Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook

Zhichao Cao, *University of Minnesota, Twin Cities, and Facebook;*
Siying Dong and Sagar Vemuri, *Facebook;*
David H.C. Du, *University of Minnesota, Twin Cities*

https://www.usenix.org/conference/fast20/presentation/cao-zhichao

This paper is included in the Proceedings of the
18th USENIX Conference on File and
Storage Technologies (FAST '20)
February 25–27, 2020 • Santa Clara, CA, USA
978-1-939133-12-0

Open access to the Proceedings of the
18th USENIX Conference on File and
Storage Technologies (FAST '20)
is sponsored by

Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook

Zhichao Cao[†‡] Siying Dong[‡] Sagar Vemuri[‡] David H.C. Du[†]

[†]University of Minnesota, Twin Cities [‡]Facebook

Abstract

Persistent key-value stores are widely used as building blocks in today's IT infrastructure for managing and storing large amounts of data. However, studies of characterizing real-world workloads for key-value stores are limited due to the lack of tracing/analyzing tools and the difficulty of collecting traces in operational environments. In this paper, we first present a detailed characterization of workloads from three typical RocksDB production use cases at Facebook: UDB (a MySQL storage layer for social graph data), ZippyDB (a distributed key-value store), and UP2X (a distributed key-value store for AI/ML services). These characterizations reveal several interesting findings: first, that the distribution of key and value sizes are highly related to the use cases/applications; second, that the accesses to key-value pairs have a good locality and follow certain special patterns; and third, that the collected performance metrics show a strong diurnal pattern in the UDB, but not the other two.

We further discover that although the widely used key-value benchmark YCSB provides various workload configurations and key-value pair access distribution models, the YCSB-triggered workloads for underlying storage systems are still not close enough to the workloads we collected due to ignorance of key-space localities. To address this issue, we propose a key-range based modeling and develop a benchmark that can better emulate the workloads of real-world key-value stores. This benchmark can synthetically generate more precise key-value queries that represent the reads and writes of key-value stores to the underlying storage system.

1 Introduction

In current IT infrastructure, persistent key-value stores (KV-stores) are widely used as storage engines to support various upper-layer applications. The high performance, flexibility, and ease of use of KV-stores have attracted more users and developers. Many existing systems and applications like file systems, object-based storage systems, SQL databases, and even AI/ML systems use KV-stores as backend storage to achieve high performance and high space efficiency [10, 16, 28, 36].

However, tuning and improving the performance of KV-stores is still challenging. First, there are very limited studies of real-world workload characterization and analysis for KV-stores, and the performance of KV-stores is highly related to the workloads generated by applications. Second, the analytic methods for characterizing KV-store workloads are different from the existing workload characterization studies for block storage or file systems. KV-stores have simple but very different interfaces and behaviors. A set of good workload collection, analysis, and characterization tools can benefit both developers and users of KV-stores by optimizing performance and developing new functions. Third, when evaluating underlying storage systems of KV-stores, it is unknown whether the workloads generated by KV-store benchmarks are representative of real-world KV-store workloads.

To address these issues, in this paper, we characterize, model, and benchmark workloads of RocksDB (a high-performance persistent KV-store) at Facebook. To our knowledge, this is the first study that characterizes persistent KV-store workloads. First, we introduce a set of tools that can be used in production to collect the KV-level query traces, replay the traces, and analyze the traces. These tools are open-sourced in RocksDB release [20] and are used within Facebook for debugging and performance tuning KV-stores. Second, to achieve a better understanding of the KV workloads and their correlations to the applications, we select three RocksDB use cases at Facebook to study: 1) UDB, 2) ZippyDB, and 3) UP2X. These three use cases are typical examples of how KV-stores are used: 1) as the storage engine of a SQL database, 2) as the storage engine of a distributed KV-store, and 3) as the persistent storage for artificial-intelligence/machine-learning (AI/ML) services.

UDB is the MySQL storage layer for social graph data at Facebook, and RocksDB is used as its backend storage engine. Social graph data is maintained in the MySQL tables, and table rows are stored as KV-pairs in RocksDB. The conversion from MySQL tables to RocksDB KV-pairs is achieved by MyRocks [19, 36]. ZippyDB is a distributed KV-store that uses RocksDB as the storage nodes to achieve data persistency and reliability. ZippyDB usually stores data like photo metadata and the metadata of objects in storage. In this paper, the workloads of ZippyDB were collected from shards that store the metadata of an object storage system at Facebook

(called *ObjStorage* in this paper). The key usually contains the metadata of an ObjStorage file or a data block, and the value stores the corresponding object address. UP2X is a special distributed KV-store based on RocksDB. UP2X stores the profile data (e.g., counters and statistics) used for the prediction and inferencing of several AI/ML services at Facebook. Therefore, the KV-pairs in UP2X are frequently updated.

Based on a set of collected workloads, we further explore the specific characteristics of KV-stores. From our analyses, we find that 1) read dominates the queries in UDB and ZippyDB, while read-modify-write (Merge) is the major query type in UP2X; 2) key sizes are usually small and have a narrow distribution due to the key composition design from upper-layer applications, and large value sizes only appear in some special cases; 3) most KV-pairs are cold (less accessed), and only a small portion of KV-pairs are frequently accessed; 4) Get, Put, and Iterator have strong key-space localities (e.g., frequently accessed KV-pairs are within relatively close locations in the key-space), and some key-ranges that are closely related to the request localities of upper-layer applications are extremely hot (frequently accessed); and 5) the accesses in UDB explicitly exhibit a diurnal pattern, unlike those in ZippyDB and UP2X, which do not show such a clear pattern.

Benchmarks are widely used to evaluate KV-store performance and to test underlying storage systems. With real-world traces, we investigate whether the existing KV benchmarks can synthetically generate real-world-like workloads with storage I/Os that display similar characteristics. YCSB [11] is one of the most widely used KV benchmarks and has become the gold standard of KV-store benchmarking. It provides different workload models, various query types, flexible configurations, and supports most of the widely used KV-stores. YCSB can help users simulate real-world workloads in a convenient way. However, we find that even though YCSB can generate workloads that have similar key-value (KV) query statistics as shown in ZippyDB workloads, the RocksDB storage I/Os can be quite different. This issue is mainly caused by the fact that the YCSB-generated workloads ignore key-space localities. In YCSB, hot KV-pairs are either randomly allocated across the whole key-space or clustered together. This results in an I/O mismatch between accessed data blocks in storage and the data blocks associated with KV queries. Without considering key-space localities, a benchmark will generate workloads that cause RocksDB to have a bigger read amplification and a smaller write amplification than those in real-world workloads.

To develop a benchmark that can more precisely emulate KV-store workloads, we propose a workload modeling method based on the hotness of key-ranges. The whole key-space is partitioned into small key-ranges, and we model the hotness of these small key-ranges. In the new benchmark, queries are assigned to key-ranges based on the distribution of key-range hotness, and hot keys are allocated closely in each key-range. In our evaluation, under the same configuration, YCSB causes at least 500% more read-bytes and delivers only 17% of the cache hits in RocksDB compared with real-world workloads. The workloads generated by our proposed new benchmark have only 43% more read-bytes and achieve about 77% of the cache hits in RocksDB, and thus are much closer to real-world workloads. Moreover, we use UDB as an example to show that the synthetic workloads generated by the new benchmark have a good fit of the distributions in key/value sizes, KV-pair accesses, and Iterator scan lengths.

This paper is organized as follows. First, we introduce RocksDB and the system background of three RocksDB use cases in Section 2. We describe our methodology and tools in Section 3. The detailed workload characteristics of the three use cases including the general query statistics, key and value sizes, and KV-pair access distributions are presented in 4, 5, and 6 respectively. In Section 7, we present the investigation results of the storage statistics of YCSB, and describe the proposed new modeling and benchmarking methods. We also compare the results of YCSB with those of the new benchmark. Related work is described in Section 8, and we conclude the paper in Section 9.

2 Background

In this section, we first briefly introduce KV-stores and RocksDB. Then, we provide background on three RocksDB use cases at Facebook, UDB, ZippyDB, and UP2X, to promote understanding of their workloads.

2.1 Key-Value Stores and RocksDB

KV-store is a type of data storage that stores and accesses data based on {key, value} pairs. A key uniquely identifies the KV-pair, and the value holds the data. KV-stores are widely used by companies as distributed hash tables (e.g., Amazon Dynamo [14]), in-memory databases (e.g., Redis [39]), and persistent storage (e.g., BigTable [8] from Google and RocksDB [15, 21] from Facebook).

RocksDB is a high-performance embedded persistent KV-store that was derived from LevelDB [23] by Facebook [15, 21] and was optimized for fast storage devices such as Solid State Drives. RocksDB is also used by many large websites, like Alibaba [44], Yahoo [37], and LinkedIn [24]. At Facebook, RocksDB is used as the storage engine for several data storage services, such as MySQL [19, 36], Laser [9], Cassandra [16], ZippyDB [1], and AI/ML platforms.

