Leaper: A Learned Prefetcher for Cache Invalidation in LSM-tree based Storage Engines

Lei Yang[1]*, Hong Wu[2], Tieying Zhang[2], Xuntao Cheng[2], Feifei Li[2], Lei Zou[1], Yujie Wang[2], Rongyao Chen[2], Jianying Wang[2], and Gui Huang[2]

{yang_lei, zoulei}@pku.edu.cn[1]

{hong.wu, tieying.zhang, xuntao.cxt, lifeifei, zhencheng.wyj, rongyao.cry, beilou.wjy, qushan}@alibaba-inc.com[2]

Peking University[1] Alibaba Group[2]

ABSTRACT

Frequency-based cache replacement policies that work well on page-based database storage engines are no longer sufficient for the emerging LSM-tree (*Log-Structure Merge-tree*) based storage engines. Due to the append-only and copy-on-write techniques applied to accelerate writes, the state-of-the-art LSM-tree adopts mutable record blocks and issues frequent background operations (i.e., compaction, flush) to reorganize records in possibly every block. As a side-effect, such operations invalidate the corresponding entries in the cache for each involved record, causing sudden drops on the cache hit rates and spikes on access latency. Given the observation that existing methods cannot address this cache invalidation problem, we propose Leaper, a machine learning method to predict hot records in an LSM-tree storage engine and prefetch them into the cache without being disturbed by background operations. We implement Leaper in a state-of-the-art LSM-tree storage engine, X-Engine, as a light-weight plug-in. Evaluation results show that Leaper eliminates about 70% cache invalidations and 99% latency spikes with at most 0.95% overheads as measured in real-world workloads.

PVLDB Reference Format:
Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. Leaper: A Learned Prefetcher for Cache Invalidation in LSM-tree based Storage Engines. *PVLDB*, 13(11): 1976-1989, 2020.
DOI: https://doi.org/10.14778/3407790.3407803

1. INTRODUCTION

Caches are essential in many database storage engines for buffering frequently accessed (i.e., hot) records in main memory and accelerating their lookups. Recently, LSM-tree (*Log-Structure Merge-tree*) based database storage engines have been widely applied in industrial database systems with notable examples including LevelDB [10], HBase [2], RocksDB [7] and X-Engine [14] for its superior write performance. These storage engines usually come with row-level and block-level caches to buffer hot records in main memory. In this work, we find that traditional page-based and frequency-based cache replacement policies (e.g., Least Recently Used (LRU) [28], Least Frequently Used (LFU) [33]) do not work well in such caches, despite their successes on B-Trees and hash indexes. The key reason is that the background operations in the LSM-tree (e.g., compactions, flushes) reorganize records within the storage periodically, invalidating the tracked statistics for the cache and disabling these replacement policies to effectively identify the hot blocks to be swapped into the cache.

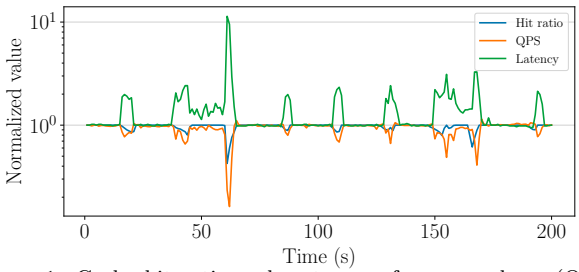


Figure 1: Cache hit ratio and system performance churn (QPS and latency of 95th percentile) caused by cache invalidations.

The root causes come from append-only and copy-on-write (CoW) techniques applied to accelerate inserts, updates and deletes, in conjunction with the mutable blocks in the storage layout of an LSM-tree. Although newly arrived records and deltas on existing records are appended into the main memory in the first place, eventually they need to be merged with existing record blocks in the durable storage through the flush and compaction operations in the LSM-tree. Because of these operations, traditional cache replacement policies that rely on tracking page/block level access frequency are no longer effective. Every time when a flush or compaction is executed in the LSM-tree, record blocks are reorganized and moved both physically and logically to or within the durable storage, along with changing key ranges for records and updated values and locations. This invalidates their corresponding entries and statistics in the cache and leads to cache miss for their lookups. Furthermore, compactions are usually executed multiple times for the same record due to the hierarchical storage layout of the

*Work performed while at Alibaba Group.

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407803

LSM-tree. In this work, we have identified that this problem often causes latency spikes due to the decreased cache hit rates. We refer to it as the *cache invalidation* problem.

Such cache invalidations happen frequently in workloads with intensive writes and updates, such as order-placement on hot merchandises in e-commerce workloads. Figure 1 shows an example where the cache misses caused by such invalidations leads up to 10× latency spikes and 90% queries per second (QPS) drops in X-Engine [14], a high-performance LSM-tree based storage engine at Alibaba and Alibaba Cloud. This level of performance instability introduces potential risks for mission-critical applications. Meanwhile, off-peak scheduling of compactions cannot maintain high performance under high stress, because it accumulates a large number of levels which lead to severe performance degradation in LSM-tree (e.g., the accrued range deletion operations have fatal performance reduction to range selection queries [35]). Beyond this, we aim to provide a general database service on the cloud, so that we cannot ignore the cache invalidation problem when users need high performance (i.e., under high stress).

The cache invalidation problem has attracted some research attention in the past [11, 1, 37]. They try to decrease the frequency of compactions by relaxing the sorted data layout of the LSM-tree [15], or maintain a mapping between records before and after compactions [1, 40]. Furthermore, they often require significant changes to the LSM-tree implementation. Hence, they either sacrifice the range query performance, or the space efficiency, or introduce significant extra overhead. These are often unacceptable because many industrial applications prefer general-purpose storage engines offering competitive performance for both point and range queries with high memory and space efficiencies.

In this paper, we introduce machine learning techniques to capture the data access trends during and after compactions that cannot be captured by existing methods. Our proposal introduces a small overhead without adding or altering any data structures in the LSM-tree. More specifically, we propose a learned prefetcher, Leaper, to predict which records would be accessed during and after compactions using machine learning models, and prefetch them into the cache accordingly. Our key insight is to capture data access trends at the range level, and intersects hot ranges with record block boundaries to identify record blocks for prefetching into the cache. We are enabled by machine learning models to find such hot ranges from the workload, which cannot be identified by conventional cache replacement policies. The identified ranges are independent of background operations, allowing Leaper to perform across multiple compactions or flushes continuously. And, our method naturally supports both point and range queries.

We design and implement Leaper to minimize both offline training overhead and online inference overhead. To this end, we have applied several optimizations in the implementation such as the *locking mechanism* and the *two-phase prefetcher*. We have evaluated Leaper using both synthetic and real-world workloads. Results show that Leaper is able to reduce cache invalidations and latency spikes by 70% and 99%, respectively. The training and inference overheads of Leaper are constrained to 6 seconds and 5 milliseconds, respectively. Our main contributions are as follows:

- We formulate the cache invalidation problem, and identify its root causes in the modern LSM-tree storage engines, which existing methods cannot address. We have proposed a machine learning-based approach, Leaper, to predict future hot records and prefetch them into the cache, without being disturbed by background LSM-tree operations that cause the cache invalidation problem.
- We have achieved a low training and inference overhead in our machine learning-based proposal by carefully formulating the solution, selecting light-weight models for predictions and optimizing the implementation. The overall overhead is often as small as 0.95% (compared to the cost of other normal execution operations excluding Leaper) as observed in real-world workloads. We have extracted effective features, achieving a high level of accuracy: 0.99 and 0.95 recall scores for synthetic and real-world workloads, respectively.
- We have evaluated our proposal by comparing it with the state-of-the-art baselines using both synthetic and real-world workloads. Experimental results show that Leaper improves the QPS by more than 50% in average and eliminates about 70% cache invalidations and 99% of latency spikes, significantly outperforming others.

The remainder of this paper is organized as follows. Section 2 introduces the background and formulates the cache invalidation problem. Section 3 presents our design overview of Leaper. We introduce details of Leaper's components in Sections 4, 5 and 6. We evaluate our proposal in Section 7 and discuss related works in Section 8. At last, we conclude in Section 9.

2. BACKGROUND AND PRELIMINARY

2.1 LSM-tree based Storage Engines

LSM-tree based storage engines have acquired significant popularity in recent years. Notable examples include LevelDB [10] from Google, RocksDB [7] from Facebook and X-Engine [14] from Alibaba, supporting applications such as Chrome [17], LinkedIn [8], and DingTalk [14]. This popularity is driven by the trend that there are increasingly more writes (e.g., inserts, updates) in database workloads, where the traditional B-tree based storages struggle to offer the expected performance at a reasonable space cost.

