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]
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 a 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.

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., LRU [31], LFU [37]) 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.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

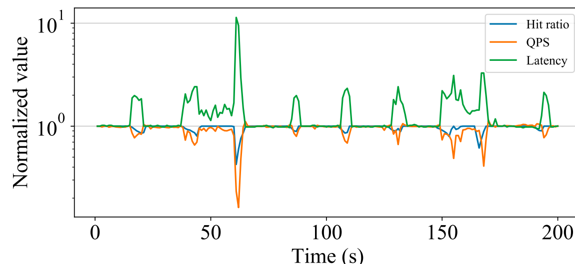


Figure 1: Cache hit ratio and system performance churn (including QPS and latency of 95th percentile) caused by cache invalidations.

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 a 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 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% 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.

The cache invalidation problem has attracted some research attention in the past [11, 1, 39]. 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, 42]. 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 conclude in Section 8.

2. BACKGROUND AND PRELIMINARY

2.1 LSM-tree based Storage Engines

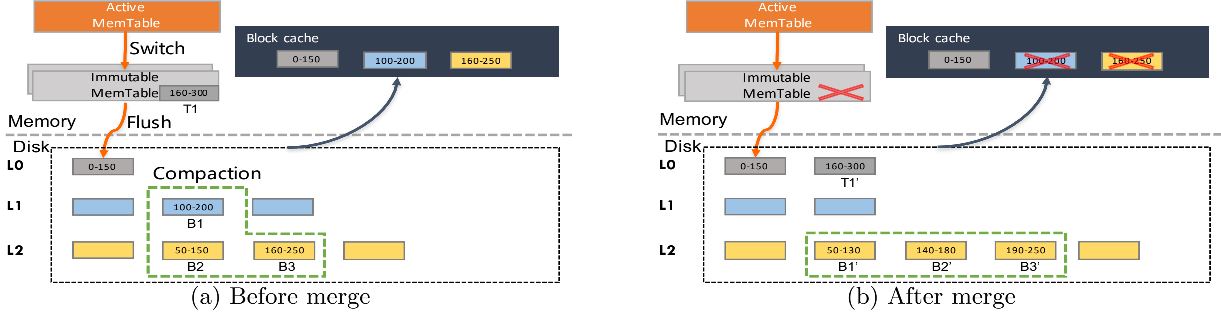
Both the idea of append-only writes using logs and the LSM-tree are not recent inventions [32], however, LSM-tree based storage engines have acquired significant popularity in recent years. Notable examples include LevelDB [10] from Google, RocksDB [7] from Facebook and X-Engine [14] from Alibaba, supporting 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(a) 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 [35, 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 with inclusive key ranges [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.

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



(a) Before merge (b) After merge

Figure 2: The left half describes the general architecture of LSM-tree based storage engines in which writes are directly appended into the active memtable and reads are retrieved in order of active memtable, immutable memtables and tiered trees on disk. The right half is an example of cache invalidations after flush and compaction.

hot records. Usually, a row cache for individual records and a block cache for record blocks are used [14]. In Figure 2(a), 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 a LSM-tree based storage engine, the cache invalidation problem happens when background compaction and flush operations move cached records either from the main memory to the disk or shuffle them within the disk, causing potential retrievals to miss the cache in the main memory and access the disk. In this work, our goal is to minimize such cache misses to improve the lookup performance. We formulate the problem as follows:

For a sequence of incoming requests for records $R = \{r_0, r_1, ...\}$, and a set of compactions (or flush) moving sets of records:

$$M = \{m^0 = \{r_0^0, r_1^0, ..., r_{n_0-1}^0\}, m^1 = \{r_0^1, r_1^1, ..., r_{n_i-1}^1\}, ...\},$$

and a database cache buffering a set of records:

$$C = \{r_0^c, r_1^c, ..., r_{L-1}^c\},$$

find a cache replacement method so that

$$|R \bigcap C| \text{ is maximized by minimizing } |R \bigcap C \bigcap M|,$$

where r_i, m^i, r_j^i, and r_i^c refer to a record, a compaction merging n_i records, a record merged by compaction, and records buffered in the cache with L records, $n_i, i, j, L \in N$.