RocksDB supports KV interfaces like Get, Put, Delete, Iterator (scan), SingleDelete, DeleteRange, and Merge. Get, Put, and Delete are used to read, write, and delete a KV-pair with certain key respectively. Iterator is used to scan a set of consecutive KV-pairs beginning with a *start-key*. The scan direction can be either forward (calling Nexts) or backward (calling Prevs). SingleDelete can only be used to delete a KV-pair that has not been overwritten [22]. DeleteRange is used to delete a range of keys between [start, end) (the end-key is excluded from the deletion). RocksDB encapsulates

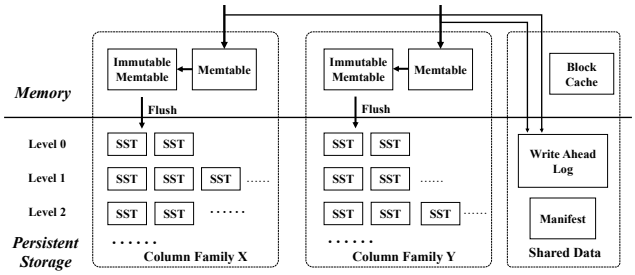


Figure 1: The basic architecture of RocksDB.

the semantics for read-modify-write into a simple abstract interface called Merge [17], which avoids the performance overhead of a random Get before every Put. Merge stores the delta of the write to RocksDB, and these deltas can be stacked or already combined. This incurs a high read overhead because a Get to one key requires finding and combining all the previously stored deltas with the same key inserted by a Merge. The combine function is defined by users as a RocksDB plugin.

RocksDB adopts a Log-Structured Merge-Tree [38] to maintain the KV-pairs in persistent storage (e.g., file systems). The basic architecture of RocksDB is shown in Figure 1. One RocksDB maintains at least one logical partition called *Column Family* (CF), which has its own in-memory write buffer (Memtable). When a Memtable is full, it is flushed to the file system and stored as a *Sorted Sequence Table* (SST) file. SST files persistently store the KV-pairs in a sorted fashion and are organized in a sequence of levels starting from Level-0. When one level reaches its limit, one SST file is selected to be merged with the SST files in the next level that have overlapping key-ranges, which is called *compaction*. Detailed information about RocksDB is described in [15, 21]

2.2 Background of Three RocksDB Use Cases

We discuss three important and large-scale production use cases of RocksDB at Facebook: 1) **UDB**; 2) **ZippyDB**; and 3) **UP2X**. Sharding is used in all three use cases to achieve load balancing. Therefore, the workloads are very similar among all shards, and we randomly select three RocksDB instances from each use case to collect the traces.

UDB: Social graph data at Facebook is persistently stored in UDB, a sharded MySQL database tier [4]. The cache read misses and all writes to social graph data are processed by UDB servers. UDB relies on the MySQL instance to handle all queries, and these queries are converted to RocksDB queries via MyRocks [19, 36]. Much of the social graph data is presented as *objects* and *associations*, and is maintained in different MySQL tables following the model introduced in [4]. RocksDB uses different Column Families (CFs) to store object- and association-related data.

There are 6 major CFs in UDB: `Object`, `Assoc`, `Assoc_count`, `Object_2ry`, `Assoc_2ry`, and `Non_SG`. `Object` stores social graph object data and `Assoc` stores social graph association data, which defines connections between two objects. `Assoc_count` stores the number of associations of each object. Association counters are always updated with new values and do not have any deletions. `Object_2ry` and `Assoc_2ry` are the CFs that maintain the secondary indexes of objects and associations, respectively. They are also used for the purpose of ETL (Extract, Transform, and Load data from databases). `Non_SG` stores data from other non-social graph related services.

Because the UDB workload is an example of KV queries converted from SQL queries, some special patterns exist. We collected the traces for 14 days. Since the workload characteristics of three UDB servers are very similar, we present only one of them. The total trace file size in this server is about 1.1 TB. For some characteristics, daily data is more important. Thus, we also analyzed the workload of the last day in the 14-day period (24-hour trace) separately.

ZippyDB: A high-performance distributed KV-store called ZippyDB was developed based on RocksDB and relies on Paxos [29] to achieve data consistency and reliability. KV-pairs are divided into shards, and each shard is supported by one RocksDB instance. One of the replicas is selected as the primary shard, and the others are secondary. The primary shard processes all the writes to a certain shard. If strong consistency is required for reads, read requests (e.g., *Get* and *Scan*) are only processed by the primary shard. One ZippyDB query is converted to a set of RocksDB queries (one or more).

Compared with the UDB use case, the upper-layer queries in ZippyDB are directly mapped to the RocksDB queries, and so the workload characteristics of ZippyDB are very different. We randomly selected three primary shards of ZippyDB and collected the traces for 24 hours. Like UDB, we present only one of them. This shard stores the metadata of ObjStorage, which is an object storage system at Facebook. In this shard, a KV-pair usually contains the metadata information for an ObjStorage file or a data block with its address information.

UP2X: Facebook uses various AI/ML services to support social networks, and a huge number of dynamically changing data sets (e.g., the statistic counters of user activities) are used for AI/ML prediction and inferencing. UP2X is a distributed KV-store that was developed specifically to store this type of data as KV-pairs. As users use Facebook services, the KV-pairs in UP2X are frequently updated, such as when counters increase. If UP2X called Get before each Put to achieve a read-modify-write operation, it would have a high overhead due to the relatively slow speed of random Gets. UP2X leverages the RocksDB Merge interface to avoid Gets during the updates.

KV-pairs in UP2X are divided into shards supported by RocksDB instances. We randomly selected three RocksDB instances from UP2X and then collected and analyzed the 24-hour traces. Note that the KV-pairs inserted by Merge are cleaned during compaction via *Compaction Filter*, which uses custom logic to delete or modify KV-pairs in the background during compaction. Therefore, a large number of KV-pairs

are removed from UP2X even though the delete operations (e.g., Delete, DeleteRange, and SingleDelete) are not used.

3 Methodology and Tool Set

To analyze and characterize RocksDB workloads from different use cases and to generate synthetic workloads, we propose and develop a set of KV-store tracing, replaying, analyzing, modeling, and benchmarking tools. These tools are already open-sourced in RocksDB release [20]. In this section, we present these tools and discuss how they are used to characterize and generate KV-store workloads.

Tracing The tracing tool collects query information at RocksDB public KV interfaces and writes to a trace file as records. It stores the following five types of information in each trace record: 1) query type, 2) CF ID, 3) key, 4) query specific data, and 5) timestamp. For *Put* and *Merge*, we store the value information in the query-specific data. For Iterator queries like *Seek* and *SeekForPrev*, the scan length (the number of *Next* or *Prev* called after *Seek* or *SeekForPrev*) is stored in the query-specific data. The timestamp is collected when RocksDB public interfaces are called with *microsecond* accuracy. In order to log the trace record of each query in a trace file, a lock is used to serialize all the queries, which will potentially incur some performance overhead. However, according to the performance monitoring statistics in production under the regular production workloads, we did not observe major throughput degradation or increased latency caused by the tracing tool.

Trace Replaying The collected trace files can be replayed through a Replayer tool implemented in *db_bench* (special plugins like MergeOperator or Comparator are required if they are used in the original RocksDB instance). The replay tool issues the queries to RocksDB based on the trace record information, and the time intervals between the queries follow the timestamps in the trace. By setting different fast forward and multithreading parameters, RocksDB can be benchmarked with workloads of different intensities. However, query order is not guaranteed with multithreading. The workloads generated by Replayer can be considered as real-world workloads.

Trace Analyzing Using collected traces for replaying has its limitations. Due to the potential performance overhead of workload tracing, it is difficult to track large-scale and long-lasting workloads. Moreover, the content of trace files is sensitive and confidential for their users/owners, so it is very hard for RocksDB users to share the traces with other RocksDB developers or developers from third-party companies (e.g., upper-layer application developers or storage vendors) for benchmarking and performance tuning. To address these limitations, we propose a way of analyzing RocksDB workloads that profiles the workloads based on information in the traces.

The trace analyzing tool reads a trace file and provides the following characteristics: 1) a detailed statistical summary of the KV-pairs in each CF, query numbers, and query types; 2) key size and value size statistics; 3) KV-pair popularity; 4) the key-space locality, which combines the accessed keys with all existing keys from the database in a sorted order; and 5) Queries Per Second (QPS) statistics.

Modeling and Benchmarking We first calculate the Pearson correlation coefficients between any two selected variables to ensure that these variables have very low correlations. In this way, each variable can be modeled separately. Then, we fit the collected workloads to different statistical models to find out which one has the lowest fitting error, which is more accurate than always fitting different workloads to the same model (like Zipfian). The proposed benchmark can then generate KV queries based on these probability models. Details are discussed in Section 7.

4 General Statistics of Workloads

In this section, we introduce the general workload statistics of each use case including query composition in each CF, KV-pair hotness distributions, and queries per second.

4.1 Query Composition

By analyzing query composition, we can figure out query intensity, the ratio of query types in different use cases, and the popularity of queries. We find that: **1) Get is the most frequently used query type in UDB and ZippyDB, while Merge dominates the queries in UP2X, and 2) query composition can be very different in different CFs**.