LSM-tree is designed to achieve a high write throughput. Figure 2 illustrates the generic architecture of a LSM-tree, consisting of a memory-resident component and a disk-resident component. Incoming records are inserted into *active memtables* in the main memory, which are implemented as skiplists in many systems [31, 14]. To update an existing record, the corresponding delta is inserted into this active memtable in the same way. This append-only design ensures that all writes other than logging are completed in the main memory without going into the durable storage where the access latency is much higher. When an active memtable is filled, it is switched to be an *immutable memtable*. As memtables accumulate in the main memory, approaching the main memory capacity, **flush** operations are triggered to flush some immutable memtables into the durable storage where incoming records are merged with existing ones. Such a merge may incur a lot of disk I/Os. And, the same record may be merged multiple times, causing write amplifications.

To bound such write amplifications and to facilitate fast lookups over recently flushed records which are still very

likely to be hot due to locality, the disk component of the latest LSM-tree storages adopts a tiered layout consisting of multiple levels storing a sorted sequence of extents. An extent packages blocks of records as well as their associated filters and indexes [14]. Each level is several times larger than the one above it. Flushed records first arrive in the first level $L0$. When $L0$ is full, parts of it are merged into the next level $L1$ through **compactions**. In the meantime, compactions also remove records marked to be deleted or old versions of records that are no longer needed and then write back the merged records in a sorted order into the target level.

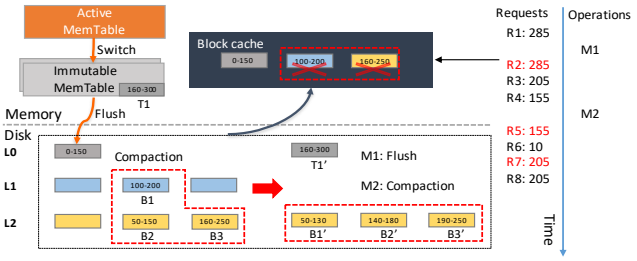


Figure 2: General architecture of LSM-tree based storage engines and examples of cache invalidations caused by flush and compaction.

The above-introduced write path achieves a high write throughput at the cost of the lookup performance. Although many lookups for newly inserted records can be served by the memtables in the main memory, the rest have to access all those levels in disk. And, to access a frequently updated record, a lookup has to merge all its deltas scattered across the storage to form the latest value. Even with proper indexes, this lookup path can be very slow due to the disk I/Os involved. A range query, even worse, has to merge records satisfying its predicates from all levels in the disk.

To resolve the slow lookups, many LSM-tree storage engines have incorporated caches in the main memory to buffer hot records. In Figure 2, we have illustrated a block cache buffering three record blocks from the disk. To maintain these caches, traditional frequency-based cache replacement policies like LRU are usually enforced to swap entries in and out of those caches. These policies work well when there is a clear and stable level of locality in the workload that they can track, however, flushes and compactions in the LSM-tree often disable such policies, which we introduce in the following.

2.2 Cache Invalidation Problem

In database storage engines, caches are essential for buffering frequently accessed (i.e., hot) records, and cache replacement policies are exploited to effectively identify hot blocks that should be swapped into the cache. Cache replacement problem can be formulated as follows:

FORMULATION 1. ***Cache replacement problem.*** *Given a set of records $R = \{r_0, r_1, \cdots, r_{n-1}\}$ based on a sequence of latest requests, and a database cache C that could buffer L records, find an algorithm to determine L ordered records in cache such that the probability $P(r_n \in C)$ for the coming request hitting the cache is maximized, where r_i and L refer to a record, and cache size respectively, $i, L \in N$.*

In most real-life scenarios, traditional cache replacement policies (e.g., LRU [28]) work well. The cache hit ratio can be kept up to 99% in the cases without compactions or flushes. However, in a LSM-tree based storage engine, background operations either flush blocks from memtable to the disk or reorganize blocks within the disk, causing the original blocks in the cache inaccessible. Further, potential retrievals of the inaccessible blocks miss the cache. We define this as the cache invalidation problem:

FORMULATION 2. ***Cache invalidation problem.*** *Given a database cache $C = \{r_0, r_1, ..., r_{L-1}\}$ determined by a cache replacement policy, and a set of records participate in a compaction (or flush) $M_i = \{r_i^0, r_i^1, \cdots, r_i^{n_i-1}\}$, the invalidated cache can be represented as $|C \bigcap M_i|$, where M_i and n_i refer to a compaction (or flush) and the number of records moved by it, $n_i, i \in N$. The cache invalidation problem is defined to minimize the invalidated cache for a compaction or flush.*

Examples. Figure 2 illustrates an example of the formulated problem by showing the status of blocks before and after a flush and a compaction. The right side gives a sequence of requests and a set of operations over time. The requests marked as black represent cache hits while the requests marked as red represent cache misses. At the beginning, flush $M1$ moves block $T1$ from the immutable memtable to the disk, and causes request $R2$ to miss the cache. In the following, compaction $M2$ reorganizes blocks $B1$, $B2$ and $B3$ (i.e., selected by the red dashed box) to blocks $B1'$, $B2'$ and $B3'$, which invalidates cached blocks $B1$ and $B3$. Therefore, it causes two cache misses of request $R5$ and request $R7$. Request $R8$ hits the cache because LRU swaps block $B3'$ into the cache after $R7$.

Existing solutions. Several methods were proposed by early works. Stepped-Merge tree (SM-tree) [15] is an LSM-tree variant where records are not fully sorted to decrease the frequency of compaction, and therefore reduce cache invalidations. However, it significantly degrades read performance when executing range queries or frequently-updated workloads.

LSbM [40] combines the idea of SM-tree [15] and bLSM [37], and aims to maintain the reference between cache and disk during compaction. Unfortunately, it increases the burden of compaction and brings storage overhead. Also, it degrades read performance when executing range queries as Stepped-Merge tree.

Incremental Warmup Algorithm [1] builds mappings between data before and after compaction through pre-sorted smallest keys. It moves newly compacted blocks sequentially into block cache whose key ranges overlap with blocks in the block cache. Before they are moved, the blocks in the block cache are evicted. However, it has two disadvantages. First, it assumes that newly compacted blocks are supposed to be frequently visited if they overlap with any blocks in the block cache. Second, blocks in the cache may overlap with more than one newly compacted block, so it might prefetch infrequently requested blocks into the block cache.

3. DESIGN OVERVIEW

3.1 Design Rationale

Existing methods are not sufficient to solve the cache invalidation problem formulated above. First, SM-tree [15] aims to delay compactions (i.e., reducing the amount of compaction M), which significantly degrades read performance. Second, LSbM [40] and Incremental Warmup [1] attempt to restore the cache C. However, compactions disable the

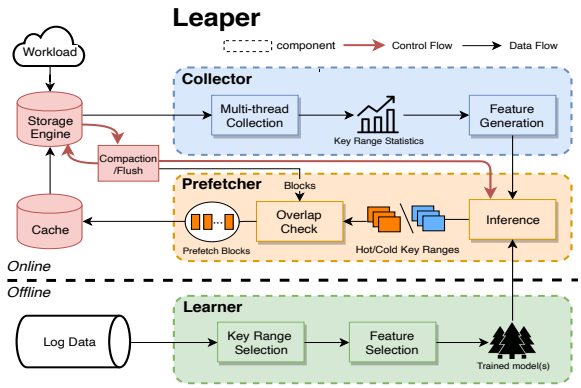


Figure 3: Workflow of LSM-tree storage engine with Leaper.

cache replacement policies to track the access frequency of each block, so that the restored cache is ineffective for the cache invalidation problem.

Fundamentally, the cache invalidation problem can be addressed by minimizing $|C \bigcap M_i|$ during each compaction (or flush). It requires us to predict which records are accessed in the near future, and then swap them into the cache in advance. It is a prediction problem that predicts the future access of each record in the set $|C \bigcap M_i|$. The prediction problem can be formally defined as a binary classification problem: given the access of a record in the latest $x \cdot t$ minutes, predict whether this record is accessed in the following T minutes, where t, x, and T refer to a statistical time interval, number of intervals, and the prediction duration. This prediction method can address the cache invalidation problem because of the following reasons. First, it achieves a higher cache hit ratio than LRU theoretically because it can detect more hot records [3]. Second, access frequency of each record tracked by the prediction method is not disturbed by compactions because it is independent of the storage structure. Third, predicting the access of records in the cache is naturally a binary classification problem because we only need the first access of records to eliminate potential cache misses.

Other formulations for the prediction problem, such as sequence prediction and time series regression, are not suitable in our case. The sequence prediction approach collects a long historical record access sequence for a past period and makes predictions on the future access sequence. The computation and memory overhead incurred in this process are more than necessary for our problem because we only need to make predictions on the invalidated cached entries, and the information on the rest is not necessary, given that policies like LRU already performs reasonably well. For the time series regression approach, the transformation from regression to classification causes considerable accuracy loss because most workloads have typical hot spots.

