X-Engine: An Optimized Storage Engine for Large-scale E-Commerce Transaction Processing

Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He,
Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li
{qushan,xuntao.cxt,beilou.wjy,zhencheng.wyj,dengcheng.hedc,tieying.zhang,lifeifei,sh.wang,mingsong.cw,
junyu}@alibaba-inc.com
Alibaba Group

Abstract

Alibaba runs the largest e-commerce platform in the world serving more than 600 million customers, with a GMV (gross merchandise value) exceeding 768 billion USDs in FY2018. Online e-commerce transactions have three notable characteristics: (1) drastic increase of transactions per second with the kickoff of major sales and promotion events, (2) large amount of hot records that can easily overwhelm system buffers, and (3) quick shift of the "temperature" (hot vs. warm vs. cold) of different records due to the availability of promotions on different categories over different short time periods. For example, Alibaba's OLTP database clusters experienced a 122 times increase of workload on the start of the Singles' Day Global Shopping Festival in 2018, processing up to 491,000 sales transactions per second which translate to more than 70 million database transactions per second. To address these challenges, we introduce X-Engine, an OLTP storage engine built at Alibaba that utilizes a tiered storage architecture with LSM-tree (log-structured merge tree) to leverage hardware acceleration such as FPGA-based compaction, and a suite of optimizations including asynchronous writes in transactions, multi-staged pipelines and incremental cache replacement during compactions. Evaluation results show that X-Engine has clearly outperformed other storage engines under such transaction workloads.

1 Introduction

Alibaba runs the world's largest and busiest e-commerce platform consisting of its C2C retail market Taobao, B2C market Tmall, and other online markets, serving more than 600 million active consumers with a GMV exceeding US$ 768 billion in FY2018. Such online shopping markets have created new ways of shopping and selling. As an example, an online promotion operation can quickly scale its attraction to global customers as soon as it starts, since online shops in a digital market are free from physical restrictions compared to offline brick and mortar stores.

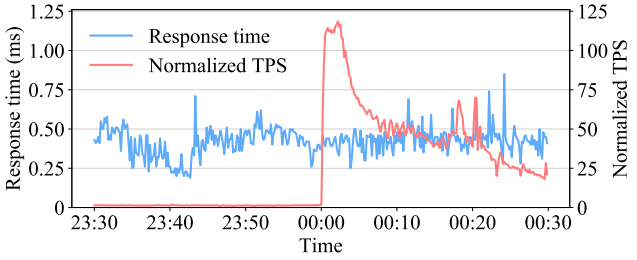


Figure 1: 122 times of sudden increase in transactions per second along with stable response time observed in an online database cluster running X-Engine during the Singles' Day Global Shopping Festival in 2018.

The processing of e-commerce transactions is the backbone of online shopping markets. We find that such transactions have three major characteristics: (1) drastic increase in transactions per second with the kickoff of major sales and promotional events, (2) large amount of hot records that can easily overwhelm system buffers, and (3) quick shift of the "temperature" (hot vs. warm v.s. cold) of different records due to the availability of promotions on different categories over different time at a short time period. We introduce them in detail in the following.

During the 2018 Singles' Day Global Shopping Festival (Nov. 11, 2018), Alibaba's database clusters process up to 491,000 sales transaction per second, which translates to more than 70 million (database) transactions per second. To stand this test, we introduce a new OLTP storage engine, X-Engine, that is optimized for e-commerce workloads, because a significant part of transaction processing performance boils down to how efficiently data can be made durable and retrieved from the storage.

Key challenges for e-commerce workloads. We start by identifying three key technical challenges.

SIGMOD'19, June 2019, Amsterdam, The Netherlands

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The tsunami problem. An e-commerce website needs to run a whole-marketplace scale major promotional event that starts at a particular time and date, for example, the Singles' Day Global Shopping Festival that runs annually on November 11, where major promotions across almost all vendors on Tmall start exactly at midnight. Although many promotions last for the day, there are significant discounts that are only available for either a limited period of time or a amount of quantities (on a first come first serve basis). This creates an abrupt increase on transaction workloads (reflected as a huge vertical spike on transaction per second v.s. time) to the underlying storage engine starting at 00:00:00 on Nov. 11, much like a huge tsunami hitting the shore.

Figure 1 plots the transaction response time and the transaction per second (TPS) (normalized by the average TPS for the day before midnight on Nov. 11) of an online database cluster running on X-Engine on Singles' Day of 2018. In the first second of this day, this cluster embraces a suddenly increased TPS that is about 122 times higher than that of the previous second. With the help of X-Engine, it successfully maintains a stable response time (0.42 ms on average).

To embrace this 122 times spike shown in Figure 1, Alibaba adopts a shared-nothing architecture for OLTP databases where sharding is applied to distribute transactions among many database instances, and then scales the number of instances up before the spike comes. Although this methodology works, it takes significant monetary and engineering costs, because it demands a large number of database instances. In this paper, we address this problem by improving the single-machine capacity of OLTP engines, so that the number of instances required for a given spike and throughput can be reduced, or the throughput achieved given a fixed cost is increased.

The flood discharge problem. The storage engine must be able to quickly move data from the main memory to the durable storage (i.e., discharge the flood building up in the memory) while processing highly concurrent e-commerce transactions. Online promotions create rush hours in online e-commerce transactions, which contain a much higher amount of writes (e.g., place order, update inventory and make payments) compared with those in non-rush hours. Even though the main memory capacity has steadily increased in recent years, it is still dwarfed by the amount of records to be inserted or updated by the large number of transactions over a massive collection of products and customers during the Singles' Day. Thus, we have to exploit the memory hierarchy consisting of RAM, SSD and HDD. X-Engine leverages this hierarchy by adopting a tiered storage layout, where we can place data in different tiers according to their temperatures (i.e., the access frequency), and use new memory technologies like NVM for some tiers. This is much like discharging a running flood through reservoirs of increasing size level by level.

The LSM-tree structure is a natural choice for a tiered storage. Standard techniques to accelerate writes include the append-only method on log-based data structures [26, 33], optimized tree-based structures [3, 17], and a hybrid form of both [30]. We find that any one of these existing methods alone is not sufficient for serving such e-commerce transactions: some assume a columnar storage that is not suitable for write-intensive transactions; others trade the performance of point and range queries to improve that of writes, which are not suitable for mixed read and write e-commerce workloads. A LSM-tree structure contains a memory-resident component where we can apply the append-only method for fast inserts, and disk-resident components containing multiple inclusive levels organized in a tree with each level significantly larger than its adjacent upper level [14, 15, 26]. This data structure well fits a tiered storage, facilitating us to address the tsunami problem and the flood discharge problem.

The fast-moving current problem. For most database workloads, hot records typically exhibit a strong spatial locality over a stable period of time. This is not always the case for e-commerce workloads, especially on a major promotion event like Singles' Day Shopping Festival. The spatial locality of records is subject to quick changes over time. This is due to the availability of different promotions on different categories or records that are purposely placed out over time. For example, throughout the day, there are "seckill" promotion events (selling merchandises so hot that you must seize the second they become available to buy it) that are held for different categories or brands to stimulate demands and attract customers to shop over different merchandises pacing out over time. This means that hot records in the database cache are subject to constant changes, and the temperature of any record may change quickly from cold/warm to hot or hot to cold/warm. If we view the database cache as a reservoir and the underlying (large) database as the ocean, this phenomenon results in a *current* (i.e., hot vs cold records) that moves quickly towards any direction in a very deep *ocean* (i.e., the huge database stored on X-Engine). The storage engine needs to ensure that emerging hot records can be retrieved from the deep water as quickly as possible and cache them effectively.