UDB In this UDB server, over 10.2 billion queries were called during the 14-day period, and there were about 455 million queries called during the last 24 hours. There are six CFs being used in UDB as discussed in 2.2. Although those CFs are stored in the same RocksDB database, the workloads are very different. It is difficult to analyze and model such a mixed workload without the separation of different CFs. The query composition in each CF is shown in Figure 2. Get, Put, and Iterator are three major query types in UDB, especially in `Object`, `Assoc`, and `Non_SG`. Get does not show up in the secondary indexes of objects (`Object_2ry`) and associations (`Assoc_2ry`). `Object_2ry` is built for the purpose of ETL, so Iterator is the major query type. `Assoc` mostly checks the existence of an association between two objects via Get, while the secondary index (`Assoc_2ry`) lists the objects that are associated with one target object. Since KV-pairs in `Assoc_2ry` have no repeating updates, SingleDelete is used in this CF to delete the invalid KV-pairs. In other CFs, regular Delete is called to remove the invalid KV-pairs. `Assoc_count` stores the number of associations of each object. Therefore, Get and Put are the major query types used in this CF to read and update the counters.

ZippyDB There is only one CF being used in ZippyDB. Get, Put, Delete, and Iterator_seek (forward Iterator) are the four query types that are used. Over the 24-hour period, there were about 420 million queries called in this shard. The ratios of each query type are: 78% Get, 13% Put, 6% Delete, and 3%

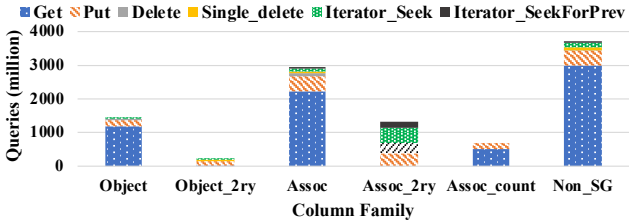
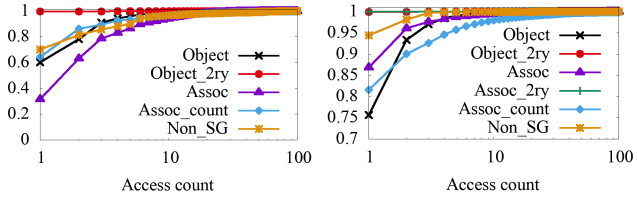


Figure 2: Distribution of different query types in 14 days.



(a) The KV-pair access count CDF by Get (b) The KV-pair access count CDF by Put

Figure 3: The KV-pair access count distribution queried by Get and Put in each CF during 24 hours.

Iterator, respectively. Get is the major query type in ZippyDB, which aligns with the read-intensive workload of ObjStorage.

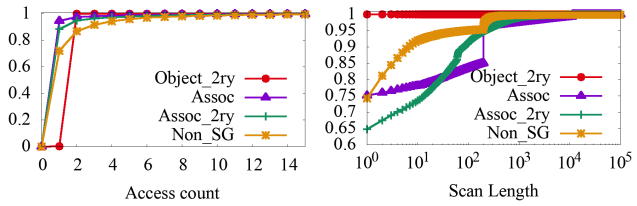
UP2X Over the 24-hour period, the RocksDB instance received 111 million queries. Among them, about 92.53% of the queries are Merge, 7.46% of them are Get, and fewer than 0.01% of the queries are Put. The query composition is very different from the UDB and ZippyDB use cases, which are read dominated. About 4.71 million KV-pairs were accessed by Merge, 0.47 million by Get, and 282 by Put. Read-and-modify (Merge) is the major workload pattern in UP2X.

4.2 KV-Pair Hotness Distribution

To understand the hotness of KV-pairs in each use case, we count how many times each KV-pair was accessed during the 24-hour tracing period and show them in cumulative distribution function (CDF) figures. The X-axis is the access count, and the Y-axis is the cumulative ratio between 0 and 1. We find that **in UDB and ZippyDB, most KV-pairs are cold**.

UDB We plot out the KV-pair access count CDFs for Get and Put. For Iterator, we show the start-key access count distribution and the scan length distribution. The CDFs of Get and Put are shown in Figure 3. Looking at Figure 3(a), more than 70% of the KV-pairs in `Assoc` are Get requests that occurred at least 2 times. In contrast, this ratio in other CFs is lower than 40%. It indicates that read misses of `Assoc` happen more frequently than the others. As shown in 3(b), in all CFs, more than 75% of the KV-pairs are Put only one time and fewer than 2% of the KV-pairs are Put more than 10 times. The majority of the KV-pairs are rarely updated.

We plot out the access count CDF of the start-keys of Iterators over the 24-hour period, as shown in Figure 4(a). Most of the start-keys are used only once, which shows a low access locality. Fewer than 1% of the start-keys are used multiple times



(a) The Iterator start-key access count CDF distribution (b) The Iterator scan length CDF distribution

Figure 4: The Iterator scan length and start-key access count CDF of four CFs during 24 hours.

by Iterators. The scan length of more than 60% of the Iterators is only 1 across all CFs, as shown in Figure 4(b). About 20% of the Iterators in `Assoc` scan more than 100 consecutive keys, while the ratios for `Assoc_2ry` and `Non_SG` are about 10% and 5%, respectively. A very large scan length (higher than 10,000) is very rare, but we can still find some examples of this type in `Non_SG` and `Assoc`. The configured range query limit in MySQL creates some special scan lengths. For example, there is a jump at 200 in both `Assoc` and `Non_SG`.

We also count the number of unique keys being accessed in different time periods. As shown in Table 1, during the last 24 hours, fewer than 3% of the keys were accessed. During the 14-day period, the ratio is still lower than 15% for all CFs. In general, most of the keys in RocksDB are “cold" in this use case. On one hand, most read requests are responded to by the upper cache tiers [5, 7]. Only the read misses will trigger queries to RocksDB. On the other hand, social media data has a strong temporal locality. People are likely accessing the most recently posted content on Facebook.

ZippyDB The average access counts per accessed KV-pair of the four query types (Get, Put, Delete, and Iterator_seek) are: 15.2, 1.7, 1, and 10.9, respectively. Read queries (Get and Iterator_seek) show very good locality, while the majority of the KV-pairs are only Put and Deleted once in the last 24-hour period. The access count distribution is shown in Figure 5. For about 80% of the KV-pairs, Get requests only occur once, and their access counts show a long tail distribution. This indicates that a very small portion of KV-pairs have very large read counts over the 24-hour period. About 1% of the KV-pairs show more than 100 Get requests, and the Gets to these KV-pairs are about 50% of the total Gets that show strong localities. In contrast, about 73% of the KV-pairs are Put only once, and fewer than 0.001% of the KV-pairs are Put more than 10 times. Put does not have as clear a locality as Get does. The CDF of Iterator_seek start-key access counts has a special distribution that we can observe very clearly through the 4 “steps" in the figure. About 55% of the KV-pairs are used as the start-key of Iterator_seek 1 time, 6% of the KV-pairs 11 times, 11% of the KV-pairs 12 times, 5% of the KV-pairs 13 times, 10% of the KV-pairs 23 times, and 10% of the KV-pairs 46 times. The special access count distribution of start-keys is caused by the metadata scanning requests in ObjStorage. For

Table 1: The ratios of KV-pairs among all existing KV-pairs being accessed during different time periods in UDB

CF name	Object	Object_2ry	Assoc	Assoc_2ry	Assoc_count	Non_SG
24 hours	2.72%	0.62%	1.55%	1.75%	0.77%	1.38%
14 days	14.14%	6.10%	13.74%	10.37%	14.05%	11.29%

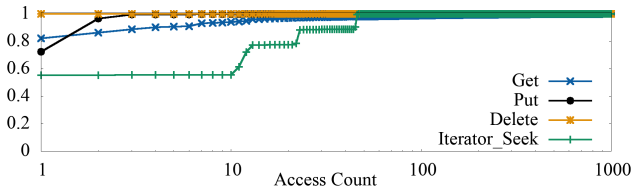


Figure 5: The KV-pair access count distribution of ZippyDB.

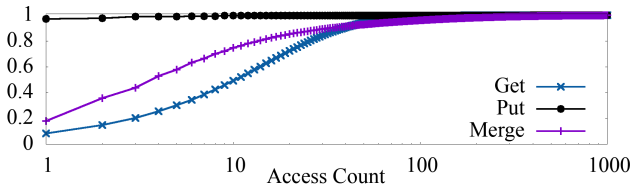


Figure 6: The access count distribution of UP2X.

example, if one KV-pair stores the metadata of the first block of a file, it will always be used as the start-key of Iterator_seek when the whole file is requested.

UP2X The CDF distribution of KV-pair access counts is shown in Figure 6. Merge and Get have wide distributions of access counts. If we define a KV-pair accessed 10 times or more during the 24-hour period as a hot KV-pair, about 50% of the KV-pairs accessed by Get and 25% of the KV-pairs accessed by Merge are hot. On the other hand, the ratio of very hot KV-pairs (accessed 100 times or more in the 24-hour period) for Merge is 4%, which is much higher than that of Get (fewer than 1%). Both Merge and Get have a very long tail distribution, as shown in the figure.