3.2 System Overview

To achieve the binary classifier introduced above, we introduce machine learning models to predict record accesses during and after compactions. Specifically, we predict at the key range level, aiming at a good trade-off between the prediction overhead and accuracy. With such hot ranges predicted, we intersect them with block boundaries to select record blocks that should be prefetched into the cache.

Figure 3 shows the workflow of our proposal, Leaper, consisting of three major parts: **Collector**, **Prefetcher**, and **Learner**. The bottom part shows the **Learner** component which is responsible for training predictive models with different time scales. The higher part shows the **Collector** component which generates featurized data and the **Prefetcher** component which interacts with the flush operation and the compaction operation and fills appropriate data blocks into the cache. The overall system can be easily plugged into the LSM-tree storage engine, shown on the left of Figure 3, without altering any existing data structures.

The learner extracts the access data from query logs, transforms the data into the format for training classification models. To reduce the overhead, we group adjacent keys into key ranges. Leaper selects the right size of key ranges according to different workloads to achieve a good system and model performance. After that, the learner trains multiple models for Leaper to predict the future access of key ranges with different time scales. We use tree-based models for classification to achieve accurate prediction.

The collector collects data in a multi-threaded way. We introduce optimized lock mechanisms to avoid write conflicts in the collector. We also introduce a sampling strategy to reduce the overhead of the collector. After that, the collector obtains the access frequency of several time intervals for different key ranges and transfers them into counting features for further prediction.

The prefetcher first predicts the access of key ranges using the features from the collector and the trained models from the learner. All blocks that predicted to be accessed from those participating in the compactions (or flushes) are handled by the Overlap Check module. Finally, the prefetcher takes actions to either insert new blocks into the cache or evict old blocks from the cache. The prefetches happen along with the processing of flush and compaction operations.

Figure 3 also depicts the control flow between the storage engine and Leaper. In a running storage engine, the collector keeps updating the access statistics. The learner trains the models periodically, depending on the changes of the workload. It outputs trained models for the prefetcher to do prediction as necessary. When the flush and the compaction operations are internally triggered by the storage engine, the prefetcher begins to run. For flush, the prefetcher directly predicts the future access of key ranges that involved in the flush. For compaction, it usually takes a long time period. We use a two-phase mechanism to predict the accesses of the key ranges that participate in the compaction.

4. OFFLINE ANALYSIS

In this section, we describe the offline component of Leaper which is responsible for data featurization and model training. We first introduce the *key range selection* to reduce the system overhead and make our approach feasible for the online components. Next, we discuss the features for the prediction problem. Last, we describe the classification models and the training methodology in Leaper.

4.1 Key Range Selection

In a running LSM-tree based storage engine, we are unable to track the access frequency of blocks because they are reorganized frequently by compactions. So we introduce the key range that is independent of compactions. The use of key ranges has three advantages. First, key ranges help reduce the overhead of offline training and online inference.

Algorithm 1: Key Range Selection

Input: Total key range T, initial size A , access information and decline threshold α
Output: Most suitable granularity A^*

```
Initialize access matrix M (N × T/A bits for N time intervals and T/A key ranges) for key range size A;
Define the number of zeroes in M as Z(A);
while 2 · Z(2A) > α · Z(A) do
    A ← 2A;
Binary search to find the maximum value A* satisfying A* · Z(A*) > αA · Z(A) from A to 2A;
return A*
```

Second, key ranges are consistent with the layout of blocks in the underlying LSM-tree, whether the key is a primary key or a secondary key. So that key ranges are efficient to check overlap with blocks. Third, statistics for access frequency of range queries are easy to collect if key ranges are used.

The key range size has significant impact on both the prediction accuracy and the overhead of the online component of Leaper. For a given storage filled with records, the key range size determines the number of key ranges and the size of statistics per key range to be collected online by the *Collector* in our design. Reducing such key range size results in more detailed statistics, and potentially a higher level of prediction accuracy for prefetch, with increasing online collection overhead.

We initialize the size of key ranges with a small value A (we use 10 in our experiments). For each key range, we use a binary digit (i.e., 1 or 0) to indicate whether it is accessed (or predicted to be accessed soon) or not. With this method, it takes a vector of N bits to store such access information for N key ranges in a single time interval. For a period time with multiple time intervals, we extend a vector into a matrix, with one row in the matrix corresponding to one time interval. We take a vector of 4 bits (0,1,1,1) for example. If we expand the key range size twice, the access vector forms logical add (i.e., $1 + 0 = 1$, $1 + 1 = 1$ and $0 + 0 = 0$) to become (1,1), and therefore the size of it shrinks to the half. When we expand the key range size, the access information loss occurs as long as the proportion of zeros in vector (or matrix) declines.

Information loss does not necessarily lead to performance penalties for the prediction. Through experimental analysis, we propose the concept of *Efficient expansion*:

Definition 1. ***Efficient expansion.*** *An expansion of key range size from L to L' ($L' > L$) is efficient, if and only if $Z(L')/Z(L) \geq \alpha$, where Z is the proportion of zeros in the access vector (or matrix), and α is a threshold based on the initial proportion of zeros and the decline ratio of prediction.*

Such expansion is efficient because associated information loss can be compensated by the learning ability of the model, so the prediction results only have a slight decrease (e.g., no more than 0.1% in recall). Therefore, we can keep expanding the key range size until the expansion is not efficient anymore. In our case, the threshold α is set to 0.6, and the final key range size is expanded to ten thousand.

Formally, Algorithm 1 depicts the procedure for computing the range size. It follows the idea of binary search to find the maximum value A^* satisfying the definition of *Efficient expansion*. By using this algorithm, Leaper selects an appropriate range size to group keys together.

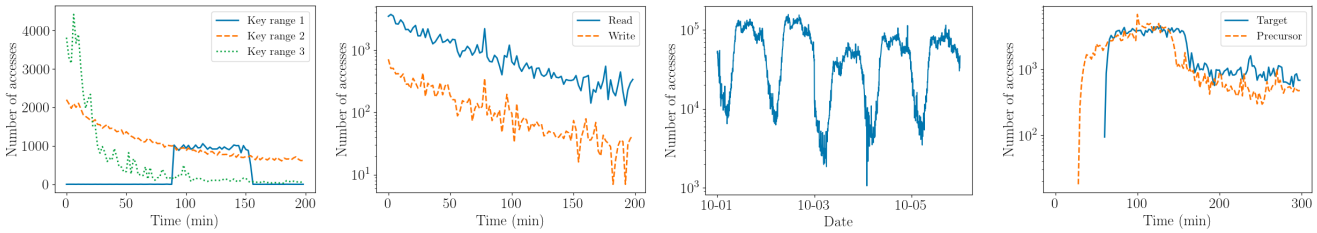
4.2 Features

In the storage layer, the access key and the access timestamp of each request are usable access information. Our challenge is to make use of them to build a classification model and predict future accesses. Another requirement is to select the most profitable features to achieve good accuracy (e.g., more than 95% in recall) and introduce small overhead (e.g., no more than 1% overall). Since data in LSM-tree based storage engines are commonly represented as key-value pairs for high performance, the query information (i.e., the structures and semantic information of query strings [23]) is unavailable in the storage layer. Based on above considerations, we perform feature engineering to transfer the usable access information into a set of features that guarantee both precision and efficiency. This section explains how we select the features.

Read/write Arrival Rate. To model the access patterns of the workload, we exploit the concept of *Arrival Rate* [34]. The read (or write) arrival rate means the number of reads (or writes) in successive time slots. Usually, the time slot is one or two minutes. Key ranges have different access patterns, as shown in Figure 4(a). Therefore, the read arrival rate is the most important feature for the model to capture the access patterns because it can reflect the user behavior at the application level. Figure 4(b) tells us for some key ranges, the write arrival rate shares similar access patterns with the read arrival rate. In other words, popular items have both frequent browses and orders in the scenario of e-commerce. So it is necessary to add the write arrival rate as a feature. In our implementation, we use 6 time slots (explained in Section 7) to collect the arrival rate. So there are 12 features in total for both the read arrival rate and the write arrival rate.

Prediction Timestamp. Figure 4(c) depicts the number of accesses on a table during five consecutive days. It tells us that for the e-commerce workload, the access of a table has periodical patterns with peaks and bottoms. Since many real workloads have time-dependent characteristics [23], it's important to add the timestamp at prediction time(i.e., prediction timestamp) as a feature. In our experiment, we use 3 features (i.e., hour, minute and second of the day) to capture the access patterns of key ranges because we find most of our testing workloads have repeated access patterns in a day. But it is reasonable to extend the prediction timestamp with more comprehensive dimensions such as year, month, and day.