Our contributions. We introduce X-Engine in this paper. X-Engine is a LSM-tree based storage engine for OTLP databases that is designed to address the above challenges faced by Alibaba's e-commerce platform. It processes most requests in the main memory by exploiting the thread-level parallelism (TLP) in multi-core processors, decouples writes from transactions to make them asynchronous, and decomposes a long write path into multiple stages in a pipeline in order to increase the overall throughput. To address the

flood discharge problem, X-Engine exploits a tiered storage approach to move records among different tiers taking advantage of a refined LSM-tree structure and optimized compaction algorithms. We also apply FPGA-offloading on compactions. Finally, to address the fast-moving current problem, we introduce a multi-version metadata index which is updated in a copy-on-write fashion to accelerate random lookups in the tiered storage regardless of data temperatures. To summarize, our contributions are:

- We identify three major challenges for an OLTP storage engine in processing e-commerce transactions, and design X-Engine based on the LSM-tree structure.
- We introduce a suite of optimizations for addressing the identified problems, such as a refined LSM-tree data structure, FPGA-accelerated compactions, asynchronous writes in transactions, multi-staged pipelines, and multi-version metadata index.
- We evaluate X-Engine extensively using both standard benchmarks and e-commerce workloads. Results show that X-Engine outperforms both InnoDB and RocksDB in e-commerce workloads during online promotions. We also show that X-Engine has stood the test of the tsunami of the Singles' Day.

X-Engine has been deployed at Alibaba and successfully supported the Singles' Day Global Shopping Festival of 2018. This paper is organized as follows. Section 2 gives the overview of X-Engine. Section 3 introduces the detailed designs of our optimizations. We evaluate our design and optimizations in X-Engine in Section 4, by comparing it against InnoDB and RocksDB using MySQL. Finally, we discuss related work in Section 5 and conclude in Section 6.

2 System Overview

X-Engine is a tiered OLTP storage engine based on an optimized LSM-tree structure. As discussed in Section 1, because the data locality in e-commerce transaction workloads is constantly changing, we are motivated to differentiate the placement of records in different storage tiers to serve hot, warm, and cold records respectively. This allows X-Engine to tailer-design data structures and access methods according to record temperatures (i.e., frequencies of being accessed in a recent window). Furthermore, because e-commerce transactions involve a large number of inserts and updates, that are mixed with point and range queries, we design X-Engine based on LSM-tree due to its excellent write performance [26]. However, we find that a standard LSM-tree is not sufficient for supporting e-commerce transaction workloads. Thus, we are motivated to design and apply a suite of optimizations in X-Engine to address the three previously identified challenges. We introduce the system overview and detailed designs of X-Engine in the following sections.

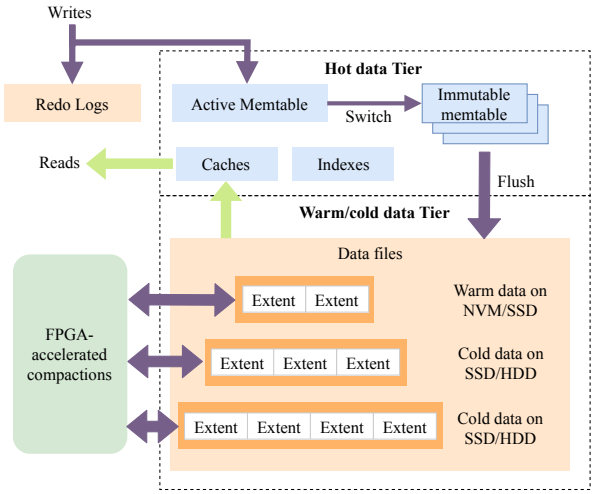


Figure 2: The architecture of X-Engine.

Storage layout. Figure 2 shows the architecture of X-Engine. X-Engine partitions each table into multiple sub-tables, and maintains a LSM-tree, the associated metasnapshots and indexes for each sub-table. X-Engine contains one redo log per database instance. Each LSM-tree consists of a hot data tier residing in main memory and a warm/cold data tier residing in NVM/SSD/HDD (that are further partitioned into different levels), where the term hot, warm, and cold refers to data temperatures, representing the ideal access frequencies of data that should be placed in the corresponding tier. The hot data tier contains an *active memtable and multiple immutable memtables*, which are skiplists storing recently inserted records, and *caches* to buffer hot records. The warm/cold data tier organizes data in a tree-like structure, with each level of the tree storing a sorted sequence of *extents*. An *extent* packages blocks of records as well as their associated filters and indexes. We explored machine learning techniques to identify appropriate data temperatures, which is briefly discussed in Appendix C and its full details are out of the scope of this work.

X-Engine exploits redo logs, metasnapshots, and indexes to support Multi-version Concurrency Control (MVCC) for transaction processing. Each *metasnapshot* has a *metadata index* that tracks all memtables, and extents in all levels of the tree in the snapshot. One or multiple neighboring levels of the tree forms a tier to be stored on NVM, SSD, and HDD, respectively. In X-Engine, tables are partitioned into several sub-tables. Each sub-table has its own hot, warm and cold data tiers (i.e., LSM-trees). X-Engine stores records in a row-oriented format. We design a multi-version memtables to store records with different versions to support MVCC (introduced in Section 3.2.1). On the disks, the metadata indexes track all the versions of records stored in extents. We introduce the details of data structures in Section 3.1.

Table 1: Summary of optimizations in X-Engine.

Optimization	Description	Problem
Asynchronous writes in transactions	Decoupling the commits of writes from the processing transactions.	Tsunami
Multi-staged pipeline	Decomposing the commits of writes to multiple pipeline stages.	
Fast flush in $Level_0$	Refining the $Level_0$ in the LSM-tree to accelerate flush.	Flood discharge
Data reuse	Reusing extents with non-overlapped key ranges in compactions.	
FPGA-accelerated compactions	Offloading compactions to FPGAs.	
Optimizing extents	Packaging data blocks, their corresponding filters, and indexes in extents.	Fast-moving current
Multi-version metadata index	Indexing all extents and memtables with versions for fast lookups.	
Caches	Buffering hot records using multiple kinds of caches.	
Incremental cache replacement	Replacing cached data incrementally during compactions.	

The read path. The read path is the process of retrieving records from the storage. The original LSM-tree design does not enjoy good read performance. A lookup first searches the memtables. On misses in the memtables, it has to traverse each successive level one by one. In the worst case, a lookup has to end up with scanning all levels till the largest level to conclude that the queried records do not exist. To accelerate this process, a manifest file has been proposed to locate the target SST containing queried keys [8]. Bloom filters are also applied within each SST to facilitate early termination.

We introduce snapshots to ensure queries read the correct versions of records. To achieve a good response time for point lookups which are common in e-commerce transactions, we optimize the design of extents, introduce a metadata index that tracks all memtables and extents, and a set of caches to facilitate fast lookups. We also propose an incremental cache replacement method in compactions to reduce unnecessary cache evictions caused by compactions.

The write path. The write path includes the physical access path and the associated process of inserting or updating records in the storage engine. In a LSM-tree KV store, arriving key-value pairs are appended or inserted into the active memtable. Once entirely filled, an active memtable is switched to be immutable waiting to be flushed to disks. Meanwhile, a new empty active memtable is created. To support highly concurrent transaction processing, the storage engine needs to make new records durable through logging in the persistent storage (e.g., SSD), and insert them into memtables at high speed. We differentiate long-latency disk I/Os and low-latency memory accesses in this process and organize them in a *multi-staged pipeline* to reduce idle states per thread and improve the overall throughput (Section 3.2.3). To achieve a high level of concurrency, we further decouple the commits of writes from the processing of transactions and optimize the thread-level parallelism for them separately.

Flush and compaction. A LSM-tree structure relies on flush and compaction operations to merge data that may overwhelm the main memory from memtables to disks and keep the merged data in a sorted order. Immutable memtables are flushed to $Level_0$, during which records are sorted and packaged in sorted sequence tables (SSTs) [15]. Each SST occupies an exclusive range of keys, and hence a level may contain multiple SSTs. When the size of SSTs in $Level_i$ reaches a threshold, they are merged with SSTs with overlapping key ranges from $Level_{i+1}$. This merge process is called *compaction* in some systems because it also removes records that are marked to be deleted. The original compaction algorithm reads SSTs from both levels, merges them, and then writes the merged results back to the LSM-tree. This process has several drawbacks: it consumes significant CPUs and disk I/Os; the same record is read from and written to the LSM-tree multiple times, causing write amplification; it invalidates the cached contents of records being merged even if their values stay the same.

In X-Engine, we apply a suite of techniques to address these drawbacks. We first optimize the flush of immutable memtables. For compactions, we apply *data reuse* that reduces the number of extents to be merged, *asynchronous I/O* that overlaps merging with disk I/Os, and FPGA-offloading to reduce CPU consumption.

