age. One of the layouts is log-structured for high write performance. CLAMs [23] implements the fast hash table operations based on the combination of DRAM and flash storage. Specifically, insertion operations are cached in the main memory and written to the flash storage in batch. All of the above systems are built with hash tables and the advanced features like range queries are not supported.



(a) The throughput improvement in DirectLoad



(b) The miss ratio of DirectLoad with one-hour delay window

Figure 10: The throughput improvement and miss ratio of DirectLoad

6.2. Key-value Stores with the LSM-Tree

A wide range of applications, e.g., Google BigTable [24] and Apache HBase [25], are built upon LSM-Tree based engines like LevelDB [2] and RocksDB [3] for high write performance to HDDs. But the write amplification introduced by compaction need to be reduced before we can maximize the write throughput on SSDs. LevelDB, the underlying storage engine of BigTable, balances the write and read throughput by ordering the multiple-way merge data-shards in the scheduling. This engine is structured into levels (6 levels by default) and the larger size of data shard ($2^{level-1}$) at a high storage level aggravates the write amplification. In contrast, RocksDB is more economic in terms of write amplification because size-based LSM-Tree compaction would invoke less redundant data reads and writes when the key ranges overlap each other.

In addition to the industrial implementations, there are other efforts improving the performance of key-value stores based on LSM-Tree. Monkey [26] explores the design space of LSM-Tree based key-value stores in terms of lookup costs and update costs. LSbM-tree [5] adds a compaction buffer to minimize the disk I/Os incurred by the compaction operations. cLSM [27] is an algorithm leveraging the power of multiprocessors to increase the throughput of LSM-Tree accesses. bLSM [28] focuses on optimizing the read performance in the LSM-Tree. VT-Tree [29] proposes to eliminate merge operations for data that have been sorted. LSM-trie [30] constructs a prefix tree for storing and compacting data in a more efficient manner so that the write amplification can be alleviated. Wisckey [6] improves the write performance by removing the colossal values, and the similar technique can be also found in Walnut [31], IndexFS [32], and Purity [33]. Kudu [34] leverages the column-based compaction to compact the high-efficient columnar data only.

6.3. Key-value Stores at Scale

The challenges also exist when building a large scale of distributed key-value store. Redis [35] and MemCache [36] are based on the hash-based access model for low-latency applications, but it suffers from the limited scalability. Particularly, a high proportion of resources are wasted in data re-sharding and copying in a large-scale system. Re-sharding also happens during the migration of data after the node failures [37]. Facebook builds their key-value store based on MemCache while also considering the ease of monitoring, debugging and operational efficiency [38]. Dynamo [39] is a large scale of key-value store in Amazon with high availability. FaRM [40] is a distributed implementation of the in-memory key-value store over Remote Direct Memory Access (RDMA) for high bandwidth and low latency. MICA [41] applies a holistic approach for fast and scalable data accesses by exploiting CPU parallelism, circumventing network stacks in kernel and optimizing memory allocation and indexing. Fastpass [42] indicates that the data center network following the conventional design principles of the Internet should be remolded to achieve high performance. f4 [43] aims at improving the storage efficiency while remaining fault tolerant for Binary Large OBjects (BLOBs).

When building key-value stores across regional data centers, the target of our system is the fast and reliable data transmission. In order to speed up the index data delivery, Peer-to-Peer (P2P) communication is an option. The P2P communication saves 50% bandwidth in our scenario [44], [45] but it is not reliable. Google MillWheel [46], Apache Spark Streaming [47], and DStream [48] are the similar transmission systems that deliver streaming data at low latency, where the reliability of update channels are one of the major concerns. Studies like [49], [50], [51], [52] prefer the efficient data transmission to the challenges of scalability and heterogeneity of the underlying architecture. The data update needs to satisfy varying networking QoS while guaranteeing the consistency across regional datacenters. The monetary costs paid for the network resources are usually the difficulty that should be first overcome when optimizing the system. Research in the field of data deduplication [50], [51] motivate our work in the effective reduction of network transmission.

7. Conclusion

In this paper, we present DirectLoad, an index updating system used in Baidu to provide the reliability and consistency management, which identifies the redundant transmission in business applications and decreases the network workload by its de-deuplication mechanism. In addition, it enhances the channel reliability by reducing traffic jam occurred in back-end processing of index storage. With production online workloads, we show the channel capacity has been dramatically increased more than 3 folds in throughput and the index updating cycle is reduced from 15 days to 3 days.

We have made a strong case for in-memory data processing by designing and implementing DirectLoad, which allows data to be accessed quickly, enabling high speed decision-making in business and web searching for billions of users of Baidu. Specifically, redundant indices are eliminated before delivering to each regional center, where indices are hashed into different storage nodes for load balancing. Fast reads are provided in each storage node by in-memory searched of keys, and loading values from SSD. High throughput writes are provided by appending operations and lazy GC in SSD. We believe our framework is applicable to other industrial applications. DirectLoad may be disruptive to conventional system design in existing memory hierarchy, where data sets are carefully stored in an indexed format in the persistent storage, such as hard disks and SSDs, and high performance relies on buffer caching in memory. In DirectLoad, we have boldly moved the entire data sets in a sorted format in the main memory, and shown its effectiveness in our production system, which can be applicable in other data center applications. We also hope our work contribute a strong case study in the system research literature.

Acknowledgments

We would like to thank the anonymous reviewers for their encouragement, constructive comments and suggestions. This work has been partially supported by the National Science Foundation under grants CCF-1513944, CCF-1629403, and IIS-1718450.

References

[1] "Baidu, Inc." http://www.baidu.com.