Precursor Arrival Rate. A precursor of a target key range is another key range that after whose accesses, the target key range's accesses follow. So the precursor arrival rate means the number of a precursor's reads in successive time slots. As shown in Figure 4(d), the target key range (i.e., the blue line) shares a similar access pattern with the precursor (i.e., the orange line). On one hand, it tells us that the access of a target key range might be affected by its precursor(s), and the correlation between key ranges in storage engines corresponds to the correlation between behaviors in real applications. For example, in the e-commerce workload, the probability for a user to purchase a piano rack increases after he purchases a piano. On the other hand, the target key range may have different numbers of precursors, so we need to capture this kind of transition patterns across different key ranges. We use their arrival rates in multiple time slots as a vector to calculate the *cosine similarity* and



(a) Read arrival rate (b) Read and write arrival rate (c) Temporal Feature (d) Precursor arrival rate

Figure 4: Access patterns in e-commerce scenario. (a) shows read arrival rate of different key ranges have different patterns; (b) shows read arrival rate and write arrival rate of the same key range share similar pattern; (c) shows e-commerce workload has temporal periodicity; (d) shows many key ranges share similar patterns with their precursors.

select the most γ similar key ranges as the precursors of the target key range. Algorithm 2 depicts the process. Using the algorithm, we add the arrival rates of γ most similar precursors in the last one slotted time interval into the features. In our experiment, γ is set to 3 so it adds 3 features to the model.

Algorithm 2: Calculation of Precursors

Input: M historical accesses $\{(k_i, t_i)\}_{i=1}^{M}$, similarity length γ and threshold ϵ
Output: Precursors $P = \{(k_i, \{p_1, p_2 ... p_\gamma\})\}_{i=1}^{n}$ for all key ranges

`/* Generate key range transfer matrix` T `*/`
for *each key range in historical accesses* **do**
  for j *from 1 to* γ **do**
    $T[k_i][k_{i-j}] \leftarrow T[k_i][k_{i-j}] + 1$;
`/* Access similarity judgement */`
Calculate read arrival rates for key ranges;
for *each key range* k_i **do**
  $count \leftarrow 0$;
  for *item* k_j *in sorted*$(T[k_i])$ **do**
    while $count < \gamma$ **do**
      if *cosine similarity* $cos\theta = \frac{\vec{V}_{k_i} \cdot \vec{V}_{k_j}}{\|\vec{V}_{k_i}\| \times \|\vec{V}_{k_j}\|} > \epsilon$ **then**
        $P(k_i).add(k_j)$;
        $count \leftarrow count + 1$;
return P

Other features. We also evaluate additional features such as query type and cache state. The results prove that adding the query type does not contribute to the prediction metrics. We also consider using the block ids in the current cache to present the cache states. However, this introduces huge extra overhead of the feature size, since the cache is usually tens of GB containing millions of block ids. Furthermore, the cache states change intensively in every second so it is not feasible to collect such data.

4.3 Model

We use Gradient Boosting Decision Tree (GDBT) [9] as the classification model due to its accuracy, efficiency and interpretability. It produces a model in the form of an ensemble of weak prediction models (decision trees). GBDT is widely used in industry and is often used for tasks such as click-through rate prediction [32] and learning to rate [5]. For every feature, GBDT needs to scan all the data instances to estimate information gain of all the possible split points. Thus, the computational complexity of GBDT is proportional to both the number of features and the number of data instances. We select LightGBM, the novel implementation of GBDT, for its excellent performance in computational speed and memory consumption [16].

Other kinds of machine learning models, including neural networks, have also been tried, but are infeasible in our case. For example, LSTM [13] suffers from low precision and long inference time because of the following reasons. First, since the access distribution of the key ranges is highly skewed, it is difficult to apply normalizations. Therefore, LSTM could not distinguish the negative examples from the positive examples with a limited number of accesses. Second, the time to do hundreds of inferences in one compaction is about 1-2 seconds, which is unbearable in the online transaction processing (OLTP) database system.

The input of the classification model is a feature vector with 18 dimensions (i.e., 6 read arrival rates, 6 write arrival rates, 3 timestamp features, and 3 precursor arrival rates). The output is a binary digit (i.e., 1 or 0) indicating whether this key range would be accessed soon (i.e., one slotted time interval). The loss functions we use are *Square Loss* and *Logarithmic Loss*. For model training, we generate training set and testing set from the e-commerce workload. We exploit *GridSearchCV* function from scikit-learn [30] to search for the optimal parameters on testing set. The main parameters we tune are *num_leaves, learning_rate, bagging_fraction* and *feature_fraction*, which help avoid over-fitting. K-fold cross validation also helps determine the ultimate parameters. Finally, *num_leaves*, *learning_rate*, *bagging_fraction*, and *feature_fraction* are set to 31, 0.05, 0.8 and 0.9, respectively. We train one global model for different key ranges. Since new key ranges would be generated in a running storage engine, one global model for all key ranges has better generalization ability than multiple models for different key ranges. Also, it can help reduce inference overhead.

5. ONLINE PROCESSING

In this section, we discuss how to collect statistics, inference and decide which blocks need to be prefetched in the online components of Leaper.

5.1 Statistics Collection

The collector maintains a global counter for each key range. In multi-threaded storage engines, if records belonging to the same key range are accessed by different threads, concurrent updates of the global counter for the key range causes the write conflicts. Therefore, it is necessary to apply locking mechanisms but minimize the extra overhead in the collector. We adopt two strategies in the design of locking mechanisms to reduce the overhead. First, we use double-checked

locking [36] and lazy initialization for initializing key ranges counters. Lazy initialization avoids initializing a key range until the first time it is accessed. Double-checked locking reduces the overhead of acquiring a lock by checking the criterion before acquiring the lock. Second, we use atomic operations in collecting the access statistics instead of a global mutex. Because a global mutex has significant impact on the system performance, as shown in Table 1. We use the decline rate of QPS to present the influence of strategies on system performance. Through our locking strategies, the decline ratio can be reduced from 40.61% to 17.21%.

Table 1: Influence of strategies

Strategies	avg QPS (k/s)	Decline rate
Raw	337.7	-
Global mutex	200.5	40.61%
Double-Checked+Atomic	279.6	17.21%
Double-Checked+Atomic +Sampling	325.4	3.66%

Further, we use sampling to reduce the overhead. Specifically, lazy initialization and double-checked locking guarantee the first access of each key range is collected in the counter and then the following accesses of the key range are recorded with the probability of P. Therefore, the estimated accesses of a key range $\hat{N}_i$ can be calculated by the collected value S_i and the sampling probability P:

$$\hat{N}_i = \frac{S_i - 1}{P} + 1 \tag{1}$$

Then, we compute the sampling error as follows:

- S_i in probability-sampling obeys the binomial distribution [41]:
$$S_i - 1 \sim B(N_i - 1, P),$$
- At the same time, the binomial distribution can be considered as the normal distribution approximately:
$$S_i \sim N((N_i - 1)P + 1, (N_i - 1)P(1 - P)),$$
- As a result, the sampling error could be described as:
$$|\hat{N}_i - N_i| \leq z_{\alpha/2}\sqrt{\frac{(N_i - 1)(1 - P)}{P}},$$
where $z_{\alpha/2}$ means standard score in Normal Distribution Tables with significance level of α.

From the results, although on average the sampling causes approximately 16.3% error rate for access statistics, it has a minor influence on the predictions (see Section 7). Table 1 shows that it helps reduce the decline rate to 3.66%.

5.2 Inference

Through inference, we divide all involved key ranges into hot key ranges and cold key ranges using the featurized data from the *collector* and the trained model from the *learner*. A hot key range means this key range is predicted to be accessed in the near future while a cold key range means the opposite.

In Leaper, we use the Treelite [6] as the inference implementation to further reduce inference overhead. There are three major considerations for using Treelite. First, it uses compiler optimization techniques to generate model-specific and platform-specific code, which includes Annotate conditional branches, Loop Unrolling, etc. Treelite achieves 3-5× speedup on the original implementation of lightGBM. Second, we use dynamic linking library to integrate the model inference logic into the storage engine. Without recompiling the inference code, we only need to update the trained model by simply replacing the dynamic library generated from the learner. Third, it supports multiple tree models such as Gradient Boosted Trees and Random Forests from different implementations (XGBoost, LightGBM, Scikit-Learn, etc). These properties are very important for us to compare Leaper with others. And it is flexible for Leaper to support many models from different training libraries. In the experimental section, we test and verify the effectiveness of Treelite and LightGBM through the cost of inference.

5.3 Overlap Check

After inference in the prefetcher, we need an overlap check between the key ranges generated by *Key Range Selection* and the target blocks moved by the compaction and flush operations to figure out which target blocks are hot.