Summary of optimizations. Table 1 summarizes the various optimizations with respect to specific problems that they are designed to address. We introduce the detailed design of X-Engine in the following section.

3 Detailed Design

In this section, we start with elaborating how X-Engine processes a transaction, and then introduce the detailed designs of X-Engine's key components, as shown in Section 2, including the read path, write path, flush and compaction.

X-Engine applies MVCC and 2PL (two-phase locking) to achieve the isolation level of SI (snapshot isolation) and RC (read committed) to guarantee the ACID properties for transactions. Each tuple in X-Engine contains a transaction ID, indicating the transaction that writes this tuple. X-Engine tracks the maximum of these versions among all committed transactions as the LSN (log sequence number). Each incoming transaction uses the LSN it sees as its snapshot. A transaction only reads tuples with the largest transactions ID that is smaller than its own LSN, and adds a row lock to each row it writes to avoid write conflicts.

Figure 3 shows an overview of processing a transaction in X-Engine. This process consists of a *Read/Write Phase* and a

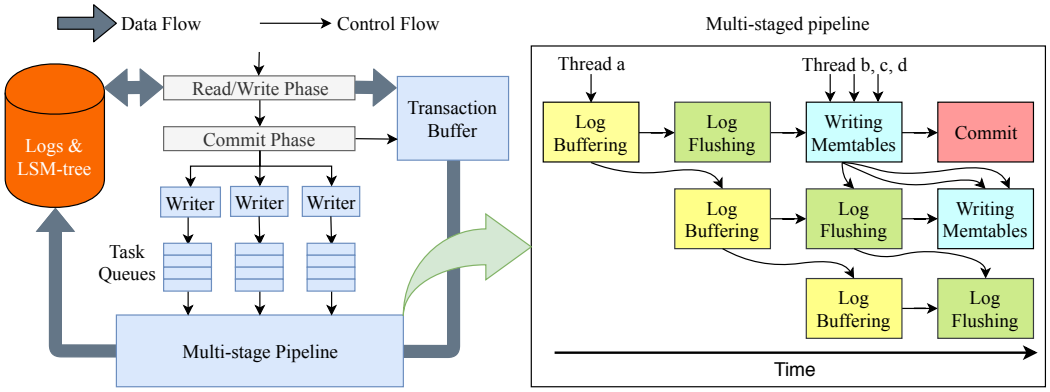


Figure 3: Detailed design of transaction processing in X-Engine.

Commit Phase. All read requests of a transaction are served in the Read/Write Phase through the read path accessing the LSM-tree. In this phase, records to be inserted or updated in a transaction are written to the *Transaction Buffer*. Next, in the Commit Phase, tasks writing records from the transaction buffer to the storage are distributed to multiple write task queues. A multi-staged pipeline is introduced to process all these write tasks by logging the corresponding records and inserting them into the LSM-tree. We introduce the details of the read path, the write path, the flush and the compaction operations of the LSM-tree in the following.

3.1 The read path

We start with our designs of data structures, including extents, caches and indexes. For each data structure, we introduce how it facilitates fast lookups in the read path.

3.1.1 Extent Figure 4 shows the layout of an extent, which consists of data blocks, the schema data, and the block index. Records are stored in the row-oriented style in data blocks. The schema data tracks the types of each column. The block index keeps the offset for each data block. Among the current deployment in our production system, we tune the total size of an extent to 2 MB across all levels of the LSM-tree. Because many e-commerce transactions access records in a highly skewed manner, keeping extents in this size allows many extents to be reusable during compaction (more details in Section 3.3.2). This design also facilitates the incremental cache replacement during compactions (an optimization introduced in Section 3.1.4).

We store the schema data with versions in each extent to accelerate DDL (Data Definition Language) operations. With this design, for example, when adding a new column to a table, we only need to enforce this new column on new extents with new versions, without modifying any existing ones. When a query reads extents with schemas of different versions, it accords with the most recent version and fills default values to the empty attributes of records with the

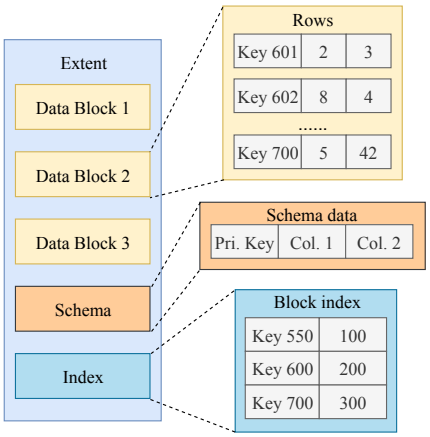


Figure 4: The layout of an extent.

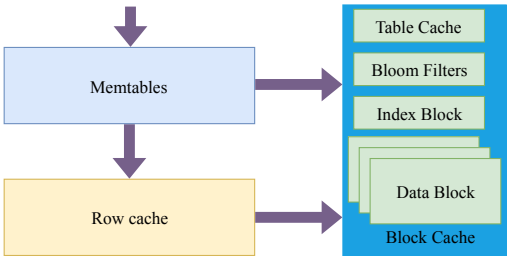


Figure 5: Structure of the caches.

old schema. This fast DDL feature is critical for online e-commerce businesses which adapt their designs of database schemas to changes in their requirements frequently.

3.1.2 Cache Figure 5 illustrates the database caches in X-Engine. We optimize the row cache specifically for point lookups that are the majority queries in e-commerce workload at Alibaba. The row cache buffers records using the LRU cache replacement policy, regardless of which levels a record resides in the LSM-tree. Thus, even the records in the largest level of the LSM-tree stands a chance to be cached, as long as a query accesses it. Once a lookup missed the memtables, the key of the query is hashed to its corresponding slot in the row cache for matches. As a result, retrieving cached records only takes O(1) time in the caches.

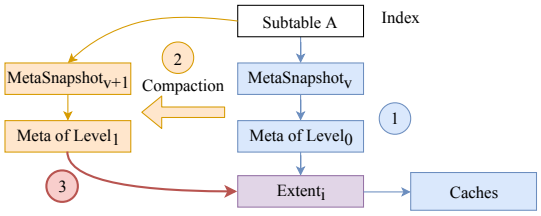


Figure 6: An example of updating the metadata index in the copy-on-write manner.

We only keep the latest versions of records in the row cache, which have the largest chance to be accessed due to the temporal locality. To achieve this, we replace the old versions of records with new ones in the row cache during flush, reducing cache misses caused by flush.

The block cache serves each request that missed the row cache, or lookups from range scans. The table cache contains the metadata information of sub-table headers leading to the corresponding extent. With the extent located, we use Bloom filters to filter unmatched keys out. Then, we search the index block to locate the record, and finally retrieve it from its data block.

The block cache is important for caching emerging hot records, once the temperatures of records shift and cause frequent row cache misses. Due to the spatial locality of records, emerging hot records and cached records may very likely come from the same data blocks. On a block cache hit, the lookup response time is much shorter than accessing the disks, and the row cache can be updated quickly.

3.1.3 Multi-version Metadata Index In X-Engine, we introduce a multi-version metadata index. Figure 6 illustrates its structure. The LSM-tree of each sub-table has its associated metadata index, which starts from a root node representing the sub-table. Each modification of the index creates a new metasnapshot, that points to all associated levels and memtables without modifying nodes of existing metasnapshots (i.e., the copy-on-write method). In Figure 6, $extent_i$ is originally part of the $Level_0$ and cached (colored in blue). When a compaction reusing this extent completes, a new $MetaSnapshot_{v+1}$ is created by the side of $MetaSnapshot_v$, linking to the newly merged $Level_1$ (colored in orange). The metadata of $Level_1$ only needs to point to $extent_i$ without actually moving it in the storage (colored in red), leaving all its cached contents intact. Taking advantage of this copy-on-write method, transactions can access any versions they want in a read-only manner, without the need to lock an index during data accesses. We apply garbage collections to remove outdated metasnapshots. Similar design has also been explored by other LSM-tree storage engines such as RocksDB [13].

3.1.4 Incremental cache replacement In LSM-trees, as compaction moves extents around in the storage, they often cause significant cache evictions that hurt the performance of lookups. Even if the values of cached records do not change, they may have been moved around in the storage if their extents share overlapping key ranges with other extents. Because a compaction merges many extents each time, caches suffer from evictions in large batches, resulting in distinct slowdowns for lookups and unstable response time.