Examples. Figure 2 illustrates an example of the formulated problem by showing the status of record blocks before and after a flush and a compaction. In Figure 2(a), the immutable memtable $T1$ can be accessed in the main memory. After the flush, all retrievals for records originally in $T1$ has to access the flushed blocks in $T1'$ in the disk. Before the compaction, one record blocks from level $L0$, $L1$, and $L2$ is cached, respectively. After a compaction merging the blocks selected by the green dashed line $B1$, $B2$ and $B3$, the corresponding cached blocks are invalidated, because the cached records have been merged with others and moved to new locations ($B1'$, $B2'$, $B3'$) in the level $L2$. In this case, retrievals of these records miss the cache and have to access these blocks in the disk.

Most relevant related work and existing solutions. Several methods were proposed to reduce cache invalidations. Stepped-Merge tree [15] is a 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 [42] combines the idea of SM-tree [15] and bLSM [39], 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 along blocks' key range. Before they are moved, the blocks in the block cache will be evicted whose key ranges overlap with them. However, it has two disadvantages. First, it assumes that newly compacted blocks will be frequently visited. Second, newly compacted blocks may overlap more than one block, it will prefetch infrequently requested blocks into the block cache.

3. DESIGN OVERVIEW

3.1 Design Rationale

Both traditional page-based cache replacement policies and existing methods that try to either delay compactions or maintain a mapping between record blocks before and after compactions are not sufficient to solve the cache invalidation problem formulated above. Firstly, record blocks of the LSM-tree are mutable and updated by flushes and compactions. This disables page-based cache replacement policies to track the access frequency of each block, unlike in page-based storage where the contents of a page rarely experience drastic changes since its formation. Secondly, in the emerging write-intensive workloads with massive inserts and updates that prefer the LSM-tree storage engine, the frequencies for flushes and compactions are inevitably high, driving up both the execution and memory overheads of the existing methods introduced above.

The cache invalidation problem can be divided into a time series classification problem and a traditional cache replacement problem. Firstly, because it is the set of records moved by compactions and flushes, M, that distinguishes the cache invalidation problem from the conventional cache replacement problem, we need to make a decision for each inval-

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

idated cache entry on whether to keep it in the cache, according to the prediction on whether or not it is going to be accessed either during the compaction or after the compaction. Or, formally, given the access of a block in the last $x \cdot t$ minutes, predict whether this block will be accessed in the following T minutes.

Secondly, for cached entries that are not invalidated by compactions and flushes, their access statistics can be maintained well by traditional policies like the LRU, making such policies sufficient to manage the cache for these entries. As a result, we propose a hybrid solution combining a classifier to guide cache swaps for invalidated entries and the LRU cache replacement policy for the other cache entries. The specific classification model is explained in Section 4.3.

Other solutions are also possible, such as sequence prediction and time series regression. For the sequence prediction approach, it collects a long historical record access sequence for a past period time, and makes predictions on the future access sequence. The computation and memory overhead incurred in this process is 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, it transforms the access pattern into arrival rates for each record to minimize the RMSE (Root Mean Squared Error). However, most workloads have obvious hot spots, which means that the transformation from regression to classification will lose considerable accuracy.

3.2 System Overview

To achieve the time series classifier introduced above, we introduce machine learning models to predict record accesses during and after compactions. Specifically, we predict at the 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 continuous primary 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 optimal performance.

The Collector collects data in a multi-thread way. We implement optimized lock mechanisms to reduce the overhead. After that, the Collector obtains the access statistics for different key ranges. The Collector finally transfers the access statistics of key ranges 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. Based on the predicted results, the Prefetcher finds out all blocks that will be accessed from those participating in the merge operations. This is 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 prefetching happens along with the processing of flush and compaction operations.