Algorithm 3: Check Overlap Algorithm

Input: Target blocks $\{(A_i, B_i)\}_{i=1}^{m}$, hot key ranges $\{(a_j, b_j)\}_{j=1}^{n}$
Output: Prefetch Data T
`/* Binary Search:` $O(nlogm) < O(m)$ `*/`
$start = A_1, end = A_m$;
for a_j *in hot key ranges* **do**
  Binary Search for $A_i \leqslant a_j < A_{i+1}$ from start to end;
  while $b_j > A_{i+1}$ **do**
    if $Min(B_i, b_j) \geqslant a_j$ **then**
      T.Add((A_i, B_i));
    $i \leftarrow i + 1$;
  $start = A_{i+1}$;
`/* Sort-merge:` $O(m) \leqslant O(nlogm)$ `*/`
for A_i *in target blocks and* a_j *in hot key ranges* **do**
  if $Min(B_i, b_j) \geqslant Max(A_i, a_j)$ **then**
    T.Add((A_i, B_i));
  if $B_i < b_j$ **then**
    $i \leftarrow i + 1$;
  else if $B_i > b_j$ **then**
    $j \leftarrow j + 1$;
  else
    $i \leftarrow i + 1$, $j \leftarrow j + 1$;
return T

We pose a *Check Overlap Algorithm* to check whether target blocks would be prefetched as Algorithm 3. There are two ways to check the overlap between predicted hot key ranges and target blocks, **binary search** and **sort-merge**. We choose one based on when the prefetcher gets the start key and the end key of target blocks. If we get it at the end of flush or compaction, **sort-merge** is invoked. Otherwise, if we get it during flush or compaction, **binary search** is invoked. If we get the start key and the end key of target blocks both during and after flush or compaction, which one is better depends on how many orders of magnitudes exist between m and n, where m means the number of target blocks and n means the number of predicted hot key ranges. In our case, m and n change dynamically in different flush or compaction operations. So we adopt a hybrid algorithm combining both of them to reduce the running time.

Other technologies like *Date reuse* [14] and *Bloom Filter* [4] are exploited to reduce the errors of *Check Overlap Algorithm*.

6. OPTIMIZATIONS FOR COMPACTION

We propose *Multi-step Prediction* and *Two-phase Prefetcher* to help the prefetcher deploy in the compactions according to the following requirements of the compaction operation. First, corresponding cached entries of moved blocks need to be dealt with in two phases (i.e., during and after the compactions) to reduce the cache misses. If they are accessed during the compaction, they should be preserved in the cache until the end of the compaction. Otherwise, they should be evicted to make use of the cache. Second, since compactions have different lengths, the prediction for two phases need to be combined by multiple steps.

The *Multi-step Prediction* helps Leaper to predict the future access of key ranges in a fine-grained way. Then *Two-phase Prefetcher* is used to distinguish the accesses during and after the compaction. In the first phase, which refers to the eviction phase, Leaper predicts the accesses during the entire compaction operation. In the second phase, which refers to the prefetch phase, Leaper predicts the accesses in an estimated time period after the compaction operation. Also, *Two-phase Prefetcher* cooperates with the cache replacement policy (LRU) in two ways. First, Leaper only evicts the invalidated blocks which are predicted as cold data. It means the evicted blocks are not requested any more and should be evicted by LRU soon. Second, since the size of blocks prefetched is relatively small compared to the total cache size (i.e., usually tens of GB in real-life applications), those blocks should not be evicted by LRU nor break the arrangement of LRU as long as they are accessed soon.

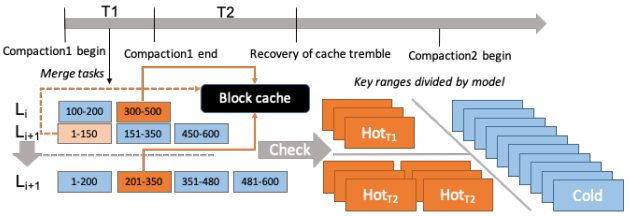


Figure 5: The design of the prefetcher for compaction.

The details of the optimizations are depicted in Figure 5. At the beginning of a compaction, multi-step prediction models trained offline are used to distinguish hot and cold key ranges. Then, Leaper combines the models to predict hot key ranges in two phase. Finally, target blocks in two phases are evicted or prefetched respectively.

6.1 Multi-step Prediction

Since compactions have different time lengths, we divide a compaction into multiple steps. A step restricts the slotted time interval mentioned in Section 4. Specifically, the slotted time interval is bounded by T_1 and T_2 in Figure 5 where T_1 means the duration of a compaction, and T_2 means recovery time of cache hit ratio decline caused by cache invalidations.

In theory, the slotted time interval t is the greatest common divisor of T_1 and T_2. Under most practical circumstances, recovery time T_2 is far less than compaction time T_1. Therefore, t approximately equals to the smallest T_2 in practice. Since t is determined, given n compactions, the number of steps k can be calculated as follows:

$$k = \lceil \frac{Max\{(T_1+T_2)^1, \cdots, (T_1+T_2)^n\}}{t} \rceil + C, \quad (2)$$

where C is a constant to handle the case when an online compaction lasts longer than all compactions in the training phase. Each step corresponds to a prediction model trained using the method of Section 4. So *Multi-step Prediction* contains k models to predict whether key ranges would be accessed in the next k slotted time intervals. Although we train C more models in learner, it introduces no additional online overhead because in most scenarios, these models are not used. Ultimately, these k models are provided for *Two-phase Prefetcher* to do prediction as necessary.

6.2 Two-phase Prefetcher

Models provided by *Multi-step Prediction* cannot be used directly because compaction has different T_1 and T_2. We first need to estimate the value of T_1 and T_2. According to our analysis, T_1 and T_2 approximately satisfy the following relations respectively:

$$\begin{cases} T_1 \approx \alpha N, \\ T_2 \approx \beta \frac{Q}{S}, \end{cases} \quad (3)$$

where N means the number of blocks needed to merge, Q means real-time QPS and S means the size of block cache. Both α and β are constants that can be computed by sampling from previous log data.

Then we combine k slotted time intervals into T_1 and T_2 to form a two-phase binary classification. Hot key ranges for T_1 and T_2 can be combined as follows:

$$\begin{cases} hot_{T1} = hot_{1t} \cup \cdots \cup hot_{k_1 t}, \\ hot_{T2} = hot_{(k_1+1)t} \cup \cdots \cup hot_{(k_1+k_2)t}, \end{cases} \quad (4)$$

where hot_{T1} means k_1 hot key ranges for T_1, hot_{T2} means k_2 hot key ranges for T_2 and hot_{it} means hot key ranges for the ith slotted time interval.

At last, we use the two-phase binary classification to perform eviction or prefetch. Since the evictions and prefetches are scattered in many merge tasks of a compaction, we take one merge task shown in the left part of Figure 5 as an example to describe the operations. In this case, block [300,500] from $Level_i$ and block [1,150] from $Level_{i+1}$ are swapped into cache before merging. After they check overlap with hot_{T1}, we know block [1,150] is predicted to have no visit before the end of compaction. So we evict block [1,150] and keep block [300,500] in the block cache. Also, the other three blocks are checked as well to make sure we do not omit any blocks to be accessed. After this merge task, two blocks from $Level_i$ and three blocks from $Level_{i+1}$ are merged to $Level_{i+1}$. It is important to note that during merge tasks, blocks are loaded into memory and we need no extra storage overhead to get exact blocks. Additionally, newly generated blocks are writable before filled up. Once a new block is full, it checks overlap with hot_{T2} and we determine whether it should be prefetched. In this way, block [201,350] is put into block cache.

In addition, extent-based compactions and row-based compactions are also supported. First, extents are divided into blocks to do the same check as we mentioned. Rows can also check overlap with hot_{T1} but swapped into key-value cache (if used). Second, no matter extents, blocks or rows, they are all reorganized in the same way in the underlying LSM-tree. So we only need to deploy the prefetcher module in the corresponding place.

7. EXPERIMENTS

7.1 Experimental Setup

Testbed. The machine we use consists two 24-core Intel Xeon Platinum 8163 CPUs (96 hardware threads in total), a

512 GB Samsung DDR4-2666 DRAM and a RAID consisting of two Intel SSDs. For the model evaluation, we train our model using LightGBM [26]. For system performance, we implement Leaper in X-Engine with MySQL 5.7 which is deployed in a single node.

Baseline. To evaluate the impact of Leaper on the system performance, we compare with *Incremental Warmup* [1], which we consider as the state-of-the-art solution, in X-Engine [14]. Incremental Warmup exploits the idea of temporal locality and assumes the newly compacted blocks are frequently accessed if they overlap with any blocks in the block cache (i.e., they were accessed recently). In offline evaluation, we implement the idea of Incremental Warmup that assumes the records accessed in the latest time interval should be accessed in the next time interval.