To address this issue, rather than evicting all compacted contents from caches, we propose an incremental replacement in the block cache. During the compaction, we check whether the blocks of an extent to be merged are cached. If so, we replace the old blocks in the cache with the newly merged ones at the same physical address, instead of simply evicting all old ones. This approach reduces the cache misses by keeping some blocks both updated and unmoved in the block cache.

3.2 The write path

In this section, we start with optimizing the memtable structures that receive incoming records for the LSM-tree of each sub-table. Next, we introduce how we design the write tasks queues and the multi-staged pipeline in the write path, which are shared by all LSM-trees.

3.2.1 Multi-version memtable We implement the memtable as a lock-free skiplist like many other systems to achieve good lookup and insert performance [8]. However, the state-of-the-art implementation of the skiplist-based memtables has a performance issue on querying hot records. Frequent updates on a single record generate many versions. If a hot record matches the predicate of a query with interests in only the latest version, the query may have to scan many old versions to locate the requested one. Online promotions in e-commerce platforms amplify these redundant accesses when customers place orders on hot merchandises.

In X-Engine, we append new versions of the same record next to the original node vertically, forming a new linked list. Figure 7 shows the proposed structure, where the blue nodes store records with distinct keys and the yellow nodes store multiple versions of the same record. The distinct keys are organized in a skiplist while each key's multiple versions are stored in a linked list. Updates on a hot record grow its corresponding linked list. Moreover, new versions that are usually referenced by incoming transactions are kept closer to the bottom level of the skiplist of unique keys, as shown in Figure 7 where version 99 is the latest version. This design reduces the overhead caused by scanning over unnecessary old versions.

3.2.2 Asynchronous writes in transactions The conventional one-thread-one-transaction method in storage engines like InnoDB has a significant drawback on the write efficiency. In this approach, a user thread executes a transaction from the

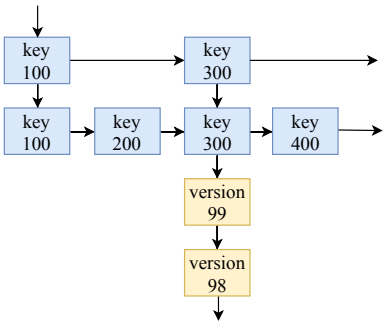


Figure 7: Structure of the multi-version memtables.

beginning to the end. Although this method makes it easy to implement both the execution of writes and concurrency control of transactions while a user thread does not write logs on its own, the thread has to wait for the long-latency of disk IOs, while writing logs, to complete.

In X-Engine, we choose another approach to decouple the commits of writes from their corresponding transactions and group them for batch processing. As shown in Figure 3, we first distribute write tasks among multiple lock-free write task queues. After this, most threads can return asynchronously to process other transactions, leaving only one thread per queue to participate in committing the write tasks in the multi-staged pipeline (to be introduced below). In this way, writes in transactions are made asynchronous. In a highly concurrent workload, this approach allows more threads to process write tasks from concurrent transactions. The optimal number of queues is bounded by the I/O bandwidth available in the machine, and contentions among multiple threads over the head of each lock-free queue. We find that eight threads per queue can saturate the I/O bandwidth in a 32-core machine (Section 4.1), and allocating more threads per queue reduces the throughput due to contentions. Furthermore, after this decoupling process, we group write tasks within the same queue together and process them in batches. Comparing to single transaction commit, the batch commit can significantly improve the I/O and therefore increase the throughput. We optimize the efficiency of the batch processing in the following discussion.

3.2.3 Multi-staged pipeline The write path is a long sequence of multiple operations accessing both the main memory and the disk with changing computation workload along its execution. This makes it challenging to hide memory accesses and disk I/Os with computations.

To address this issue, we decompose the write path into multiple stages. The right half of Figure 3 shows the overview of the four-staged pipeline, where the stages access the main memory and the disk alternately. In the first stage, log buffering, threads collect the WALs (write-ahead logs) of each write requests from the transaction buffer to memory-resident log buffers, and calculate their corresponding CRC32 error-detecting codes. This stage involves significant computations and only main memory accesses. In the second stage, log flushing, threads flush logs in the buffer to disks. With the log flushed, the log sequence number advances in the log files. These threads then push write tasks that have already been logged into the next stage, writing memtables. Here, multiple threads append records in the active memtable in parallel. This stage only accesses the main memory. All such writes can be recovered from the WAL after failures. In the last stage, commits, transactions with all its tasks finished are finally committed by multiple threads in parallel, with resources they used such as locks released.

In this pipeline, we schedule threads for each stage separately according to their requirements to match the throughput of each stage with others in order to maximize the total throughput. Although the first three stages are memory-intensive, the first and the second stages access different data structures in the main memory, while the second one writes to the disk. Thus, overlapping them improves the utilization of the main memory and disks.

Furthermore, we throttle the number of threads for each stage respectively. Due to the strong data dependencies in the first two stages, we only schedule one thread for each stage (e.g., thread 'a' in Figure 3). For the other stages, we allocate multiple threads for parallel processing (e.g., threads b, c, d in Figure 3). All threads pull tasks from stages for processing. Pulling tasks from the first two stages operate preemptively, allowing only the first arriving thread to process the stage. The other stages are embarrassingly parallel, allowing multiple threads to work in parallel.

3.3 Flush and Compaction

Next, we introduce how to optimize the flush and compaction operations that organize records in X-Engine's tiered storage.

3.3.1 Fast flush of warm extents in $Level_0$ We rely on flush to avoid OOM (out-of-memory) failures in X-Engine, which is a significant risk upon incoming spikes of transactions. In X-engine, each flush operation converts its immutable memtables to extents, append them to $Level_0$ and leave without merging them with existing records. However, this process leaves sets of unsorted extents. Queries now have to access all extents to find potential matches. The disk I/Os involved in this process are expensive. Although the size of $Level_0$ is probably less than 1% of the entire storage, it contains records that are only slightly older than those recently inserted ones in the memtables. Due to the strong temporal localities in e-commerce workloads, incoming queries may very likely require these records. Thus, we refer to extents in $Level_0$ as the *warm extents*.

We introduce intra-$Level_0$ compactions to actively merge warm extents in $Level_0$, without pushing merged extents to

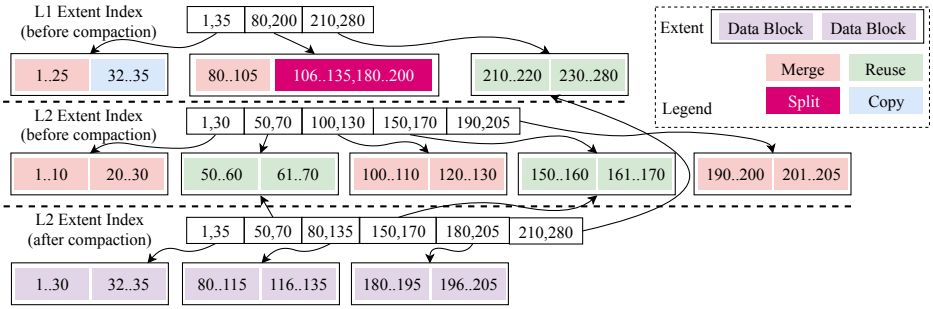


Figure 8: An example of data reuse during compactions.

the next $Level_1$. This approach keeps warm records in the first level of the LSM-tree, preventing queries from going deeper in the tree to retrieve these records. On the other hand, due to the small size of $Level_0$ compared with other levels, intra-$Level_0$ compactions only need to access a small portion of extents, unlike other compactions that merge extents extensively in deeper levels. Because of this lightweight consumption of CPUs and I/Os, the intra-$Level_0$ compactions can be executed frequently.

3.3.2 Accelerating compactions Compaction operations involve expensive merge operations. We apply three optimizations on compactions: the data reuse, the asynchronous I/O, and FPGA offloading.