Figure 3 also depicts the control flow between the storage engine and leaper. During the database running, the Collector keeps updating the access statistics. The Learner trains the models periodically, depending on the pattern changes of the workload. It always 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 involve in the flush. For compaction, since it's usually a long time period, we use a two-phase mechanism to predict more accurate access of the key ranges that participate in the compaction.

4. OFFLINE ANALYSIS

To solve the cache invalidation problem, Leaper applies machine learning models to predicts the data blocks that would be accessed after the flush and compaction operations and then do the prefetch. In this section, we describe the design of Learner, the offline component of Leaper which handles data featurization and model training. We first discuss the *key range selection* to reduce the system overhead and make our approach realistic for the online components. After that, we discuss the important features of the prediction problem. Last, we describe the classification model and training methodology we use in Leaper.

4.1 Key Range Selection

To get training data for a classification model, we need to know the access information of blocks. But we are unable to get it since the blocks keep changing along with the merge operation. Instead, we introduce the key range, which also consists of a range of continuous keys and never changes during the database running. The key range of grouping keys together has three major advantages. First, it's infeasible to record the access information for each key, so it's of great

Algorithm 1: Key Range Selection

Input: Initial size A , the number of zeroes in v_i of counting map $\{(k_i, v_i)\}_{i=1}^n$ as $M(A)$ and decline threshold α
Output: Most suitable granularity A^*
1 **while** $2M(2A) > \alpha M(A)$ **do**
2 | $A \leftarrow 2A$;
3 Binary search to find the maximum value A^* satisfying $A^* M(A^*) > \alpha A M(A)$ from A to $2A$;
4 **return** A^*

help to reduce the overhead. Second, key range can be efficiently used to do further overlap checks in the Prefetcher. Third, key range is naturally fit for range queries.

The key range size has significant impacts 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.

In this paragraph, we discuss how to determine the size of key ranges. 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 size of vector (or matrix) shrinks to the half and it forms logical add (i.e., $1 + 0 = 1$, $1 + 1 = 1$ and $0 + 0 = 0$) between merging bits. In our example, the vector turns to be (1,1). We utilize the fact that the access information loss occurs and the proportion of zeros in vector (or matrix) declines as we expand the key range size. Thus, the information loss in our case can be represented by the decline in the proportion of zeros.

However, information loss does not necessarily lead to performance penalties for the prediction. Through observation and experimental analysis, if the proportion of zeroes compared to the previous beyond a threshold α which is a heuristic value, machine learning models can still obtain fine prediction results with a reasonable decrease in accuracy. And, we call this *efficient expansion*. We keep expanding the key range size until the expansion not efficient any more. In our case, the threshold α is set to 0.6. And eventually, the key range size is expanded to ten thousand. We show the selection results in the experimental section.

More 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 $A^* M(A^*) > \alpha A M(A)$. By using this algorithm, Leaper selects an appropriate range size to group keys together.

4.2 Features

Different from getting user features from the application and generating features from query strings, only the table ID and the primary key of each query are visible inside the storage engine. Considering this fact, we perform feature engineering on the log of storage engines to transform raw access information into useful features. This section explains the most three critical folds based on their importance to the tree-based classification models.

Read/write arrival rate. Different key ranges exhibit different access patterns, as shown in Figure 4(a). So read arrival rate is the most important feature for the model to capture access patterns. Figure 4(b) shows that write operation may share access pattern with read operation for some key ranges that fits in with users' behavior. For this reason, it's meaningful to set write arrival rate as a feature to help read arrival rate structure access pattern for some key ranges. The arrival rate is calculated within a time slot which is usually one or two minutes. In our scenario, we use 6 time slots (explained in Section 7) to represent the arrival rate. So there are 12 features for read and write in total.