4.3 QPS (Queries Per Second)

The QPS metric shows the intensiveness of the workload variation over time. **The QPS of some CFs in UDB have strong diurnal patterns, while we can observe only slight QPS variations during day and night time in ZippyDB and UP2X. The daily QPS variations are related to social network behaviors.**

UDB The QPS of UDB is shown in Figure 7. Some CFs (e.g., `Assoc` and `Non_SG`) and some query types (e.g., Get and Put) have strong diurnal patterns due to the behaviors of Facebook users around the world. As shown in Figure 7(a), the QPS for either Get or Put usually increases from about 8:00 PST and reaches a peak at about 17:00 PST. Then, the QPS quickly drops and reaches its nadir at about 23:00 PST. The QPS of Delete, SingleDelete, and Iterator shows variations, but it is hard to observe any diurnal patterns. These queries are triggered by Facebook internal services, which have low correlation with user behaviors. The QPS of six CFs are shown in Figure 7(b). `Assoc` and `Non_SG` have a strong diurnal variation, but the QPS of `Non_SG` is spikier. Since ETL requests are not triggered by Facebook users, the QPS of `Object_2ry` is spiky and we cannot find any clear patterns.

ZippyDB The QPS of ZippyDB is different from that of UDB. The QPS of ZippyDB varies over the 24-hour period, but we do not find a diurnal variation pattern, especially for Put, Delete, and Iterator_Seek. Since ObjStorage is an object stored at Facebook, object read is related to social network behaviors. Therefore, the QPS of Get is relatively lower at night and higher during the day (based on Pacific Standard Time). Because range queries (Iterator_Seek) are usually not triggered by Facebook users, the QPS for this query type is stable and is between 100 and 120 most of the time.

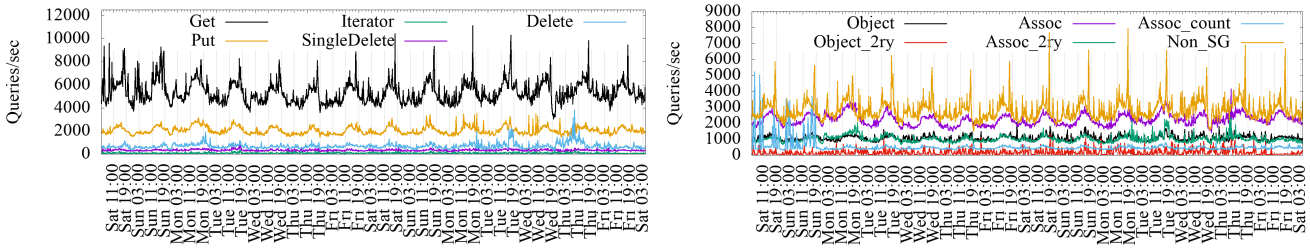
UP2X The QPS of either Get or Put in UP2X does not have a strong diurnal variation pattern. However, the usage of Merge is closely related to the behavior of Facebook users, such as looking at posts, likes, and other actions. Therefore, the QPS of Merge is relatively lower at night (about 1000) and higher during the day (about 1500).

5 Key and Value Sizes

Key size and value size are important factors in understanding the workloads of KV-stores. They are closely related to performance and storage space efficiency. The average (AVG) and standard deviation (SD) of key and value sizes are shown in Table 2, and the CDFs of key and value sizes are shown in Figure 8. In general, **key sizes are usually small and have a narrow distribution, and value sizes are closely related to the types of data. The standard deviation of key sizes is relatively small, while the standard deviation of value size is large. The average value size of UDB is larger than the other two**.

UDB The average key size is between 16 and 30 bytes except for `Assoc_2ry`, which has an average key size of 64 bytes. The keys in `Assoc_2ry` consist of the 4-byte MySQL table index, two object IDs, the object type, and other information. Therefore, the key size of `Assoc_2ry` is usually larger than 50 bytes and has a long tail distribution as shown in Figure 8(a). For other CFs, the keys are composed of the 4-byte MySQL table index as the prefix, and 10 to 30 bytes of primary or secondary keys like object IDs. Thus, the keys show a narrow distribution. Note that the key sizes of a very small number of KV-pairs are larger than 1 KB, which is not shown in the key size CDF due to the X-axis scale limit.

The value size distribution is shown in Figure 8(b). `Object`



(a) Overall QPS for each query type at different dates and times in a 14-day time span

(b) Overall QPS of each CF at different dates and times in a 14-day time span

Figure 7: The QPS variation at different dates and times in a 14-day time span.

Table 2: The average key size (AVG-K), the standard deviation of key size (SD-K), the average value size (AVG-V), and the standard deviation of value size (SD-V) of UDB, ZippyDB, and UP2X (in bytes)

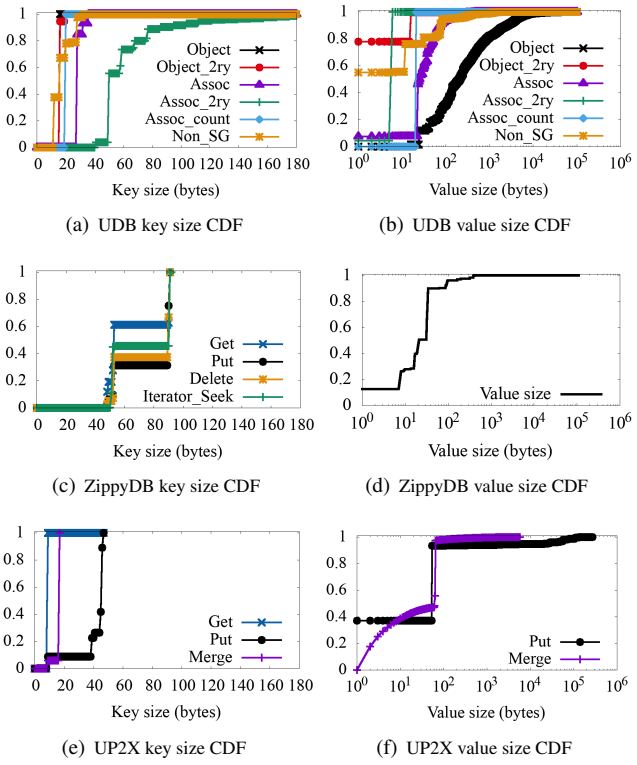
	AVG-K	SD-K	AVG-V	SD-V
UDB	27.1	2.6	126.7	22.1
ZippyDB	47.9	3.7	42.9	26.1
UP2X	10.45	1.4	46.8	11.6

and `Assoc` have a long tail distribution. The value sizes of `Object` vary from 16 bytes to 10 KB, and more than 20% of the value sizes are larger than 1 KB. The average value size of KV-pairs in `Object` is about 1 KB and the median is about 235B, which is much larger than those in other CFs. User data, like the metadata of photos, comments, and other posted data, is stored in this CF, which leads to a large value size. In `Assoc`, the value sizes are relatively small (the average is about 50 bytes) and vary from 10 bytes to 200 bytes.

A very special case is `Assoc_count`, whose key size and value size are exactly 20 bytes. According to the design of this CF, the key is 20 bytes (*bigint* association ID) and is composed of a 10-byte counter and 10 bytes of metadata. Since all the information used in secondary index CFs (`Assoc_2ry` and `Object_2ry`) is stored in its key, the value does not contain any meaningful data. Therefore, the average value size is less than 8 bytes and there are only three possible value sizes in the distribution (1 byte, 6 bytes, or 16 bytes) as shown in Figure 8(b). For CFs with large value sizes like `Object`, optimizations like separating key and value [32] can effectively improve performance.

ZippyDB Since a key in ZippyDB is composed of ObjStorage metadata, the key sizes are relatively large. The CDF of the key sizes is shown in Figure 8(c). We can find several "steps" in the CDF. **Nearly all of the key sizes are in the two size ranges: [48, 53] and [90, 91].** The ratio of KV-pairs in these two key size ranges are different for different query types. For example, about 60% of the key sizes of Get are in the [48, 53] range, while the ratio for Put is about 31%.

The value sizes are collected from Put queries. As shown in Figure 8(d), the value size distribution has a very long tail: about 1% of the value sizes are larger than 400 bytes, and



(a) UDB key size CDF

(b) UDB value size CDF

(c) ZippyDB key size CDF

(d) ZippyDB value size CDF

(e) UP2X key size CDF

(f) UP2X value size CDF

Figure 8: The key and value size distributions of UDB, ZippyDB, and UP2X.

about 0.05% of the value sizes are over 1 KB. Some of the value sizes are even larger than 100 KB. However, most of the KV-pairs have a small value. More than 90% of the value sizes are smaller than 34 bytes, which is even smaller than the key sizes.