Data reuse. We reuse extents and data blocks wherever applicable during compactions to reduce the number of I/Os required for merging between two adjacent levels (i.e., $Level_i$ and $Level_{i+1}$). To increase the opportunities for reuse and make it effective, we reduce the size of an extent to 2 MB, and further divide an extent into multiple 16 KB data blocks. If the key range of an extent involved in a compaction does not overlap with those of other extents, we reuse it by simply updating its corresponding metadata index (as introduced in Section 3.1.3) without actually moving it on the disk. We show an example in Figure 8, in which three extents from $Level_1$ with key ranges $[1, 35]$, $[80, 200]$ and $[210, 280]$ are to be compacted with five extents from $Level_2$ with key ranges $[1, 30]$, $[50, 70]$, $[100, 130]$, $[150, 170]$ and $[190, 205]$. We list different cases of reuse in the following:

- Extent $[210, 280]$ from $Level_1$, $[50, 70]$ from $Level_2$ are reused directly.
- $Level_1$ extent $[1, 35]$ overlaps with $Level_2$ extent $[1, 30]$. However, only one data block $[1, 25]$ in the former overlaps with data blocks in the later. Thus, data block $[32, 35]$ is reused.
- $Level_1$ extent $[80, 200]$ overlaps with multiple extents in $Level_2$. Its second data block overlaps with three extents in $Level_2$. However, keys in this data block are sparse, as there is no key between 135 and 180. Thus, we split it into two data blocks: $[106, 135]$, and $[180, 200]$, and merge them with extent $[100, 130]$ and $[190, 205]$ from $Level_2$, respectively. Extent $[150, 170]$ is reused directly.

Asynchronous I/O. At the extent level, a compaction operation consists of three disjoint phases: (1) retrieving two input extents from the storage, (2) merge them, and (3) write merged extents (one or many) back to the storage. While the first and third phases are I/O phases, the second phase is a computation phase. We issue asynchronous I/O requests in the first and third phase. The computation phase is implemented as a call back function of the first I/O phase. Threads can process the computation phase of other compaction tasks while waiting for its first phase to complete, when the storage engine is running multiple compaction tasks in parallel.

FPGA offloading. We introduce FPGAs to accelerate compactions, and reduce their resource consumption on CPUs. With the two optimizations introduced above, compactions running on CPUs still consume multiple threads. Because compactions work on independent pairs of extents from two consecutive levels of the LSM-tree, a compaction task is embarrassingly parallel at the granularity of such pairs. Thus, it can be split into multiple small tasks. We offload such small tasks to FPGAs, and process them in a streaming manner. Each compacted extent is transferred back to disks. With such offloading, CPU threads are released from the heavy burden of merging extents. Consequently, we are able to allocate more threads to process concurrent transactions. The details of this FPGA-based compaction acceleration is beyond the scope of this paper and will be explored separately.

3.3.3 Scheduling compactions LSM-trees rely on compactions to keep records in the sorted order and remove records marked as to be deleted from the storage. Without removing deleted records in time, lookup queries may have to traverse over many invalid records in the storage, hurting the read performance significantly. Compactions also help to merge sub-levels in $Level_0$ to reduce the lookup costs in this level.

In X-Engine, we introduce rule-based scheduling for compactions to exploit these benefits. We distinguish compactions based on what levels they operate on: intra-$Level_0$ compactions (merging sub-levels within $Level_0$), minor compactions (merging two adjacent levels), major compactions

(merging the largest level and the level above it), and self-major compactions (merging within the largest level to reduce fragmentations and remove deleted records). Compaction is triggered when the total size or the total number of extents of a level reach their predefined thresholds. All triggered compaction jobs are enqueued into a priority queue. The rules for determining the priority are open for configurations, which further depends on the requirements of different applications on top of the database.

In the following, we show one example of the configuration tailored for one online application at Alibaba. In this application, there are frequent deletions. When the number of records that should be deleted reaches its threshold, a compaction is triggered to remove these records, and this compaction is given the highest priority to prevent wastes of storage spaces. After deletions, intra-$Level_0$ compactions are prioritized to help to accelerate the lookups of recently inserted records in $Level_0$. Minor compactions, major compactions, and self-major compactions are prioritized in this order, respectively:

(1) Compactions for deletions.
(2) Intra-$Level_0$ compactions.
(3) Minor compaction.
(4) Major compaction.
(5) Self-major compactions.

Compactions of different sub-tables, and thus different LSM-trees, can be scheduled for concurrent executions. This parallelism is embarrassing because of the data independence among sub-tables. Within each sub-table, only one compaction is executed at a time. While it is possible to execute multiple compactions of the same LSM-tree without corrupting the data or causing any data contentions, we have sufficient opportunities for concurrent compactions at the level of sub-tables for performance benefits.

4 Evaluation

In this section, we first show how X-Engine performs when processing e-commerce OLTP workloads with mixtures of different types of queries. We then evaluate X-Engine's contributions in addressing the three identified challenges: the tsunami problem, the flood discharge problem, and the fast-moving current problem. We have already measured the real-world performance of MySQL databases running X-Engine during the Singles' Day in Figure 1.

4.1 Experimental setup

We compare X-Engine with two other popular storage engines: InnoDB [25] and RocksDB [14]. InnoDB is the default storage engine of MySQL. RocksDB is a widely used LSM-tree storage engine, developed from LevelDB [15]. At Alibaba, we use both InnoDB and X-Engine with MySQL 5.7 to process e-commerce transactions in many of our database clusters. Thus, MySQL 5.7 is a natural choice to evaluate InnoDB and X-Engine. We used the latest release (Nov 2018) of RocksDB [13], and MyRocks (MySQL on RocksDB) [10].

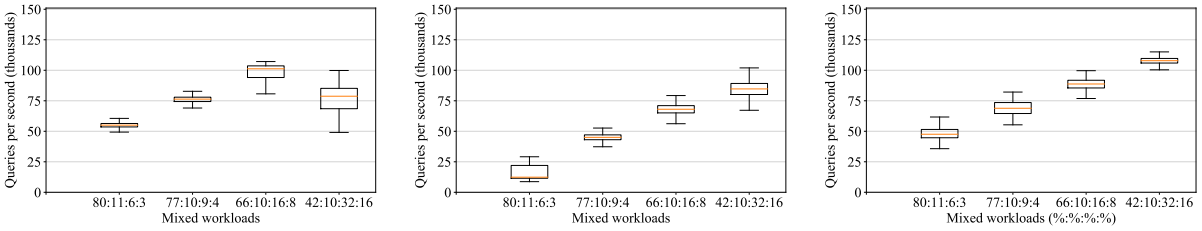
As discussed in Section 1, we apply sharding on multiple database instances for distributed processing in production, the details of sharding is not the focus of this paper. We deploy each storage engine and its associated MySQL instance in a single node setup (i.e., the target machine). The target machine consists of two 16-core Intel E5-2682 processors (64 hardware threads in total), a 512 GB Samsung DDR4-2133 main memory and a 340 GB Intel SSD. The OS is Linux 4.9.79. We use another machine with the same hardware configuration (i.e., the client machine) to issue SQL queries for the target machine. These two machines are located in the same data center with a network latency of around 0.07 ms between them, which is included in all measurements.

In addition to popular benchmark toolkits such as SysBench and dbbench, we apply X-Bench, a benchmark toolkit developed for stress-testing at Alibaba, to synthesize the e-commerce workload consisting of a mixture of point lookups, range lookups, updates, and inserts. With the help from Alibaba's database administrators and database users who run the e-commerce businesses online, we are able to simulate the real-world e-commerce workload closely by tuning the mixing ratio of these four query types. More details about X-Bench are introduced in Appendix B.

Note that for all box plots we used 25% and 75% quantiles for box boundaries, average as the yellow line in the box, and min and max values for external bounds (horizontal lines).