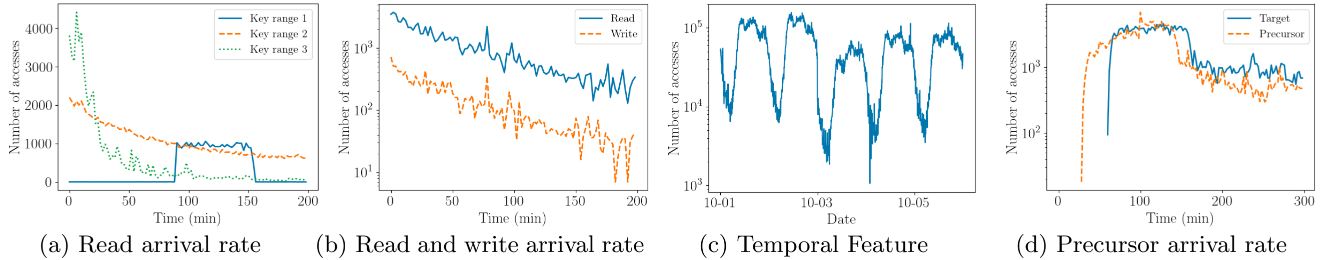
Prediction Timestamp. Figure 4(c) is the access load on one database table of E-commerce scenario in five days. Load peaks during daytime hours and dips at night. Since the load has obvious time-dependent characteristics [24], it is reasonable to add the prediction timestamp as a feature. We use 3 features, hour, minute and second of the day, to help the model to capture the access patterns of key ranges.

Precursor Arrival Rate. Figure 4(d) tells us some key ranges may share similar patterns with others. So we try to capture application patterns through studying the correlation between key ranges' access. If the target key range is often accessed following another key range's access pattern, we call that key range is the *precursor* of the target key range. We use *Cosine similarity* to verify the target key range shares a similar access pattern with the precursor. We pose Algorithm 2 to calculate γ most similar precursors of each key range. Through this algorithm, we add γ most similar precursors' arrival rate in the last one slotted time interval into features to help to capture the application patterns. In our case, γ is set to 3 which introduces 3 features for the model.

Algorithm 2: Calculation of Precursors

Input: Access sequence of all key ranges $\{(k_i, t_i)\}_{i=1}^n$
Output: Precursors map $\{(k_i, \{p_1, p_2 ... p_\gamma\})\}_{i=1}^n$ for key ranges
1 Initialize transfer matrix T of key ranges and define precursor constant γ and similarity threshold ϵ;
2 **for** *each item i in* $\{(k_i, t_i)\}_{i=1}^n$ **do**
3 | **for** *j from 1 to* γ **do**
4 | | $T[k_i][k_{i-j}] \leftarrow T[k_i][k_{i-j}] + 1$;
5 Calculate access vectors $\vec{V_i}$(a vector of N bits for N time intervals) of key ranges;
6 **for** *each key range* k_i **do**
7 | *count* $\leftarrow 0$;
8 | **for** *item* k_j *in sorted*$(T[k_i])$ **do**
9 | | **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**
10 | | | Add k_j into k_i's precursors;
11 | | | *count* $\leftarrow$ *count* $+ 1$;
12 | | **if** *count* $= \gamma$ **then**
13 | | | **break**;

Other things need to be determined are the slotted size



(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.

and the feature length. The slotted size needs to be atomic but as long as possible in order to reduce overhead. Usually, the slotted size is set to one or two minutes, but it would also be limited by other constraints that will be discussed in Section 6. The feature length is reached from the experiment to be six, and details are shown in Section 7.

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 [36] and learning to rate [4]. For every feature, GBDT needs to scan all the data instances to estimate information gain of all the possible split points. Thus, we could learn the computational complexity of GBDT is proportional to both the number of features and the number of data instances. When it comes to big data or strict time requirements, GBDT may not be a suitable choice. Fortunately, there are several works speed up GBDT like training via GPU [44, 29], parallel training [27] and novel implementations of GBDT including XGBoost [5] and LightGBM [16]. According to the experiments shown in [16], LightGBM can outperform XGBoost and other implementations in terms of computational speed and memory consumption. Our experiments in Section 7 also confirm that, and thereby, we select the implementation of LightGBM.