Metrics. We evaluate both recall and precision of the proposed prediction model in Leaper. We need high recall to increase cache hit rates, and high precision to reduce the memory footprint of the prefetched records in the cache. We also adopt the Area Under Curve (AUC) metric [22] to evaluate the generalization ability of our model. Performance-wise, we evaluate the cache hit rate, QPS and latency of 95th percentile.

Workloads. We use synthetic workloads, generated by SysBench [18] with varying configurations (e.g., skewness), to evaluate the performance of Leaper in different scenarios. We further adopt two real-world workloads, e-commerce (i.e., placing orders for online purchases) from *Tmall* and instant messaging (i.e., online chats) from *DingTalk*, to evaluate how Leaper performs. These two workloads correspond to two typical applications in which cache invalidation problems are ubiquitous. Table 2 introduces the statistical details of these workloads.

Dataset. The dataset we use for offline evaluation is generated from the e-commerce workload for its typicality and universality (i.e., similar conclusions can be drawn by using the other two workloads). Specifically, we use the data in the first three days as the training set and data of the following one day as the testing set. Data dependencies and temporal relations are preserved in these data sets. The training and testing data sets contain 3,404,464 and 807,829 records, respectively.

7.2 Offline Evaluation

7.2.1 Overall results

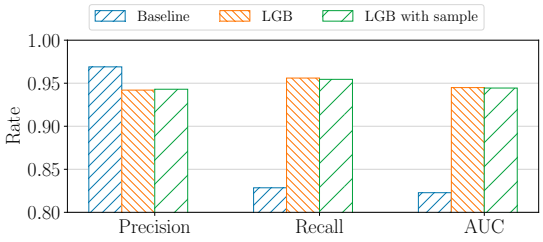


Figure 6: Results of metrics among baseline, LightGBM and LightGBM with sampling.

From Figure 6, LightGBM model used in Leaper (orange bars) performs much better than the baseline (blue bars). The recall score of 0.83 achieved by the baseline implies that 83% data has strong temporal localities and it could potentially perform well if the cache of the storage engine is large enough. And the higher recall score and AUC of LightGBM model indicate that it has a better predictive capability. Meanwhile, it is observed that the features we chose cannot always distinguish normal inputs from abnormal noises, causing the proposed model to mis-predict future accesses for those noisy inputs.

7.2.2 Influence of data sampling

Then we evaluate the impact of data sampling in collector. The green bars in Figure 6 show the precision, recall scores and AUC when we apply sampling. The sampling rate we use is 0.01. Despite sampling errors, our model with sampled inputs still achieves similar results as the accurate statistics. This is mainly because our model is a binary classifier, and the lazy initialization and double-checked locking in collector guarantee the 0/1 property. So such sampling errors have negligible impact on the results of the model.

7.2.3 Features

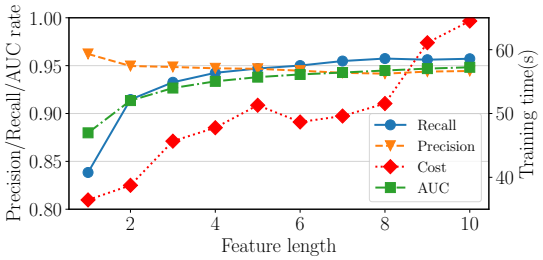


Figure 7: Variation of metrics and training time alongside feature length.

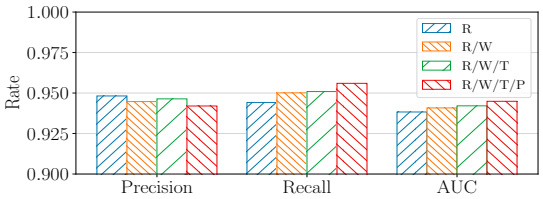


Figure 8: Results of metrics using different feature types.

Figure 7 shows the precision, recall and AUC metrics and training times for different feature lengths. With feature length increasing, both recall and AUC increase significantly until the length up to six, while the precision has a minor decrease and stabilizes at around 0.95. Meanwhile, the training cost rises generally with the length, which might be influenced by different early stopping rounds. We use the feature length of 6 based on the evaluation.

When it comes to feature types, Figure 8 gives an ablation study of feature types including R (read arrive rate), W (write arrival rate), T (prediction timestamp) and P (precursor arrival rate). Through the ablation study, read arrival rate, write arrival rate and precursor arrival rate contribute to the improvement of the recall score, while the prediction timestamp has minor impact on the recall. The importance of features generated by the LightGBM library shown in Table 3 also obtains the same conclusion. The AUC increases step by step, showing that all these features contribute to the robustness of the model.

7.2.4 Models

After determining the features we use in the model, we compare different models by evaluating their corresponding metrics and training time costs. We experiment with tens of models and select six best-performing ones to compare. Figure 9 presents the results. The blue, orange, green, red, brown and purple bars represent Logistic Regression (LR), Random Forest (RF), GBDT (implemented

Table 2: Detailed information of different workloads

Workload Type	Point lookups	Range lookups	Updates	Inserts	R/W ratio[1]	Table size[2]	Skewness[3]
Default synthetic workload	75%	0%	20%	5%	3:1	20m	0.5
E-commerce workload	75%	10%	10%	5%	6:1	10m	0.3
Instant messaging workload	40%	0%	35%	25%	2:3	8m	0.9

[1] Read-write Ratio is the approximate ratio.
[2] The workloads we use are all single-table, the table size also means the number of records.
[3] All three workloads approximately follow Power-law Distributions, we use zipf factor to demonstrate their skewness.

Table 3: Importance of features

Feature Type	Feature Length	Sum of Importance
Read	6	35.61%
Write	6	23.53%
Time	3	9.88%
Precursor	3	30.98%

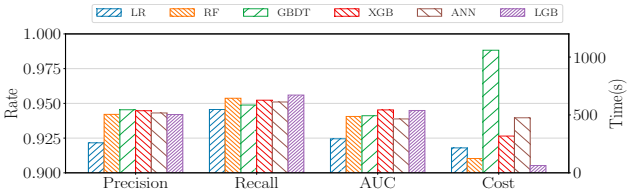


Figure 9: Results of metrics using different models.

by scikit-learn [30]), XGBoost (XGB), Artificial Neural Network (ANN, also called Multi-Layer Perceptron, with 2 hidden layers containing 120 and 3 cells), and LightGBM, respectively. LightGBM has the highest recall score and the second highest AUC score, with the least training time. Among other models, only XGBoost performs similarly with LightGBM. However, it consumes five times more training time. Therefore, we choose LightGBM in this work.

7.3 Online Performance

We implement Leaper in X-Engine [14] and compare it with the *Incremental Warmup* baseline. The initial data size is 10 GB with a key-value pair size of 200 bytes. The total buffer cache is set to 4 GB including 1 GB reserved for the write buffers (i.e., memtables), and 3 GB for the block cache. We run a 200-second test on each workload. During the test, about 10 background operations are triggered. The report interval for intermediate statistics is set to 1 second.

7.3.1 Cache invalidation in Flush

First of all, we run the test using the default synthetic workload to observe Leaper's effect on flush operations. We disable compactions to avoid the influence of compactions. The results are shown in Figure 10. The blue and orange lines represent the Incremental Warmup baseline and Leaper, respectively. Although shutting down compactions causes layers to accumulate in $Level_0$ without merging into the next level, and results in step descent of the QPS and step ascent of latency shown in the lower sub-figure of Figure 10, it does not interfere with our evaluation on flush because we only examine the performance of Leaper on cache invalidation problem. The results show that our method eliminates almost all the cache invalidations (reflected as cache misses) caused by flush and the QPS drops and latency spikes are also smoothed.

7.3.2 Cache invalidation in Compaction

With Leaper addressing the cache invalidations caused by flush, we now move on to evaluate its efficiency against the cache invalidation problem caused by compactions. We first use two real-world applications and then use synthetic workloads with varying configurations.

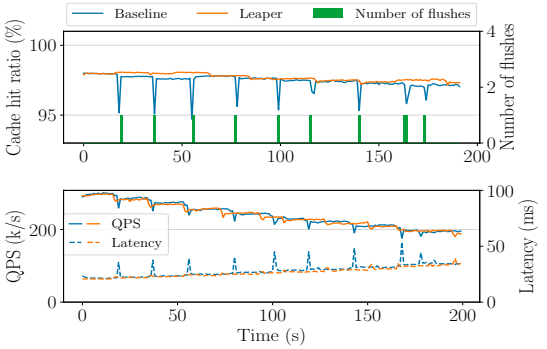


Figure 10: Cache hit ratio, QPS and latency of synthetic workload for flush operations over 200 seconds among baseline and Leaper.

Real-world Application

Figure 11(a) shows the cache hit rates, QPS and latency of the e-commerce workload over 200 seconds achieved by the Incremental Warmup baseline and Leaper. The green bars represent the number of compaction tasks running in the background.