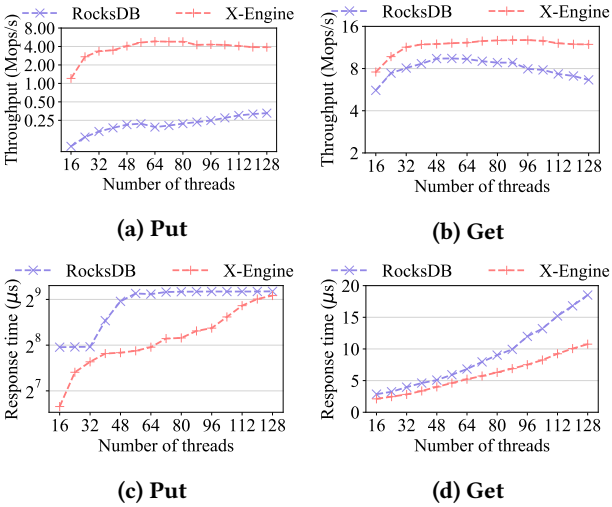
4.2 E-commerce transactions

Figure 9 shows the performance while processing the e-commerce workload. We start with a mixture of 80% point lookups, 11% range lookups, 6% updates, and 3% inserts (denoted as 80:11:6:3 in the figure). This mixture best resembles the read-intensive workload in many Alibaba's databases without the impacts of online promotions. We gradually scale the shares of reads (point and range lookups) down to 42% and 10% while keeping the same percents of updates and inserts, respectively. This 42:10:32:16 mixture is very close to the mixture we observe during online promotions like the Singles' Day. For mixtures that are more read-intensive, InnoDB is both faster and more stable than RocksDB and X-Engine. In LSM-tree systems, lookups missing memtables and caches in the main memory suffer from accessing one or more levels in the disk which drag down the response time and increase the variance. With a suite of optimizations on the read path, X-Engine performs only slightly worse (<10%) than InnoDB in these read-intensive cases. In the crucial 42:10:32:16 case representing the Singles' Day workload, X-Engine achieves very stable performance in terms of QPS, and outperforms InnoDB and RocksDB by 44% and 31% in average, respectively.



(a) InnoDB (b) RocksDB (c) X-Engine

Figure 9: Throughput of MySQL with the three storage engines using the e-commerce workloads with different mixtures of point lookups, range scans, updates, and inserts. The y-axis of these figures are the same.



(a) Put (b) Get

(c) Put (d) Get

Figure 10: Throughput and response time of storage engines through key-value interfaces.

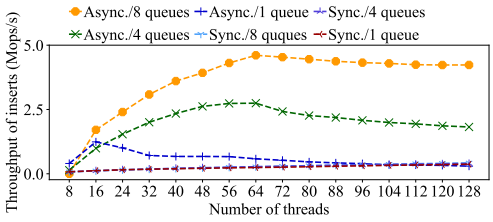


Figure 11: Throughput of inserts with and without optimizing the write path.

Next, we evaluate the impacts of optimizations that help X-Engine process highly concurrent e-commerce transactions with a significant amount of writes.

4.3 The tsunami problem

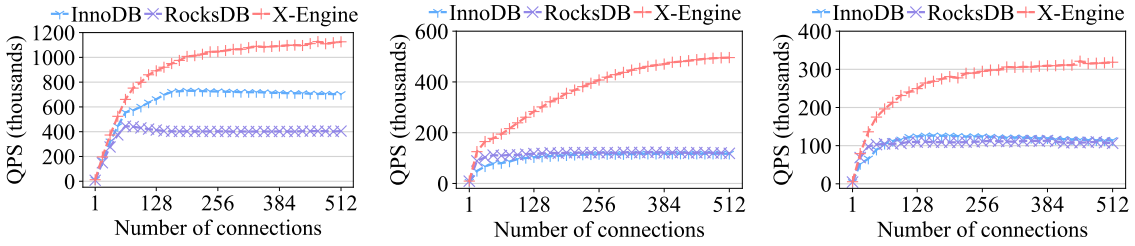
Recall that addressing the tsunami problem translates to improve the peak throughput for the storage engine. In this section, we use dbbench [24] to evaluate storage engines through key-value (KV) interfaces directly, and use SysBench [19] to evaluate their corresponding MySQL instances through their SQL interfaces. In these evaluations, each KV pair has 16-byte key and a 10-byte value; each record contains a 12-byte key and 500-byte attributes. These sizes are common in e-commerce transactions. We omit InnoDB in the KV evaluation because it does not have a KV interface.

4.3.1 Running KV operations without MySQL As Figure 10 shows, we pressure the KV interfaces of the storage engines with an increasing number of software threads up to 128. For put, X-Engine achieves up to 23 times higher throughput than RocksDB with comparable response times. For get, X-Engine is up to 1.68 and 1.67 times faster that RocksDB in terms of throughput and response time, respectively.

To explain these results, we further show the impacts of asynchronous writes, the write task queues, and the multi-staged pipeline (introduced in Section 3.2.2) in Figure 11. With the same eight write task queues, the peak of X-Engine with asynchronous writes is 11 times faster than that of synchronous writes. By increasing the number of write tasks queues from one to eight, X-Engine becomes 4 times faster. Note that using only one task queue disables the multi-staged pipeline as there is only one thread in the pipeline, so that no stage can be executed in parallel. The highest throughput is achieved when all hardware threads are used, showing that X-Engine exploits the thread-level parallelism efficiently.

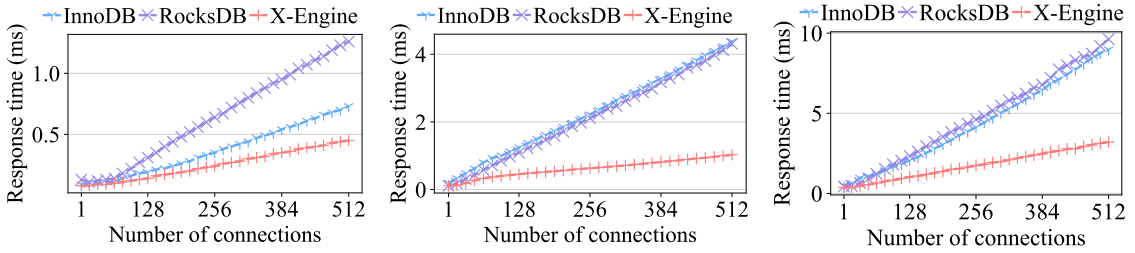
4.3.2 Running SQL queries with MySQL Figure 12 and Figure 13 plot the measured throughput and response time of three storage engines through their SQL interfaces, respectively. In these experiments, we scale the number of connections from 1 to 512. In terms of throughput, X-Engine is 1.60, 4.25, and 2.99 times faster than the second best for point lookups, insert, and update queries respectively. The throughput and response time of X-Engine also scales much better with increasing number of connections, as the result of its improved thread-level efficiency shown in Figure 11.

Although range lookups account for only a small minority of queries in Alibaba's e-commerce workload, it is a necessary feature for online databases. Figure 14 shows the throughput of range lookups in three storage engines with varying numbers of records scanned. Both X-Engine and RocksDB perform worse than InnoDB, because their LSM-tree storage layouts are not friendly to scans. It is our ongoing work to adopt scan optimizations in X-Engine.



(a) Point lookup (b) Insert (c) Update

Figure 12: Throughput of storage engines through SQL interfaces.



(a) Point lookup (b) Insert (c) Update

Figure 13: Response time of storage engines through SQL interfaces.

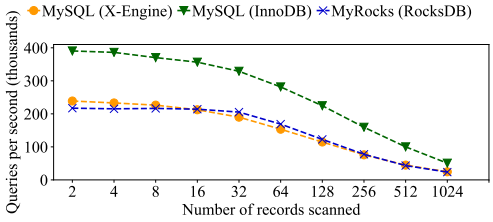


Figure 14: Throughput of range scans.

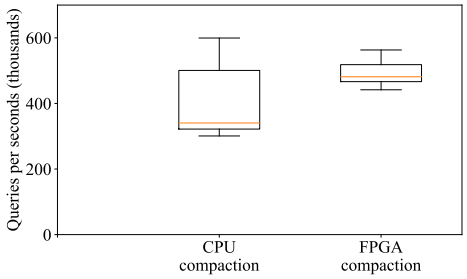


Figure 15: Throughput of MySQL (X-Engine) with and without FPGA-offloading for compactions.

4.4 The flood discharge problem

X-Engine contributes to the discharging of floods by improving the performance of compactions. Figure 15 compares the throughput of X-Engine with and without offloading them to FPGAs, running oltp_insert of SysBench. FPGA improves throughput by 27% with reduced variances. We show the CPUs saved by FPGA-offloading in Appendix A.