UP2X The key sizes do not have a wide distribution, as shown in Figure 8(e). **More than 99.99% of the KV-pairs accessed by Get have a key size of 9 bytes**. About 6% of the KV-pairs inserted by Merge have a key size of 9 bytes, and 94% are 17 bytes. The 17-byte KV-pairs are all cleaned during compaction, and they are never read by upper-layer applications through Get. Put is rarely used in UP2X. Among the 282 KV-pairs inserted by Put, about 8.9% of the key sizes

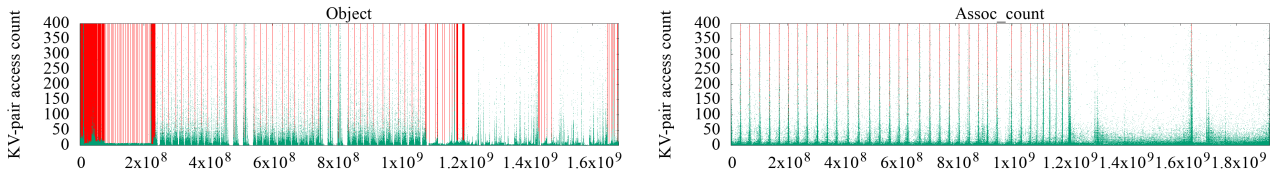


Figure 9: The heat-map of Get in `Object` and `Assoc_count` during a 24-hour period. The X-axis represents the key-ID of keys in the whole key-space, and the Y-axis represents the KV-pair access counts. The red vertical lines are the MySQL table boundaries.

are smaller than 10 bytes, and 47% of them are 46 bytes.

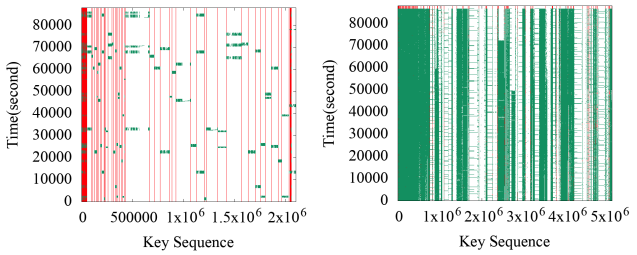
The value size distribution is shown in Figure 8(f). The value sizes of some KV-pairs inserted by Put are extremely large. The average is about 3.6 KB, and about 3% of the KV-pairs are over 100 KB. The value sizes of KV-pairs inserted by Merge have a special distribution. About 40% of the values are smaller than 10 bytes, and about 52% of the values are exactly 64 bytes. A large portion of the updates in UP2X are the counters and other structured information. Thus, the value sizes of those KV-pairs are fixed to 64 bytes.

6 Key-Space and Temporal Patterns

KV-pairs in RocksDB are sorted and stored in SST files. In order to understand and visualize the key-space localities, we sort all the existing keys in the same order as they are stored in RocksDB and plot out the access count of each KV-pair, which is called the *heat-map* of the whole key-space. Each existing key is assigned a unique integer as its *key-ID*, based on its sorting order and starting from 0. We refer to these *key-IDs* as the *key sequence*.

The KV-pair accesses show some special temporal patterns. For example, some KV-pairs are intensively accessed during a short period of time. In order to understand the correlation between temporal patterns and key-space locality, we use a time series sequence to visualize these patterns. We sort the keys in ascending order and assign them with key-IDs as previously discussed, and this *key sequence* is used as the X-axis. The Y-axis shows the time when a query is called. To simplify the Y-axis value, we shift the timestamp of each query to be relative to the tracing start time. Each dot in the time series figure represents a request to a certain key at that time. In the UDB use case, the first 4 bytes of a key are the MySQL table index number due to the key composition of MyRocks. We separate the key-space into different key-ranges that belong to different tables by red vertical lines.

The heat-maps of the three use cases show a strong key-space locality. Hot KV-pairs are closely located in the key-space. The time series figures of Delete and SingleDelete for UDB and Merge for UP2X show strong temporal locality. For some query types, KV-pairs in some key-ranges are intensively accessed during a short period of time.



(a) The time series of Delete queries in `Object`

(b) The time series of SingleDelete queries in `Assoc_2ry`

Figure 10: The time series figure of a 24-hour trace.

UDB We use the heat-map of Get in `Object` and `Assoc_count` over a 24-hour period as an example to show the key-space localities. As shown in Figure 9, hot KV-pairs (with high access counts) are usually located in a small key-range and are close to each other. That is, they show a strong key-space locality (indicated by the dense green areas). **Some MySQL tables (the key-ranges between the red vertical lines) are extremely hot (e.g., the green dense area in `Object`), while other tables have no KV-pair accesses**. One interesting characteristic is that the KV-pairs with high access counts in `Assoc_count` are skewed toward the end of the table. In social graphs, new objects are assigned with relatively larger IDs, and new associations are frequently added to the new objects. Therefore, new KV-pairs in `Assoc_count` are hot and are usually at the end of the MySQL table. Moreover, the heat-maps of Get and Put are similar. Usually, the keys with the most Get queries are the ones with the most Put queries.

Most KV-pairs are deleted only once, and they are unlikely to be reinserted. Therefore, there are no hot KV-pairs in Delete and SingleDelete queries. However, they show some special patterns. For example, some nearby KV-pairs are deleted together in a short period of time as shown in Figure 10.

In Figure 10(a), the deleted KV-pairs in the same table for `Object` are removed together in a short period of time (indicated by green dots with close Y values). After that, deletions will not happen for a long period of time. Similar patterns also appear in the SingleDelete time series for `Assoc_2ry`, as shown in Figure 10(b). In some MySQL tables, SingleDelete is intensively called in several short time intervals to remove

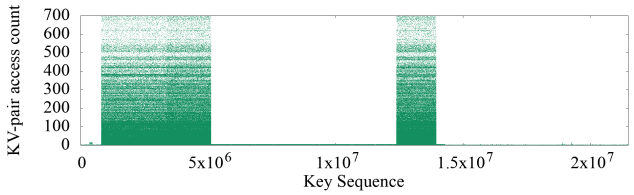


Figure 11: Heat-map of KV-pairs accessed by Get in ZippyDB.

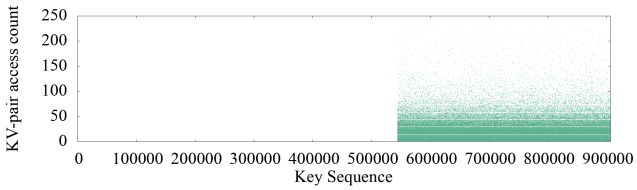


Figure 12: Heat-map of KV-pairs accessed by Get in UP2X.

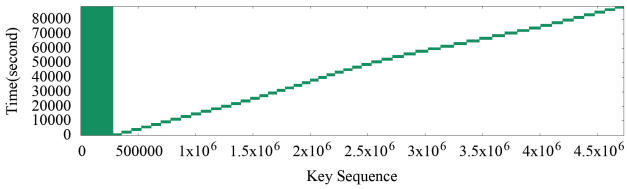


Figure 13: The time series of Merge in UP2X.

KV-pairs in the same table. Between any two sets of intensive deletions, SingleDelete is never called, which causes the "green blocks" in the time series figures.

In general, KV-pairs are not randomly accessed in the whole key-space. The majority of KV-pairs are not accessed or have low access counts. Only a small portion of KV-pairs are extremely hot. These patterns appear in the whole key-space and also occur in different key-ranges. KV-pairs belonging to the same MySQL table are physically stored together. Some SST files at different levels or data blocks in the same SST file are extremely hot. Thus, the compaction and cache mechanisms can be optimized accordingly.

ZippyDB The heat-map of Get in ZippyDB shows a very good key-space locality. For example, as shown in Figure 11, **the KV-pairs accessed by Get have high access counts and are concentrated in several key-ranges (e.g., between** 1×10^6 **and** 5×10^6**)**. Hot KV-pairs are not randomly distributed: instead, these KV-pairs are concentrated in several small key-ranges. The hotness of these key-ranges is closely related to cache efficiency and generated storage I/Os. The better a key-space locality is, the higher the RocksDB block cache hit ratio will be. Data blocks that are associated with hot key-ranges will most likely be cached in the RocksDB block cache. These data blocks are actually cold from a storage point of view. With a good key-space locality, the number of data block reads from SST files will be much lower than a random distribution. A similar locality is also found in the Put and Iterator_seek heat-maps. Since all the KV-pairs are deleted once, we did not observe any key-space locality for Delete. In general, the ZippyDB workload is read-intensive and has very good key-space locality.

UP2X If we look at the heat-map of all KV-pairs accessed by Get as shown in Figure 12, **we can find a clear boundary between hot and cold KV-pairs**. Note that the whole key-space was collected after the tracing was completed. In the heat-map, the KV-pairs from 0 to about 550,000 are never accessed by Gets, but the KV-pairs from 550,000 to 900,000 are frequently accessed. A similar locality is also shown in the heat-map of Merge. While KV-pairs from 0 to about 550,000 are sometimes accessed by Merge, their average access counts are much lower than those of the KV-pairs from 550,000 to 900,000. This special locality might be caused by a unique behavior of AI/ML services and their data update patterns.