The upper sub-figure indicates that our approach reduces about half of the cache invalidations. And, the cache hit rates recover about 2x faster after compactions with Leaper. The lower sub-figure indicates that our approach performs similarly with the baseline when there is no compaction, and outperforms the baseline during compactions.

Figure 11(b) shows the cache hit ratio, QPS and latency of the instant messaging workload over 200 seconds. The upper sub-figure also indicates that Leaper reduces about half of the cache invalidations. Although the instant messaging workload has more write operations and more compactions than the e-commerce workload, the lower sub-figure shows Leaper can still prevent significant QPS drops and latency raises caused by compactions, achieving a smooth overall performance.

We also evaluate the efficiency of *Key Range Selection* in the e-commerce workload. The initial value and the threshold are set to 10 and 0.6, respectively. The experimental results are shown in Figure 12. The key range with asterisk (i.e., 10^4) is the most suitable range size calculated. The results in Figure 12 shows 10^4 performs stabler cache hit ratio than other sizes and outperforms others in QPS.

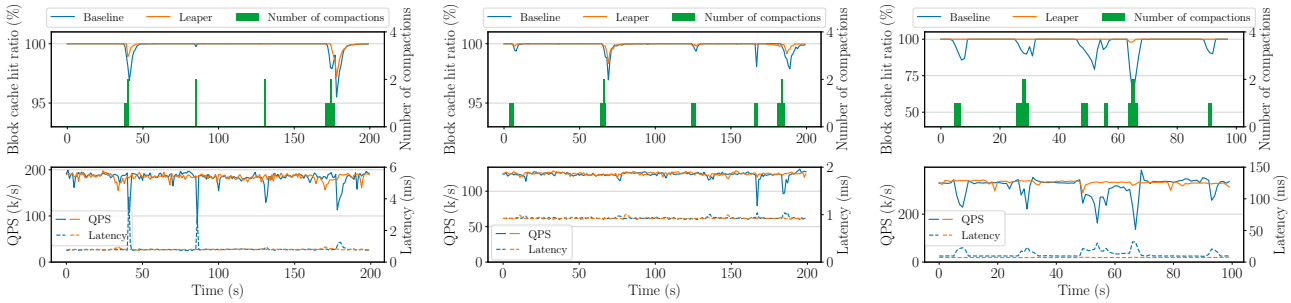
Synthetic workloads

Table 4: Achievements of Leaper

Workload Type	QPS	Latency	Cache misses
E-commerce	+83.71%	-46.35%	-40.56%
Instant messaging	+18.30%	-7.49%	-64.23%
Synthetic	+66.16%	-62.24%	-97.10%

[1] Statistics are collected during and after compactions.

Figure 11(c) shows the cache hit ratio, QPS and latency of the default synthetic workload over 200 seconds. From the upper sub-figure, Leaper still removes almost all cache invalidations while the baseline performs much worse than real-world workloads. This is mainly because compactions in synthetic workload contain more blocks than that in real-world applications. QPS and latency performance in the



(a) E-commerce (b) Instant messaging (c) Synthetic workload

Figure 11: Cache hit ratio, QPS and latency over 200 seconds among baseline and Leaper for E-commerce, Instant messaging and synthetic workload, respectively.

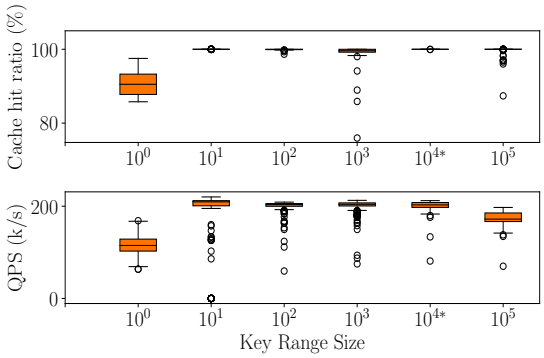


Figure 12: Cache hit ratio and QPS over different key range sizes ranging from 1 to 10^5. The key ranges with * means calculated effective ranges in the offline part.

lower sub-figure can also draw the same conclusion as above real-world workloads.

Table 4 summarizes the speedups of QPS and the smooths of latency achieved by Leaper over the baseline. Please note that the comparison are collected only during and after compactions (i.e., $T_1 + T_2$ in section 6), because Leaper is exploited to stabilize system performance and smooth latency rather than speed up overall performance. Combining the speedups and detailed information of these workloads, Leaper achieves the best speedup of QPS in the e-commerce workload, because of its high read-write ratio. Leaper has less QPS speedup and latency smooth but more cache misses eliminated in the instant messaging workload, because of its more skewed data distribution.

A. *Range-intensive workloads*

In this part, we study the influence of range-intensive workloads. The read-write ratio is fixed to 3:1 and we tune the range query ratio from 0 to 100% of the total read queries. From the lower sub-figure of Figure 13(a) we find that the QPS declines with the rise of range query ratio. And, Leaper always outperforms the baseline during and after compactions (i.e., $T_1 + T_2$ in section 6). At the ratio of 20%, the gap between baseline and Leaper is the biggest. In the upper sub-figure, the churn of the baseline's cache hit ratio worsens with increasing range query ratios, while Leaper performs much better for all ratios. We find that proper range queries (i.e., 20% of range query ratio) accelerate the collection of key range statistics for hot key ranges and then help to make accurate predictions.

B. *Data skew*

In Figure 14, we vary the skewness of keys accessed by both reads and writes according to a Zipf distribution, and measure the speedup of QPS achieved. When accesses obey uniform random distribution (i.e., the Zipf factor is 0), Leaper achieves no speedup, and even slows down because of the computational and storage overhead. With skewed accesses, Leaper can predict hot records and achieve high speedups. When the Zipf factor reaches up to about 1.0, the baseline can also work well because of the small number of hot records with such a high level of skewness. The skewness of most real-world workloads (e.g., the e-commerce workload and the instant messaging workload) are ranging from 0.3 to 0.9, so that Leaper can work well in them.

C. *Different mixtures*

Figure 13(b) shows the performance of baseline and Leaper while processing different mixtures of point lookups, updates and inserts. We start with the default mixture of 75% point lookups, 20% updates and 5% inserts, which represent common scenarios with many write operations. In this case, Leaper increases the cache hit rates from 97.66% to 98.59%. We gradually scale the shares of reads up to 85% and 90% while fixing the update/insert ratio (i.e., 4:1), Leaper always outperforms the baseline.

D. *Cache size*

We vary the cache size from 1GB to 8GB to explore its impact on the performance. Figure 13(c) indicates that Leaper outperforms the baseline no matter how we vary the cache size. Leaper obtains the maximum speedup of the cache hit ratio when the cache size is minimum (i.e., 1GB), because decreasing cache size affects more on baseline than Leaper. However, larger cache size can tolerate more incorrect evictions (i.e., incorrect prediction) caused by Leaper, so that the maximum speedup of QPS is obtained when the cache size is 2GB.

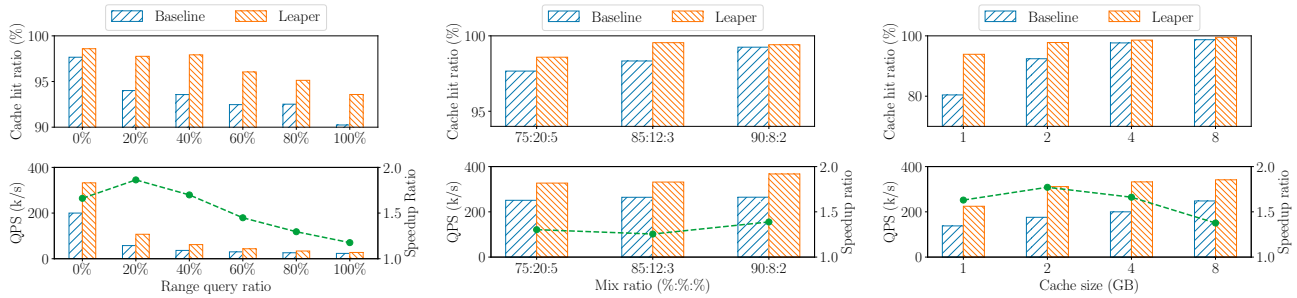
7.3.3 Computation & Storage Overhead

Table 5: Computation & Storage Overhead of Leaper

		Collector	LGB	LGB*	Check Overlap	Overall
Computation	Default synthetic	<1μs/query	3ms	1ms	1ms	-4.68%
	E-commerce	<1μs/query	21ms	5ms	3ms	-0.77%
	Instant messaging	<1μs/query	9ms	2ms	2ms	-0.95%
Storage	Default synthetic	3.2KB	4.9KB	8KB	-	-
	E-commerce	16.3KB	5.0KB	8KB	-	-
	Instant messaging	8.9KB	5.0KB	8KB	-	-

To better understand the computational and storage overhead of Leaper, we recorded the time and space it spends in its three main components and the overall overhead is expressed by QPS decline rate.