To evaluate the efficiency of X-Engine's data reuse during compactions, we prepare two runs of records, containing 50 million and 500 million records, respectively. All keys are distributed uniformly at random. We vary the percentage of distinct keys in the 50-million run from 70% to 99%. The rest contains groups of different versions of the same record with the same key. Figure 16 shows the throughput of RocksDB and X-Engine when compacting these two runs. When 90%, 99% of keys are distinct, X-Engine is 2.99 and 19.83 times faster, respectively. In these cases, the distribution of records with different versions of the same key are highly skewed, similar to what we observe in e-commerce workloads with strong hot spots. Keeping the small size of extents and data blocks (2MB, and 16KB, respectively) increases the number of reusable extents and contributes to the reduction of the compaction overhead.

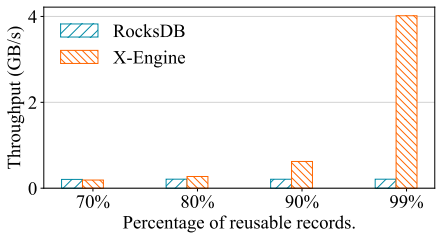
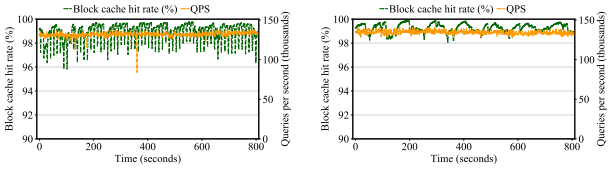


Figure 16: Throughput of compactions with different percentages of distinct records.

In Figure 17, we evaluate the block cache hit rates of X-Engine with and without the incremental cache replacement while processing the e-commerce workload (the 42:10:32:16 case with a high amount of writes). The incremental cache replacement has significantly reduced the variations of the block cache hit rates, and avoided occasional sudden drops of the QPS (about 36% decrease), contributing to the overall stability of X-Engine's performance. We acknowledge that



(a) Without incremental cache replacement. **(b) Wth incremental cache replacement.**

Figure 17: The block cache hit rates of X-Engine while processing the e-commerce workload.

operations like compactions and flushes still cause cache evictions and hurt cache hit rates on a regular basis.

4.5 The fast-moving current problem

The solution to the fast-moving current problem depends on the efficiency of caches, which determine how fast we can access a cold record (likely stored on disk) and buffer it in caches. In Figure 18, we vary the skewness of keys accessed by point queries according to a Zipf distribution, and measure the row and block cache hit rates as well as the QPS achieved. When accesses are uniformly distributed at random, almost all accesses miss the row cache because future queries may hardly access records cached. With more skewed accesses, some records become hot, and get cached in the row cache, increasing the row cache hit rate. These results show that the row cache performs well in highly skewed scenarios which are common in the e-commerce workload. Block cache is however less impactful for point queries.

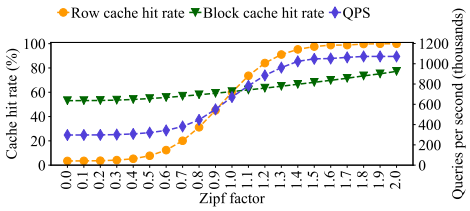


Figure 18: Impacts of the row and block caches with skewed lookups.

5 Related work

Storage engine is a critical component of any OLTP database system. The Amazon Aurora storage engine, developed from a fork of InnoDB, exercises parallel and asynchronous writes to reduce the latency and improve the throughput of writes [1, 32]. We improve this design principle with a multi-staged pipeline in X-Engine. Exploiting log-based structures is another efficient way to optimize the write performance [22, 23, 29]. The LSM-tree was proposed by O'Neil et al. [26], and has attracted many subsequent efforts to optimize its design [7, 18, 30]. Chandramouli et al. proposed a highly cache-optimized concurrent hash index with a hybrid log, supporting fast in-place updates [4]. Dong et al. optimized the space amplification in RocksDB with a careful study of related trade-offs [8]. RocksDB also incorporates a separate stage for WAL writes [12], and concurrent writes in the memtable [11]. In X-Engine, we adopt the idea of memtables from LevelDB [15], and propose a suite of optimizations to achieve highly concurrent fast writes that reduce the write amplification of LSM-trees significantly.

Many studies have optimized how records are merged in LSM-tree systems. Jagadish et al. divided a level into multiple sub-levels and merged them together in the next level during compactions [16]. Sears et al. explored merges at each level of the LSM-tree progress incrementally in a steady manner, reducing their negative impacts on the read performance [30]. Raju et al. proposed to only sort records within guards and allowing unsorted orders outside guards [27]. Teng et al. introduced a compaction buffer to maintain frequently visited data, reducing cache evictions caused by compactions [31]. Ren et al. redesigned the data block indexes for semi-sorted data in scenarios like graph-based systems [28]. Dayan et al. modeled the costs of tiering and leveling compactions, and improved the trade-off between update costs and point lookup/space costs [7]. X-Engine reduces the overhead of compactions through extensive data reuse that avoids significant unnecessary merges, and proposes to offload compaction to FPGAs for hardware-assisted accelerations.

To achieve efficient read operations, multiple data structures have been optimized in LSM-trees, including indexes [5, 14, 20, 21], caches [2, 9, 22], and filters [6]. X-Engine practices all these data structures in a hierarchical manner and extends indexes to all extents.

6 Conclusion

We introduce X-Engine, an OLTP storage engine, optimized for the largest e-commerce platform in the world at Alibaba, serving more than 600 million active customers globally. Online e-commerce transactions bring some distinct challenges, especially during the annual Singles' Day Global Shopping Festival when customers shop extensively in a short period of time. We design X-Engine on top of an optimized LSM-tree to leverage hardware acceleration such FPGA-accelerated compaction, and a suite of optimizations such as asynchronous writes in transactions, multi-staged pipeline, incremental cache replacement, and multiple kinds of caches. X-Engine has outperformed other storage engines processing the e-commerce workloads during online promotions and successfully served the annual Singles' Day. Our ongoing and future work includes adopting a shared storage design to improve its scalability for shared-nothing distributed transaction processing under sharding, and applying machine learning methods to predict data temperatures in order to facilitate intelligent scheduling of record placements in a tiered storage.

References

[1] Steve Abraham. 2018. Introducing the Aurora Storage Engine. https://aws.amazon.com/cn/blogs/database/introducing-the-aurora-storage-engine/.

[2] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. 2003. Cache Tables: Paving the Way for an Adaptive Database Cache. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29 (VLDB '03)*. VLDB Endowment, 718–729. http://dl.acm.org/citation.cfm?id=1315451.1315513

[3] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. 2007. Cache-oblivious Streaming B-trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '07)*. ACM, New York, NY, USA, 81–92.

[4] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 275–290.

[5] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. 2001. Improving Index Performance Through Prefetching. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD '01)*. ACM, New York, NY, USA, 235–246. https://doi.org/10.1145/375663.375688

[6] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Transactions on Database Systems* (2018).

[7] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 505–520.

[8] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB.. In *CIDR*, Vol. 3. 3.

[9] Klaus Elhardt and Rudolf Bayer. 1984. A Database Cache for High Performance and Fast Restart in Database Systems. *ACM Trans. Database Syst.* 9, 4 (Dec. 1984), 503–525. https://doi.org/10.1145/1994.1995

[10] Facebook. 2018. MyRocks. https://github.com/facebook/mysql-5.6/releases/tag/fb-prod201803.

[11] Facebook. 2018. RocksDB MemTable. https://github.com/facebook/rocksdb/wiki/MemTable.

[12] Facebook. 2018. RocksDB Pipelined Write. https://github.com/facebook/rocksdb/wiki/Pipelined-Write.

[13] Facebook. 2018. RocksDB Release v5.17.2. https://github.com/facebook/rocksdb/releases/tag/v5.17.2.

[14] Facebook. 2019. RocksDB: A persistent key-value store for fast storage environments. https://rocksdb.org/.

[15] Sanjay Ghemawat and Jeff Dean. 2011. LevelDB. *URL: https://github.com/google/leveldb,% 20http://leveldb. org* (2011).

[16] Goetz Graefe and Harumi Kuno. 2010. Self-selecting, Self-tuning, Incrementally Optimized Indexes. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT '10)*. ACM, New York, NY, USA, 371–381.

[17] Sándor Héman, Marcin Zukowski, Niels J. Nes, Lefteris Sidirourgos, and Peter Boncz. 2010. Positional Update Handling in Column Stores. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 543–554.