The UP2X use case shows a very strong key-space locality and temporal locality in Merge. However, about 90% of the KV-pairs inserted by Merge are actually cleaned during compaction. Since the key-space heat-map does not show the existence of KV-pairs cleaned by compactions, we plot out the time series sequence for Merge, which can indicate Merge accesses of all KV-pairs. As shown in Figure 13, KV-pairs between 0 and 250,000 are frequently accessed during the 24-hour period. These are KV-pairs between 0 and 900,000 in the whole key-space. The KV-pairs between 250,000 and 4,700,000 show very special key-space and temporal localities. The green blocks indicate that **a small range of KV-pairs are intensively called by Merge during half an hour**. After that, a new set of KV-pairs (with incrementally composed keys) are intensively accessed by Merge during the next half an hour. These KV-pairs are cleaned during compactions. Get and Put do not have similar temporal and key-space localities.

7 Modeling and Benchmarking

After understanding the characteristics of some real-world workloads, we further investigate whether we can use existing benchmarks to model and generate KV workloads that are close to these realistic workloads. We do not consider deletions in our current models.

7.1 How Good Are the Existing Benchmarks?

Several studies [6, 26, 47] use YCSB/db_bench + LevelDB/RocksDB to benchmark the storage performance of KV-stores. Researchers usually consider the workloads generated by YCSB to be close to real-world workloads. YCSB can generate queries that have similar statistics for a given query type ratio, KV-pair hotness distribution, and value size distribution as those in realistic workloads. However, it is unclear whether their generated workloads can match the I/Os for underlying storage systems in realistic workloads.

To investigate this, we focus on storage I/O statistics such as *block reads*, *block cache hits*, *read-bytes*, and *write-bytes* collected by *perf_stat* and *io_stat* in RocksDB. To exclude other factors that may influence the storage I/Os, we replay

the trace and collect the statistics in a clean server. The benchmarks are also evaluated in the same server to ensure the same setup. To ensure that the RocksDB storage I/Os generated during the replay are the same as those in production, we replay the trace on a snapshot of the same RocksDB in which we collected the traces. The snapshot was made at the time when we started tracing. YCSB is a benchmark for NoSQL applications and ZippyDB is a typical distributed KV-store. Therefore, the workloads generated by YCSB are expected to be close to the workloads of ZippyDB, and we use ZippyDB as an example to investigate. Due to special plugin requirements and the workload complexities of UDB and UP2X, we did not analyze storage statistics for those two use cases.

Before we run YCSB, we set the YCSB parameters of `workloada` and `workloadb` to fit ZippyDB workloads as much as possible. That is, we use the same cache size, ensure that the request distribution and scan length follows Zipfian, set the fieldlength as the average value size, and use the same Get/Put/Scan ratios as those shown in Section 4. Since we cannot configure the compression ratio in YCSB to make it the same as ZippyDB, we use the default configuration in YCSB. We normalize the results of the RocksDB storage statistics based on those from the trace replay.

The number of block reads from YCSB is at least 7.7x that of the replay results, and the amount of read-bytes is about 6.2x. The results show an extremely high read amplification. Although the collected amount of write-bytes from YCSB is about 0.74x that of the replay, the actual amount of write-bytes is much lower if we assume YCSB achieves the same compression ratio as ZippyDB (i.e., if the YCSB compression ratio is 4.5, the amount of write-bytes is about 0.41x that of the replay). Moreover, the number of block cache hits is only about 0.17x that of the replay results. This evaluation shows that, even though the overall query statistics (e.g., query number, average value size, and KV-pair access distribution) generated by YCSB are close to those of ZippyDB workloads, the RocksDB storage I/O statistics are actually quite different. db_bench has a similar situation.

Therefore, using the benchmarking results of YCSB as guidance for production might cause some misleading results. For example, the read performance of RocksDB under a production workload will be higher than what we tested using YCSB. The workload of YCSB can easily saturate the storage bandwidth limit due to its extremely high read amplification. Also, the write amplification estimated from the YCSB benchmarking results are lower than in real production. The write performance can be overestimated and might also lead to incorrect SSD lifetime estimates.

With detailed analyses, we find that the main factor that causes this serious read amplification and fewer storage writes is the ignorance of key-space locality. RocksDB reads data blocks (e.g., 16 KB) instead of a KV-pair from storage to memory when it encounters a cache miss. In YCSB, even though the overall KV-pair hotness follows the real-world workload distribution, the hot KV-pairs are actually randomly distributed in the whole key-space. The queries to these hot KV-pairs make a large number of data blocks hot. Due to the cache space limit, a large number of hot data blocks that consist of the requested KV-pairs will not be cached, which triggers an extremely large number of block reads. In contrast, in ZippyDB, hot KV-pairs only appear in some key-ranges, so the number of hot data blocks is much smaller. Similarly, a random distribution of hot KV-pairs causes more updated KV-pairs to be garbage collected in the newer levels during compactions. Therefore, old versions of cold KV-pairs that are being updated are removed earlier in the newer levels, which leads to fewer writes when compacting older levels. In contrast, if only some key-ranges are frequently updated, old versions of cold KV-pairs are continuously compacted to the older levels until they are merged with their updates during compactions. This causes more data to be written during compactions.

7.2 Key-Range Based Modeling

Unlike workloads generated by YCSB, real-world workloads show strong key-space localities according to the workload characteristics presented in Sections 6. Hot KV-pairs are usually concentrated in several key-ranges. Therefore, to better emulate a real-world workload, we propose a key-range based model. The whole key-space is partitioned into several smaller key-ranges. Instead of only modeling the KV-pair accesses based on the whole key-space statistics, we focus on the hotness of those key-ranges.

How to determine the key-range size (the number of KV-pairs in the key-range) is a major challenge of key-range based modeling. If the key-range is extremely large, the hot KV-pairs are still scattered across a very big range. The accesses to these KV-pairs may still trigger a large number of data block reads. If the key-range is very small (e.g., a small number of KV-pairs per range), hot KV-pairs are actually located in different key-ranges, which regresses to the same limitations as a model that does not consider key-ranges. Based on our investigation, when the key-range size is close to the average number of KV-pairs in an SST file, it can preserve the locality in both the data block level and SST level. Therefore, we use average number of KV-pairs per SST file as key-range size.

We first fit the distributions of key sizes, value sizes, and QPS to different mathematical models (e.g., Power, Exponential, Polynomial, Webull, Pareto, and Sine) and select the model that has the minimal fit standard error (FSE). This is also called the root mean squared error. For example, for a collected workload of ZippyDB, the key size is fixed at either 48 or 90 bytes, the value sizes follow a Generalized Pareto Distribution [25], and QPS can be better fit to Cosine or Sine in a 24-hour period with very small amplitude.

Then, based on the KV-pair access counts and their sequence in the whole key-space, the average accesses per KV-pair of each key-range is calculated and fit to the distribution

model (e.g., power distribution). This way, when one query is generated, we can calculate the probability of each key-range responding to this query. Inside each key range, we let the KV-pair access count distribution follow the distribution of the whole key-space. This ensures that the distribution of the overall KV-pair access counts satisfies that of a real-world workload. Also, we make sure that hot KV-pairs are allocated closely together. Hot and cold key-ranges can be randomly assigned to the whole key-space, since the locations of key-ranges have low influence on the workload locality.

Based on these models, we further develop a new benchmark using db_bench. When running the benchmark, the QPS model controls the time intervals between two consecutive queries. When a query is issued, the query type is determined by the probability of each query type calculated from the collected workload. Then, the key size and value size are determined by the probability function from the fitted models. Next, based on the access probability of each key-range, we choose one key-range to respond to this query. Finally, according to the distribution of KV-pair access counts, one KV-pair in this key range is selected, and its key is used to compose the query. In this way, the KV queries are generated by the benchmark and follow the expected statistical models. At the same time, it better preserves key-space locality.

7.3 Comparison of Benchmarking Results

We fit the ZippyDB workload to the proposed model (Delete is excluded) and build a new benchmark called Prefix_dist [20]. To evaluate the effectiveness of key-range-based modeling, we also implement three other benchmarks with different KV-pair allocations: 1) Prefix_random models the key-range hotness, but randomly distributes the hot and cold KV-pairs in each key-range; 2) similar to YCSB, All_random follows the distribution of KV-pair access counts, but randomly distributes the KV-pairs across the whole key-space; and 3) All_dist puts the hot keys together in the whole key-space instead of using a random distribution. All four benchmarks achieve a similar compression ratio as that of ZippyDB.

Similar to the process described in Section 7.1, we configure YCSB *workloada* and *workloadb* to fit the ZippyDB workload as closely as possible. We run YCSB with the following 4 different request distributions: 1) uniform (YCSB_uniform), 2) Zipfian (YCSB_zipfian), 3) hotspot (YCSB_hotspot), and 4) exponential (YCSB_exp). We use the same pre-loaded database (with 50 million randomly inserted KV-pairs that have the same average key and value sizes as those of a real-world workload) for the 8 benchmarks. The RocksDB cache size is configured with the same value as the production setup. We run each test 3 times (the following discussion uses average value) and normalize the results based on that of replay.