[2] S. Ghemawat and J. Dean, "LevelDB, a fast and lightweight key/value database library by google," https://github.com/google/leveldb.

[3] "Facebook RocksDB," http://code.google.com/apis/protocolbuffers /docs/overview.html.

[4] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[5] D. Teng, L. Guo, R. Lee, F. Chen, S. Ma, Y. Zhang, and X. Zhang, "Lsbm-tree: Re-enabling buffer caching in data management for mixed reads and writes," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 68–79.

[6] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: separating keys from values in ssd-conscious storage," *ACM Transactions on Storage (TOS)*, vol. 13, no. 1, p. 5, 2017.

[7] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang, "Mega-kv: a case for gpus to maximize the throughput of in-memory key-value stores," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1226–1237, 2015.

[8] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668–676, 1990.

[9] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan, "Designing access methods: The rum conjecture." in *EDBT*, vol. 2016, 2016, pp. 461–466.

[10] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 1. ACM, 2009, pp. 181–192.

[11] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance." in *USENIX Annual Technical Conference*, vol. 57, 2008.

[12] C. Dirik and B. Jacob, "The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 279–289.

[13] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011, pp. 266–277.

[14] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of lsm-tree based key-value store on open-channel ssd," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 16.

[15] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami, "Nvmkv: A scalable, lightweight, ftl-aware key-value store." in *USENIX Annual Technical Conference*, 2015, pp. 207–219.

[16] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda, "Beyond block i/o: Rethinking traditional storage primitives," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011, pp. 301–311.

[17] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn, "Dfs: A file system for virtualized flash storage," *ACM Transactions on Storage (TOS)*, vol. 6, no. 3, p. 14, 2010.

[18] M. Saxena, M. M. Swift, and Y. Zhang, "Flashtier: a lightweight, consistent and durable storage cache," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 267–280.

[19] B. Debnath, S. Sengupta, and J. Li, "Flashstore: high throughput persistent key-value store," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1414–1425, 2010.

[20] 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.

[21] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: A fast array of wimpy nodes," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 1–14.

[22] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "Silt: A memory-efficient, high-performance key-value store," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 1–13.

[23] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath, "Cheap and large cams for high performance data-intensive networked systems." in *NSDI*, vol. 10, 2010, pp. 29–29.

[24] 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," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[25] "Apache hbase," https://hbase.apache.org.

[26] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal navigable key-value store," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 79–94.

[27] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar, "Scaling concurrent log-structured data stores," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 32.

[28] R. Sears and R. Ramakrishnan, "blsm: a general purpose log structured merge tree," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 217–228.

[29] P. Shetty, R. P. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with vt-trees." in *FAST*, 2013, pp. 17–30.

[30] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "Lsm-trie: an lsm-tree-based ultra-large key-value store for small data," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 2015, pp. 71–82.

[31] J. Chen, C. Douglas, M. Mutsuzaki, P. Quaid, R. Ramakrishnan, S. Rao, and R. Sears, "Walnut: a unified cloud object store," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 743–754.

[32] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 237–248.

[33] J. Colgrove, J. D. Davis, J. Hayes, E. L. Miller, C. Sandvig, R. Sears, A. Tamches, N. Vachharajani, and F. Wang, "Purity: Building fast, highly-available enterprise flash storage from commodity components," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1683–1694.

[34] T. Lipcon, D. Alves, D. Burkert, J.-D. Cryans, A. Dembo, M. Percy, S. Rus, D. Wang, M. Bertozzi, C. P. McCabe *et al.*, "Kudu: storage for fast analytics on fast data," *Cloudera, inc*, vol. 28, 2015.

[35] "redis," https://redis.io/.

[36] B. Fitzpatrick and A. Vorobey, "Memcached: a distributed memory object caching system," 2011.

[37] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[38] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, "Scaling memcache at facebook." in *nsdi*, vol. 13, 2013, pp. 385–398.

[39] 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 *ACM SIGOPS operating systems review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.

[40] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "Farm: Fast remote memory," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, 2014, pp. 401–414.

[41] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage." USENIX, 2014.

[42] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized zero-queue datacenter network," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 307–318, 2015.

[43] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang *et al.*, "f4: Facebooks warm blob storage system," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 2014, pp. 383–398.

[44] A. R. Bharambe, C. Herley, and V. N. Padmanabhan, "Analyzing and improving bittorrent performance," *Microsoft Research, Microsoft Corporation One Microsoft Way Redmond, WA*, vol. 98052, pp. 2005–03, 2005.

[45] D. Qiu and R. Srikant, "Modeling and performance analysis of bittorrent-like peer-to-peer networks," in *ACM SIGCOMM computer communication review*, vol. 34, no. 4. ACM, 2004, pp. 367–378.

[46] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[47] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. Mccauley, M. Franklin, S. Shenker, and I. Stoica, "Fast and interactive analytics over hadoop data with spark," *Usenix Login*, vol. 37, no. 4, pp. 45–51, 2012.

[48] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters." *HotCloud*, vol. 12, pp. 10–10, 2012.

[49] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, "Paxos replicated state machines as the basis of a high-performance data store," in *Proc. NSDI11, USENIX Conference on Networked Systems Design and Implementation*, 2011, pp. 141–154.

[50] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips, "The bittorrent p2p file-sharing system: Measurements and analysis," in *International Workshop on Peer-to-Peer Systems*. Springer, 2005, pp. 205–216.

[51] A. Tridgell, P. Mackerras *et al.*, "The rsync algorithm," 1996.

[52] T. Suel and N. Memon, "Algorithms for delta compression and remote file synchronization," *Lossless Compression Handbook*, 2002.