[18] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and Rama Kanneganti. 1997. Incremental Organization for Data Recording and Warehousing. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 16–25. http://dl.acm.org/citation.cfm?id=645923.671013

[19] Alexey Kopytov. 2019. Scriptable database and system performance benchmark. https://github.com/akopytov/sysbench.

[20] Tobin J. Lehman and Michael J. Carey. 1986. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB '86)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 294–303. http://dl.acm.org/citation.cfm?id=645913.671312

[21] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 38–49.

[22] Justin Levandoski, David Lomet, and Sudipta Sengupta. 2013. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *Proc. VLDB Endow.* 6, 10 (Aug. 2013), 877–888.

[23] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE, 302–313.

[24] MemSQL. 2019. Database Benchmark Tool. https://github.com/memsql/dbbench.

[25] MySQL. 2018. Introduction to InnoDB. https://dev.mysql.com/doc/refman/8.0/en/innodb-introduction.html.

[26] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[27] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 497–514.

[28] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A Space-efficient Key-value Storage Engine for Semi-sorted Data. *Proc. VLDB Endow.* 10, 13 (Sept. 2017), 2037–2048.

[29] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.* 10, 1 (Feb. 1992), 26–52.

[30] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 217–228. https://doi.org/10.1145/2213836.2213862

[31] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Siyuan Ma, Yanfeng Zhang, and Xiaodong Zhang. 2017. 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, 68–79.

[32] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 1041–1052.

[33] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. 2012. LogBase: A Scalable Log-structured Database System in the Cloud. *Proc. VLDB Endow.* 5, 10 (June 2012), 1004–1015.

A More evaluations

In Figure 19, we tune the number of connections to make X-Engine operating at the same QPS with and without FPGA-offloading, and then measure and report the CPU utilization of all processors. With and without FPGA-offloading, the CPU utilization averages to about 37 and 29 hardware threads (recall that our cpu caters to 64 threads in total with 32 cores and hyperthreading, so the maximum utilization is 6400%), respectively. FPGA-offloading also decreases the variance of CPU utilization by about six times by moving the computation overhead to the FPGA. By exploiting these saved CPU resources, X-Engine is able to increase the overall throughput by 27% as shown in Figure 15.

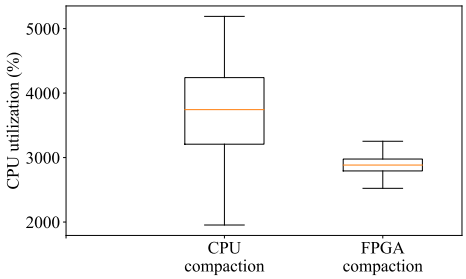


Figure 19: The CPUs usage of MySQL with X-Engine using CPUs or FPGAs for compactions while delivering the same level of throughput.

Figure 20 plots bytes read and bytes written per transaction while running write-intensive transactions, comparing RocksDB and X-Engine, two LSM-tree based systems. All data are normalized to the results of X-Engine. Because both RocksDB and X-Engine exercise indexes to accelerate point lookups, X-Engine's extents only reduces the read amplification by about 9%. However, such extents and indexes contribute to the data reuse in compactions, and help reduce the write amplification by 63%.

B Generating E-Commerce Workloads

X-Bench takes in a configuration of SQL workloads, where we specify several types of transactions. Each transaction mimics certain e-commerce operations such as browsing the hottest merchandises in a given category, placing orders, and making payments. To achieve this, we fill SQL templates into each transaction in the form like "UPDATE TABLE_NAME SET PAY_STATUS = STATUS WHERE ORDER_ID = ID;" where "TABLE_NAME", "STATUS" and "ORDER_ID" are placeholders for values to be filled in. By preparing different SQL templates, we are able to generate point lookups, range lookups, updates, and inserts in the workload. X-Bench further allows the configuration of the weight of each transaction, so that we can vary the percentage of each transaction in the workload to simulate different workloads. X-Bench then calls SysBench to populate such placeholders with values according to specified distributions and other conditions, and finally queries the target database instance with these synthesized queries. Assisted by Alibaba's database administrators and users, the SQL templates we use in our evaluations are based on real workloads from Alibaba's e-commerce platforms.

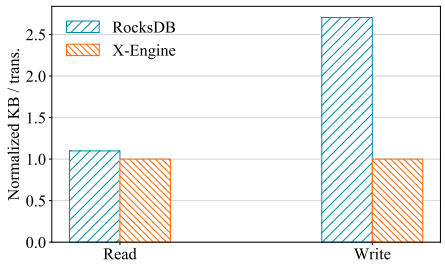


Figure 20: Normalized KB read or written per transaction.

C Identifying cold records

As discussed before, internal levels in warm/cold layer (shown in Figure 2) are differentiated by the temperature of data (extent). An extent's temperature is calculated by its access frequency in a recent window. When a compaction is performed, X-Engine selects the coldest extents with the number specified by a threshold, say 500 extents, and pushes these extents to the deeper level to do compaction. By doing so, X-Engine keeps the warm extents in upper levels and cold extents in deeper levels. But this method cannot handle dynamic workloads well. For example, when the current access frequency of an extent is low, our algorithm will treat the extent as cold data but it might become hot in near future. To this end, we have investigated machine learning based algorithms to identify the proper temperature of an extent. The intuition is that, in addition to extent, we also use row level (record) as a granularity to infer temperature (warm/cold). If a record has never been accessed in a recent window, it is identified as being cold. Otherwise, it is considered warm. So temperature identification is translated into a binary classification problem and can be solved using a classification model, such as using random forest or a neural network based approach. The details of our machine learning based algorithm for temperature identification is out of the scope of this paper and will be elaborated in another work.

D Configurations for Storage Engines and Benchmarks

We have open-sourced our scripts for evaluating InnoDB and RocksDB using SysBench and dbbench online [1]. We also show the configurations used for MySQL (InnoDB), MyRocks (RocksDB), and MySQL (X-Engine) in our evaluations in Table 2, Table 3, and Table 4, respectively. Table 2 and Table 3 contain well-known configuration options. Most options

[1] https://github.com/x-engine-dev/test_scripts

Options	Values
innodb_buffer_pool_size	256G
innodb_doublewrite	1
innodb_flush_method	O_DIRECT
innodb_flush_log_at_trx_commit	1
innodb_log_file_size	2 GB
innodb_thread_concurrency	64
innodb_max_dirty_pages_pct_lwm	10
innodb_read_ahead_threshold	0
innodb_buffer_pool_instances	16
thread_cache_size	256
max_binlog_size	500 MB
read_buffer_size	128 KB
read_rnd_buffer_size	128 KB
table_open_cache_instances	16

Table 2: Configurations of MySQL (InnoDB)

Options	Values
rocksdb_block_cache_size	170 GB
rocksdb_block_size	16384
rocksdb_max_total_wal_size	100 GB
rocksdb_max_background_jobs	15
rocksdb_max_subcompactions	1
target_file_size_base	256 MB
target_file_size_multiplier	1
level0_file_num_compaction_trigger	4
write_buffer_size	256 MB
max_write_buffer_number	4
max_bytes_for_level_multiplier	10
compression_per_level	No for all
num_levels	7 (default)
level_compaction_dynamic_level_bytes	True

Table 3: Configurations of MyRocks (RocksDB)

Options	Values
xengine_row_cache_size	45 GB
xengine_block_cache_size	170 GB
xengine_db_memtable_size	256 MB
xengine_db_total_memtable_size	100 GB
xengine_max_total_wal_size	100 GB
xengine_data_block_size	16384
xengine_max_compactions	8
xengine_max_flushes	3
xengine_max_memtable_number	2/sub-table
level0_extents_compaction_trigger	64
level1_extents_compaction_trigger	1000
xengine_compression	False
xengine_num_levels	3
xengine_thread_pool_size	128

Table 4: Configurations of MySQL (X-Engine)

in X-Engine are similar to those of RocksDB and InnoDB. "level0_extents_compaction_trigger" and "level1_extents_compaction_trigger" are the maximum number of extents in $Level_0$ and $Level_1$, respectively. Compactions are triggered when thees threshold values are reached.

Lastly, "xengine_max_compactions" and "xengine_max_flushes" refer to the maximum number of compaction threads and flush threads running in X-Engine, respectively.