Figure 14 compares the I/O statistics of the 8 benchmarks. The total number of block reads and the amount of read-bytes by YCSB_zipfian workloads are at least 500% higher than those of the original replay results. Even worse, the number of block reads and the amount of read-bytes of the other three YCSB benchmarking results are even higher, at 1000% or more compared with the replay results. In contrast, the amount of read-bytes of Prefix_dist are only 40% higher, and are the closest to the original replay results. If we compare the 4 benchmarks we implemented, we can conclude that Prefix_dist can better emulate the number of storage reads by considering key-space localities. All_dist and Prefix_random reduce the number of extra reads by gathering the hot KV-pairs in different granularities (whole key-space level vs. key-range level). Note that if YCSB achieves a similar compression ratio, the RocksDB storage I/Os can be about 35-40% lower. However, this is still much worse than the storage I/Os of All_dist, Prefix_random, and Prefix_dist.

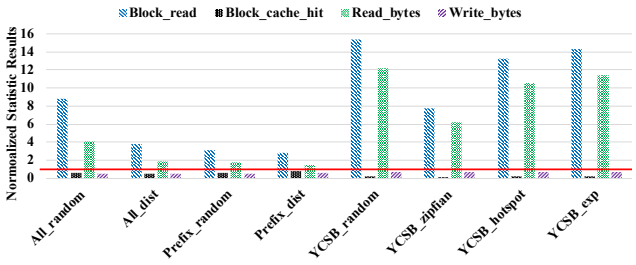


Figure 14: The normalized block read, block cache hit, read-bytes, and write-bytes of benchmarks based on that of the replay. We collected statistics from ZippyDB trace replay results and normalized the statistics from 8 benchmarks. The red line indicates the normalized replay results at 1. The closer the results are to the red line, the better.

If the same compression ratio is applied, the actual amount of write-bytes by YCSB should be less than 50% of the original replay. Prefix_dist achieves about 60% write-bytes of the original replay. Actually, the mismatch between key/value sizes and KV-pair hotness causes fewer write-bytes compared with the original replay results. In general, YCSB can be further improved by: 1) adding a key-range based distribution model as an option to generate the keys, 2) providing throughput control to simulate the QPS variation, 3) providing key and value size distribution models, and 4) adding the ability to simulate different compression ratios.

7.4 Verification of Benchmarking Statistics

We select the `Assoc` workload from UDB as another example to verify whether our benchmark can achieve KV query statistics that are very similar to those of real-world workloads. Since 90% of keys are 28 bytes and 10% of keys are 32 bytes in `Assoc`, we can use these two fixed key sizes. We find that Generalized Pareto Distribution [25] best fits the value sizes and Iterator scan length. The average KV-pair access count of key-ranges can be better fit in a two-term power model [33, 34], and the distribution of KV-pair access counts follows a power-law that can be fit to the simple power model [33, 34]. As we discussed in Section 4.3, because the QPS variation has a strong diurnal pattern, it can be better fit

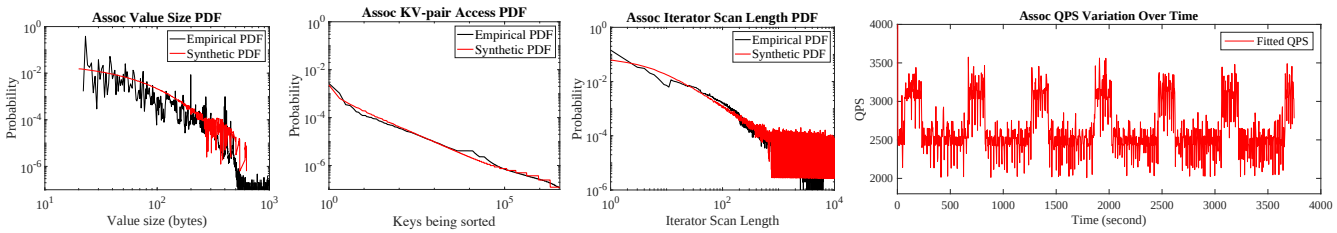


Figure 15: The synthetic workload QPS, and the PDF comparisons between the collected workload and the synthetic workload.

to the Sine model [35] with a 24-hour period.

To compare the workload statistics obtained from benchmarking with those of realistic workloads, we run the new benchmark with a different workload scale: 1) 10 million queries, 2) 30 million existing keys, 3) a 600-second period of QPS *sine*, and 4) a {Get, Put, Iterator} ratio of {0.806, 0.159, 0.035}, respectively (the same as in UDB `Assoc`). We collect the trace during benchmarking and analyze the trace. Figure 15 shows the QPS variation and the probability density function (PDF) comparison of value sizes, KV-pair access counts, and Iterator scan lengths between the UDB `Assoc` workload and the generated workload. Although the scale of the workload generated from our benchmark is different from that of UDB `Assoc`, the PDF figures show that they have nearly the same distribution. This verifies that the generated synthetic workload is very close to the UDB `Assoc` workload in terms of those statistics.

8 Related Work

During the past 20 years, the workloads of storage systems, file systems, and caching systems have been collected and analyzed in many studies. Kavalanekar *et al.* collected block traces from production Windows servers at Microsoft and provided workload characterizations that have benefitted the design of storage systems and file systems tremendously [27]. Riska *et al.* analyzed the disk-level workload generated by different applications [40]. The file system workloads were studied by industrial and academic researchers at different scales [30, 41, 42]. The workloads of the web server caches were also traced and analyzed [2, 3, 43, 46]. While the web cache can be treated as a KV-store, the query types and workloads are different from persistent KV-stores.

Although KV-stores have become popular in recent years, the studies of real-world workload characterization of KV-stores are limited. Atikoglu *et al.* analyzed the KV workloads of the large-scale Memcached KV-store at Facebook [5]. They found that reads dominate the requests, and the cache hit rate is closely related to the cache pool size. Some of their findings, such as the diurnal patterns, are consistent with what we present in Section 4. Major workload characteristics of RocksDB are very different from what Atikoglu *et al.* found in Memcached. Other KV-store studies, such as SILT [31], Dynamo [14], FlashStore [12], and SkimpyStash [13], evaluate designs and implementations with some real-world workloads. However, only some simple statistics of the workloads are mentioned. The detailed workload characteristics, modeling, and synthetic workload generation are missing.

Modeling the workloads and designing benchmarks are also important for KV-store designs and their performance improvements. Several benchmarks designed for big data NoSQL systems, such as YCSB [11], LinkBench [4], and BigDataBench [45], are also widely used to evaluate KV-store performance. Compared with these benchmarks, we further provide the tracing, analyzing, and key-range based benchmarking tools for RocksDB. The users and developers of RocksDB can easily develop their own specific benchmarks based on the workloads they collect with better emulation in both the KV-query level and storage level.

9 Conclusion and Future Work

In this paper, we present the study of persistent KV-store workloads at Facebook. We first introduce the tracing, replaying, analyzing, and benchmarking methodologies and tools that can be easily used. The findings of key/value size distribution, access patterns, key-range localities, and workload variations provide insights that can help optimize KV-store performance. By comparing the storage I/Os of RocksDB benchmarked by YCSB and those of trace replay, we find that many more reads and fewer writes are generated by benchmarking with YCSB. To address this issue, we propose a key-range based model to better preserve key-space localities. The new benchmark not only provides a good emulation of workloads at the query level, but also achieves more precise RocksDB storage I/Os than that of YCSB.

We have already open-sourced the tracing, replaying, analyzing, and the new benchmark in the latest RocksDB release (see the Wiki for more details [20]). The new benchmark is part of the benchmarking tool of *db_bench* [18]. We are not releasing the trace at this time. In the future, we will further improve YCSB workload generation with key-range distribution. Also, we will collect, analyze, and model the workloads in other dimensions, such as correlations between queries, the correlation between KV-pair hotness and KV-pair sizes, and the inclusion of additional statistics like query latency and cache status.

Acknowledgments

We would like to thank our shepherd, George Amvrosiadis, and the anonymous reviewers for their valuable feedback. We would like to thank Jason Flinn, Shrikanth Shankar, Marla Azriel, Michael Stumm, Fosco Marotto, Nathan Bronson, Mark Callaghan, Mahesh Balakrishnan, Yoshinori Matsunobu, Domas Mituzas, Anirban Rahut, Mikhail Antonov, Joanna Bujnowska, Atul Goyal, Tony Savor, Dave Nagle, and many others at Facebook for their comments, suggestions, and support in this research project. We also thank all the RocksDB team members at Facebook. This work was partially supported by the following NSF awards 1439622, 1525617, 1536447, and 1812537, granted to authors Cao and Du in their academic roles at the University of Minnesota, Twin Cities.

References

[1] M. Annamalai. Zippydb: a modern, distributed key-value data store. https://www.youtube.com/watch?v=DfiN7pG0D0k, 2015.

[2] M. Arlitt and T. Jin. A workload characterization study of the 1998 world cup web site. *IEEE network*, 14(3):30–37, 2000.

[3] M. F. Arlitt and C. L. Williamson. Web server workload characterization: The search for invariants. *ACM SIGMETRICS Performance Evaluation Review*, 24(1):126–137, 1996.

[4] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1185–1196. ACM, 2013.

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[6] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. Speicher: Securing lsm-based key-value stores using shielded execution. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 173–190, 2019.

[7] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, et al. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.

[8] 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)*, 26(2):4, 2008.

[9] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz. Realtime data processing at facebook. In *Proceedings of the International Conference on Management of Data*, pages 1087–1098. ACM, 2016.