Other kinds of machine learning models including neural networks are also been tried. However, they are been proved not feasible because of accuracy nor inference overhead. Take LSTM used in [13] as an example. The time to do hundreds of inferences in one compaction is about 1-2 seconds, which is unbearable in the 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 temporal features and 3 precursors' 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 to search for the optimal parameters on testing set. The main parameters we tune are *num_leaves, learning_rate, bagging_fraction* and *feature_fraction*. All the above parameters help avoid overfitting. K-fold cross validation also helps determine the ultimate parameters. Finally, *num_leaves*, *learning_rate*, *bagging_fraction*, and *feature_fraction* are 31, 0.05, 0.8 and 0.9, respectively.

What is more, we only train one global model for different key ranges. Because new key ranges would be generated during the database running, one global model for all key ranges has better generalization ability. Also, it can help reduce inference overhead.

5. ONLINE PROCESSING

In this section, we mainly present how to collect statistics, how to implement prediction and how to decide which blocks need to be prefetched in the online components of Leaper.

5.1 Locking Mechanism

In multi-thread storage engines, collecting statistics into Collector will lead to write conflicts. To prevent this, it's necessary to apply a locking mechanism in Collector. There exists a trade-off between the system overhead and the recording accuracy and our first principle here is not affecting the throughput of the database systems. We try to sacrifice some collecting accuracy which means we don't record all access data. We find if the recording error is within a reasonable range, it has little influence on the accuracy of prediction. Experiment results in Section 7 will also prove this.

Table 1: Influence of locking mechanisms

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

Specifically, we adopt three strategies in the design of Collector. First, double-checked locking [38] and lazy initialization are used for initializing key ranges. Lazy initialization avoids initializing a key range until the first time it is accessed and double-checked locking is typically used to reduce locking overhead when implementing lazy initialization. In fact, double-checked locking reduces the overhead of acquiring a lock by testing the locking criterion before acquiring the lock. Second, atomic operations are used in statistics to replace the mutex. Global mutex itself has a tremendous impact on system performance, as shown in Table 1. If no statistics are recorded, the average QPS is about 337703 per second. When global mutex adopted, there will arise 40.61% QPS decline. But if atomic operations adopted, the decline ratio can be reduced to 17.21%. Third, sampling strategy helps further reduce the total overhead. Because the initialization and the first record of a key range are guaranteed

by double-checked locking, the following records of the same key range are sampled at the probability of P. Therefore, the estimate of key range $\hat{N}_i$ can be calculated by the statistical value S_i and sampling probability P:

$$\hat{N}_i = \frac{S_i - 1}{P} + 1 \quad (1)$$

Then, we compute the sampling error of our locking mechanism as follows:

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

Although our approach in Collector can't guarantee the accuracy of collecting the key range statistics, it has superior performance than other strategies and controls the QPS decline to only 3.66%.

5.2 Inference

After the flush or the compaction is triggered, the *Prefetcher* begins to work. The inference module predicts the hot and cold key ranges using the featurized data from the *Collector* and the trained model from the *Learner*. Through inference, we divide all involved key ranges into hot key ranges and cold key ranges. 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.

Although grouping keys into key ranges reduce the number of inferences, it is still challenging to minimize the cost of each inference. In Leaper, we use the Treelite [6] to implement the model inference part. 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. In other words, without recompiling the inference code, we just 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 do comparison experiments and are flexible for Leaper to support more models from more 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 to figure out what target blocks from the compaction and flush operations should be prefetched. Since there is no one-to-one correspondence between the key ranges generated by *Key Range Selection* and the target blocks, we need to check whether the target blocks contain hot key ranges.