Collector: The time to record the key of a query and update the key range counter, and the maximum space the key range counter occupies in the memory.



(a) Range query (b) Mixture (c) Cache size

Figure 13: Cache hit ratio and QPS among baseline and Leaper over different range query ratio, mixture, and cache size, respectively.

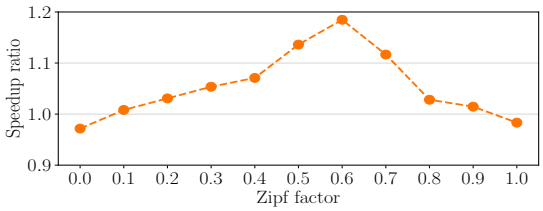


Figure 14: Speedup Ratio of QPS between Leaper and baseline with different zipf factors ranging from 0 to 1.0.

Inference: The time to do inferences in one compaction with CPU only, and the size of the model object containing both model parameters and tree structure for LightGBM.

Check Overlap: The time to run check overlap algorithm once.

Overall: Since some other components (such as input blocks of predict and prefetch, the prefetch operation) are shared with the storage engine, we could not compute the overhead of them singly. As a result, we adopt the decline of QPS exclude during and after flushes and compactions to capture the overall overhead of Leaper.

Table 5 shows that all of Leaper's components have acceptable computation and storage overhead. For the Collector, since we exploit sampling, the overhead of one query can be reduced to below 1 microsecond. For inference, we compare the inference time whether we use Treelite. Treelite achieves 3-5× speedup in the inference of LightGBM. For check overlap, the computation overhead is about 1-3 millisecond. The storage overhead is not computed because all the inputs are loaded into memory by the storage engine. The overall decline of QPS shows that the overhead of Leaper is kept below 5% and 0.95% for synthetic and real-world workload respectively.

7.4 Limitations of Leaper

There are two main limitations of Leaper. First, the models can be trained periodically but not incrementally. Therefore if the workload pattern changes dramatically or the newly generated key ranges have unknown patterns, the accuracy of prediction is affected. In this case, we exploit a simple rollback strategy, called Slow SQL Anomaly Detection (SSAD), to reduce the performance degradation. If the SSAD component detects the number of queries exceed predefined thresholds, it notifies the prefetcher to stop working until the model is updated. In the future, it is valuable to consider training and updating the model in a smart way. Second, the offline model training is decoupled from DBMS running. Although such design reduces the negative impact on online performance, it adds complexity for the deployment and the installation of DBMS. Especially for the traditional on-premise environment, the model training requires extra hardware resources that are not easily provided. We are exploring a lightweight and efficient machine learning pipeline coupled with DBMS in our future work.

8. RELATED WORKS

We have discussed the existing solutions for the cache invalidation problem in Section 2. Here we summarize other works using machine learning methods to solve problems in the database systems.

Machine learning methods assist the database administrators (DBAs) to manage the database. OtterTune [42] introduces a machine learning pipeline to recommend the optimal knobs configuration across different workloads. CDBTune [43] and Qtune [21] model the knobs tuning process as decision-making steps and use reinforcement learning algorithms to learn this process. QueryBot 5000 [23] proposed a forecasting framework that predicts the future arrival rate of database queries based on historical data. iBTune [39] uses a pairwise DNN model to predict the upper bounds of the response time which saves memory usage of the database. DBSeer [27] performs statistical performance modeling and prediction to help DBA understanding resource usage and performance.

Machine learning methods optimize modules of the database system. Learned index [20] uses the mixture of expert networks to learn the data distribution and use learned models to replace inherent data structures. ReJOIN [25] and Neo [24] use reinforcement learning methods to optimize the join order in the query execution plan. Logistic regression is used for transaction scheduling [38] and LSTM is used for memory access prediction [12].

Machine learning methods for the database architecture. Self-driving database systems [29] optimizes itself automatically using deep neural networks, modern hardware, and learned database architectures. SageDB [19] also proposes a vision where lots of components of the database systems can be optimized via learning the data distributions.

9. CONCLUSIONS

We introduce Leaper, a **Lea**rned **P**refetch**er**, to reduce cache invalidations caused by background operations (i.e., compaction, flush) in LSM-tree based storage engines. In Leaper, we introduce a machine learning method to predict hot records and prefetch them into caches accordingly. Evaluation results show that Leaper eliminates about 70% cache invalidations and 99% latency spikes with at most 0.95% overheads as measured in real-world workloads.

10. REFERENCES

[1] M. Y. Ahmad and B. Kemme. Compaction management in distributed key-value datastores. *PVLDB*, 8(8):850–861, 2015.

[2] Apache. Hbase. `http://hbase.apache.org/`.

[3] C. Berthet. Approximation of lru caches miss rate: Application to power-law popularities. *arXiv preprint arXiv:1705.10738*, 2017.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[5] C. J. Burges. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11(23-581):81, 2010.

[6] DMLC. Treelite. `http://treelite.io/`.

[7] Facebook. Rocksdb. `https://github.com/facebook/rocksdb`.

[8] T. Feng. Benchmarking apache samza: 1.2 million messages per second on a single node. *URL https://engineering. linkedin. com/performance/benchmarking-apache-samza-12-million-messagessecond-single-node*, 2015.

[9] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[10] Google. Leveldb. `https://github.com/google/leveldb`.

[11] L. Guo, D. Teng, R. Lee, F. Chen, S. Ma, and X. Zhang. Re-enabling high-speed caching for lsm-trees. *arXiv preprint arXiv:1606.02015*, 2016.

[12] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan. Learning memory access patterns. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80, pages 1919–1928. PMLR, 2018.

[13] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[14] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. *2019 International Conference on Management of Data (SIGMOD'19)*, 2019.

[15] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental organization for data recording and warehousing. In *VLDB'97*, pages 16–25. Morgan Kaufmann, 1997.

[16] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3146–3154, 2017.

[17] S. Kimak and J. Ellman. Performance testing and comparison of client side databases versus server side. *Northumbria University*, 2013.

[18] A. Kopytov. Sysbench: A system performance benchmark, 2004, 2004.

[19] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, J. Ding, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. Sagedb: A learned database system. 2019.

[20] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504, 2018.

[21] G. Li, X. Zhou, S. Li, and B. Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *PVLDB*, 12(12):2118–2130, 2019.

[22] J. M. Lobo, A. Jiménez-Valverde, and R. Real. Auc: a misleading measure of the performance of predictive distribution models. *Global ecology and Biogeography*, 17(2):145–151, 2008.

[23] L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 631–645. ACM, 2018.

[24] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer. *PVLDB*, 12(11):1705–1718, 2019.

[25] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–4, 2018.

[26] Microsoft. Lightgbm. `https://github.com/microsoft/LightGBM`.

[27] B. Mozafari, C. Curino, and S. Madden. Dbseer: Resource and performance prediction for building a next generation database cloud. In *CIDR*, 2013.

[28] E. J. O'neil, P. E. O'neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[29] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, et al. Self-driving database management systems. In *CIDR*, volume 4, page 1, 2017.

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[31] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6), 1990.

[32] M. Richardson, E. Dominowska, and R. Ragno. Predicting clicks: estimating the click-through rate for new ads. In *Proceedings of the 16th international conference on World Wide Web*, pages 521–530. ACM, 2007.

[33] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 134–142, 1990.

[34] S. Salza and M. Terranova. Workload modeling for relational database systems. In *Database Machines, Fourth International Workshop, Grand Bahama Island, March 1985*, pages 233–255. Springer, 1985.

[35] S. Sarkar, T. I. Papon, D. Staratzis, and M. Athanassoulis. Lethe: A tunable delete-aware LSM engine. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 893–908. ACM, 2020.

[36] D. C. Schmidt and T. Harrison. Double-checked locking. *Pattern languages of program design*, (3+):363–375, 1997.

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

[38] Y. Sheng, A. Tomasic, T. Zhang, and A. Pavlo. Scheduling oltp transactions via learned abort prediction. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '19, New York, NY, USA, 2019. Association for Computing Machinery.

[39] J. Tan, T. Zhang, F. Li, J. Chen, Q. Zheng, P. Zhang, H. Qiao, Y. Shi, W. Cao, and R. Zhang. ibtune: individualized buffer tuning for large-scale cloud databases. *PVLDB*, 12(10):1221–1234, 2019.

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

[41] H. D. Thoreau. Probability and random processes with applications to signal processing - united states edition. *Experimental Cell Research*, 139(1):63–70, 2002.

[42] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1009–1024, 2017.

[43] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, pages 415–432, 2019.