[10] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[12] B. Debnath, S. Sengupta, and J. Li. Flashstore: high throughput persistent key-value store. In *Proceedings of the VLDB Endowment*, volume 3, pages 1414–1425. VLDB Endowment, 2010.

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

[14] 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*, volume 41, pages 205–220. ACM, 2007.

[15] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strumm. Optimizing space amplification in RocksDB. In *CIDR*, volume 3, page 3, 2017.

[16] Facebook. Cassandra on rocksdb at instagram. https://developers.facebook.com/videos/f8-2018/cassandra-on-rocksdb-at-instagram. 2018.

[17] Facebook. Merge operator. https://github.com/facebook/rocksdb/wiki/Merge-Operator, 2018.

[18] Facebook. db_bench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools, 2019.

[19] Facebook. Myrocks. http://myrocks.io/, 2019.

[20] Facebook. Rocksdb trace, replay, analyzer, and workload generation. https://github.com/facebook/rocksdb/wiki/RocksDB-Trace%2C-Replay%2C-Analyzer%2C-and-Workload-Generationr, 2019.

[21] Facebook. Rocksdb. https://github.com/facebook/rocksdb/, 2019.

[22] Facebook. Single delete. https://github.com/facebook/rocksdb/wiki/Single-Delete. 2019.

[23] S. Ghemawat and J. Dean. Leveldb. *URL: https://github.com/google/leveldb,% 20http://leveldb. org*, 2011.

[24] A. Gupta. Followfeed: Linkedin's feed made faster and smarter. https://engineering.linkedin.com/blog/2016/03/followfeed--linkedin-s-feed-made-faster-and-smarter, 2016.

[25] J. R. Hosking and J. R. Wallis. Parameter and quantile estimation for the generalized pareto distribution. *Technometrics*, 29(3):339–349, 1987.

[26] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-r. Choi. Slm-db: single-level key-value store with persistent memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, 2019.

[27] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda. Characterization of storage workload traces from production windows servers. In *IEEE International Symposium on Workload Characterization (IISWC 08)*, pages 119–128. IEEE, 2008.

[28] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong. Atlas: Baidu's key-value storage system for cloud data. In *IEEE 31st Symposium on Mass Storage Systems and Technologies (MSST 15)*, pages 1–14, 2015.

[29] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[30] A. W. Leung, S. Pasupathy, G. R. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 08)*, volume 1, pages 2–5, 2008.

[31] 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 (SOSP 11)*, pages 1–13. ACM, 2011.

[32] 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)*, 13(1):5, 2017.

[33] MathWorks. Power series models. https://www.mathworks.com/help/curvefit/power.html. 2019.

[34] MathWorks. Power series. https://en.wikipedia.org/wiki/Power_series. 2019.

[35] MathWorks. Sine fitting. https://www.mathworks.com/matlabcentral/fileexchange/66793-sine-fitting. 2019.

[36] Y. Matsunobu. Innodb to myrocks migration in main mysql database at facebook. USENIX Association, May 2017.

[37] S. Nanniyur. Sherpa scales new heights. https://yahooeng.tumblr.com/post/120730204806/sherpa-scales-new-heights, 2015.

[38] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[39] Redis. Redis documentation. https://redis.io/documentation, 2019.

[40] A. Riska and E. Riedel. Disk drive level workload characterization. In *Proceedings of the USENIX Annual Technical Conference (ATC 06)*, pages 97–102, 2006.

[41] D. Roselli and T. E. Anderson. *Characteristics of file system workloads*. University of California, Berkeley, Computer Science Division, 1998.

[42] D. S. Roselli, J. R. Lorch, T. E. Anderson, et al. A comparison of file system workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 00)*, pages 41–54, 2000.

[43] W. Shi, R. Wright, E. Collins, and V. Karamcheti. Workload characterization of a personalized web site and its implications for dynamic content caching. In *Proceedings of the 7th International Workshop on Web Caching and Content Distribution (WCW 02)*, 2002.

[44] J. Wang. Myrocks: best practice at alibaba. https://www.percona.com/live/17/sessions/myrocks-best-practice-alibaba, 2017.

[45] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–499, 2014.

[46] A. Williams, M. Arlitt, C. Williamson, and K. Barker. Web workload characterization: Ten years later. In *Web content delivery*, pages 3–21. Springer, 2005.

[47] S. Zheng, M. Hoseinzadeh, and S. Swanson. Ziggurat: a tiered file system for non-volatile main memories and disks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 207–219, 2019.

A Appendix

A.1 Trace Replay

./db_bench –benchmarks=replay –trace_file=./trace_<Trace Name> –num_column_families=1 -use_direct_io_for_flush_and_compaction=true -use_direct_reads=true -trace_replay_fast_forward=24 –perf_level=2 –trace_replay_threads=3 -use_existing_db=true -db=./<Directory of Existing RocksDB Database for Replay>

A.2 Trace Analyzing

./trace_analyzer -analyze_get -analyze_put -analyze_merge -analyze_delete -analyze_single_delete -analyze_iterator -output_access_count_stats -output_dir=./result_<Trace Name> -output_key_stats -output_qps_stats -output_value_distribution -output_key_distribution -output_time_series -print_overall_stats -print_top_k_access=6 -value_interval=1 -output_prefix= <Trace Name>_result -trace_path=./trace_<Trace Name> ./ <Trace Name>_general.txt

A.3 New Benchmarks

Before running the benchmark, user needs to compile RocksDB db_bench and run it via command lines. Note that, if user runs the benchmark following the 24 hours Sine period, it will take about 22-24 hours. In order to speedup the benchmarking, user can increase the sine_d to a larger value such as 45000 to increase the workload intensiveness and also reduce the sine_b accordingly.

Create a database with 50 million random inserted KV-pairs

./db_bench –benchmarks=fillrandom –perf_level=3 -use_direct_io_for_flush_and_compaction=true -use_direct_reads=true -cache_size=268435456 -key_size=48 -value_size=43 -num=50000000 -db=./<Directory of Generated Database with 50 million KV-pairs>

All_random

./db_bench –benchmarks="mixgraph" -use_direct_io_for_flush_and_compaction=true -use_direct_reads=true -cache_size=268435456 -keyrange_num=1 -value_k=0.2615 -value_sigma=25.45 -iter_k=2.517 -iter_sigma=14.236 -mix_get_ratio=0.83 -mix_put_ratio=0.14 -mix_seek_ratio=0.03 -sine_mix_rate_interval_milliseconds=5000 -sine_a=1000 -sine_b=0.000073 -sine_d=4500 –perf_level=2 -reads=420000000 -num=50000000 -key_size=48 -db=./<Directory of Generated Database with 50 million KV-pairs> -use_existing_db=true

All_dist

./db_bench –benchmarks="mixgraph" -use_direct_io_for_flush_and_compaction=true -use_direct_reads=true -cache_size=268435456 -key_dist_a=0.002312 -key_dist_b=0.3467 -keyrange_num=1 -value_k=0.2615 -value_sigma=25.45 -iter_k=2.517 -iter_sigma=14.236 -mix_get_ratio=0.83 -mix_put_ratio=0.14 -mix_seek_ratio=0.03 -sine_mix_rate_interval_milliseconds=5000 -sine_a=1000 -sine_b=0.000073 -sine_d=4500 –perf_level=2 -reads=420000000 -num=50000000 -key_size=48 -db=./<Directory of Generated Database with 50 million KV-pairs> -use_existing_db=true

Prefix_random

./db_bench –benchmarks="mixgraph" -use_direct_io_for_flush_and_compaction=true -use_direct_reads=true -cache_size=268435456 -keyrange_dist_a=14.18 -keyrange_dist_b=-2.917 -keyrange_dist_c=0.0164 -keyrange_dist_d=-0.08082 -keyrange_num=30 -value_k=0.2615 -value_sigma=25.45 -iter_k=2.517 -iter_sigma=14.236 -mix_get_ratio=0.83 -mix_put_ratio=0.14 -mix_seek_ratio=0.03 -sine_mix_rate_interval_milliseconds=5000 -sine_a=1000 -sine_b=0.000073 -sine_d=4500 –perf_level=2 -reads=420000000 -num=50000000 -key_size=48 -db=./<Directory of Generated Database with 50 million KV-pairs> -use_existing_db=true

Prefix_dist

./db_bench –benchmarks="mixgraph" -use_direct_io_for_flush_and_compaction=true -use_direct_reads=true -cache_size=268435456 -key_dist_a=0.002312 -key_dist_b=0.3467 -keyrange_dist_a=14.18 -keyrange_dist_b=-2.917 -keyrange_dist_c=0.0164 -keyrange_dist_d=-0.08082 -keyrange_num=30 -value_k=0.2615 -value_sigma=25.45 -iter_k=2.517 -iter_sigma=14.236 -mix_get_ratio=0.83 -mix_put_ratio=0.14 -mix_seek_ratio=0.03 -sine_mix_rate_interval_milliseconds=5000 -sine_a=1000 -sine_b=0.000073 -sine_d=4500 –perf_level=2 -reads=420000000 -num=50000000 -key_size=48 -db=./<Directory of Generated Database with 50 million KV-pairs> -use_existing_db=true