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
   /* Case 1:  O(nlogm) < O(m)                          */
1  start = A_1, end = A_m ;
2  for a_j in hot key ranges do
3      Binary Search for A_i ⩽ a_j < A_{i+1} from start to
         end;
4      while b_j > A_{i+1} do
5          if Min(B_i, b_j) ⩾ a_j then
6              T.Add((A_i, B_i));
7          i ← i + 1;
8      start = A_{i+1};
   /* Case 2:  O(m) ⩽ O(nlogm)                          */
9  for A_i in target blocks and a_j in hot key ranges do
10     if Min(B_i, b_j) ⩾ Max(A_i, a_j) then
11         T.Add((A_i, B_i));
12     if B_i < b_j then
13         i ← i + 1;
14     else if B_i > b_j then
15         j ← j + 1;
16     else
17         i ← i + 1, j ← j + 1;
```

We pose a *Check Overlap Algorithm* ti check whether target blocks would be prefetched as Algorithm 3. There are two ways to do overlap check between predicted hot key ranges and target blocks. **Case 1** follows the thought of binary search and **Case 2** follows the thought of sort-merge. Which implementation we apply is determined by when the Prefetcher gets specific information of target blocks. If we get it at the end of flush or compaction, **Case 2** will be invoked. Otherwise, if we get it during flush or compaction, **Case 1** will be invoked. If we get the specific information 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. As a result, we adopt a hybrid algorithm combining both of them to obtain the optimal result.

What's more, other technologies like *Date reuse* [14] and *Bloom Filter* [3] are exploited to reduce the errors of *Check Overlap Algorithm*.

6. OPTIMIZATIONS

Leaper designs prefetch methods for both flush and compaction. For flush, all records in immutable memtables would not be modified, so blocks that need to be prefetched can be computed directly by *Check Overlap Algorithm* with the inputs of flushed records and hot key ranges predicted by trained models. Leaper only puts these blocks into block cache and then solves cache invalidation caused by flush.

For compaction, firstly, blocks for prediction and prefetch are not the same. Therefore, the inputs of *Check Overlap Algorithm* should be newly compacted blocks. Second, compaction involves much more records than flush. To reduce

the influence of Prefetcher on block cache, Prefetcher kicks off old blocks to save memory. Third, compaction brings version switches. If those kicked off blocks accessed before the end of compaction (the records requested are old-version), it will lead to a different type of cache miss.

Leaper introduces another kind of prefetch method called *Two-phase Prefetcher* to respond to the above problems. *Two-phase Prefetcher* exploits a combination of innovative methods both in prediction and storage engine. In prediction, *Multi-step Prediction* helps Leaper predict the future access of data in a more fine-grained way. In storage engine, *Two-phase Prefetcher* is bound to merge tasks in compaction, so that the inputs of *Check Overlap Algorithm* can be compacted blocks and hot key ranges. Meanwhile, *Two-phase Prefetcher* predicts the future access of the key ranges that participate in the compaction in two phases. In the first phase (eviction phase), it predicts the access during the entire process of the compaction operation. In the second phase (prefetch phase), it predicts the access in an estimated period of time after the compaction operation is done.

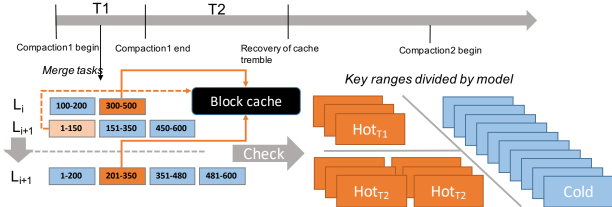


Figure 5: The design of Prefetcher for compaction.

Two-phase Prefetcher is designed like Figure 5. At the beginning of one compaction, we call multi-step prediction models trained offline to distinguish hot and cold key ranges. Then, we compute T_1 and T_2 based on the number of blocks participate in the compaction and previous log data. At last, target blocks in two phases are operated (i.e., evicted or prefetched), respectively.

6.1 Multi-step Prediction

Firstly, we need multi-step prediction models to fit *Two-phase Prefetcher*. One step corresponds to the slotted size. It has been mentioned in Section 4 that slotted size needs to be atomic but as long as possible to reduce overhead. Slotted size is limited by T_1 and T_2 in Figure 5 where T_1 means the duration of one compaction, and T_2 means recovery time of hit ratio decline caused by cache invalidations.

In theory, slotted size is the greatest common divisor of T_1 and T_2. But in fact, every compaction has its own T_1 and T_2 and recovery time T_2 is far less than compaction time T_1. As a result, the slotted size can be expressed as T_2. Since we determine slotted size t, the number of steps k can be calculated by the constraint as follows:

$$k \cdot t \geq Max(T_1 + T_2) \tag{2}$$

In fact, *Multi-step Prediction* contains k models to predict whether key ranges would be accessed in the next k slotted time intervals. 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* can't be used directly because different compaction has different T_1 and T_2. We first need to estimate the value of T_1 and T_2 and then combine k slotted time intervals into T_1 and T_2 for formation of two-phase binary classification. At last, we use the two-phase binary classification to perform prefetching.

According to our analysis, T_1 and T_2 approximately satisfy the following equations respectively:

$$\begin{cases} T_1 = \alpha N \\ T_2 = \beta \dfrac{Q}{S} \end{cases} \tag{3}$$

Where N means the number of blocks needed to merge, Q means QPS (short for queries per second) and S means the size of block cache. Both α and β are constants related to workloads. They can be computed approximately by sampling other variables.

After determining the value of T_1 and T_2, hot key ranges for T_1 and T_2 can be computed by Equation (4):

$$\begin{cases} hot_{T1} = hot_{1t} \cup hot_{2t} \cup \cdots \cup hot_{k_1 t}, & k_1 t \leq T_1 \leq (k_1 + 1)t \\ hot_{T2} = hot_{(k_1+1)t} \cup \cdots \cup hot_{k_2 t}, & k_2 t \leq T_1 + T_2 \leq (k_2 + 1)t \end{cases} \tag{4}$$

Where hot_{T1} means hot key ranges for T_1, hot_{T2} means hot key ranges for T_2 and hot_{it} means hot key ranges for the ith slotted time interval. Since one compaction is made up of many merge tasks, we use one merge task as shown in the left part of Figure 5 to describe the process of prefetching for example. In this case, block [300,500] from $Level_i$ and block [1,150] from $Level_{i+1}$ are added into cache before merging. They check overlap with hot_{T1} and we know block [1,150] won't be accessed before the end of compaction, so we evict it from block cache while block [300,500] remains kept in block cache. Also, the other three blocks and blocks reused will be checked as well to make use we don't let any block will be accessed go. 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 don't need extra storage overhead to get exact blocks. Additionally, newly generated blocks are of writable state until filled up. Once a new block is full, it checks overlap with hot_{T2} and we determine whether it will be requested in T_2. In this way, block [201,350] is put into block cache.

For another thing, merge tasks are not only block-based but also extent-based or key-based. In the first phase, extents will be divided into blocks to do the same check as we mentioned. For keys, it can also check overlap with hot_{T1} but added into KV cache (if used). In the second phase, no matter extents, blocks or keys, they are all reorganized in the same way. We only need to arrange the Prefetcher module in the right place.

7. EXPERIMENTS

7.1 Experimental setup

Testbed. The machine we use consists of two 24-core Intel Xeon Platinum 8163 CPUs (96 hardware threads in total), a 512 GB Samsung DDR4-2666 DRAM, a RAID consisting of two Intel SSDs and a Nvidia Tesla P100 GPU. For model evaluation, we train our model with LightGBM [28]. For system performance, we implement Leaper in X-Engine with MySQL 5.7 which deployed in a single node.

Baseline. To evaluate the prediction accuracy for hot records, we compare Leaper with *a temporal locality* baseline that assumes records accessed in the previous second are predicted to be accessed again in the next second. To evaluate

the impact of Leaper on the system performance, we compare it with the state-of-the-art, *Incremental Warmup* [1]. Both Leaper and Incremental Warmup are implemented in X-Engine.

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 adopt the Area Under Curve (AUC) metric [23] to evaluate the generalization ability of our model. Performance-wise, we evaluate the cache hit rate, query per second (QPS) and latency of 95% confidence level.

Workloads. We use synthetic workloads, generated using 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) and instant messaging (i.e., online chats), to evaluate how Leaper performs serving these popular applications of LSM-tree storage engines. Table 2 introduces the details of these workloads.

Dataset. The dataset we use for model training is generated from the E-commerce workload. 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*

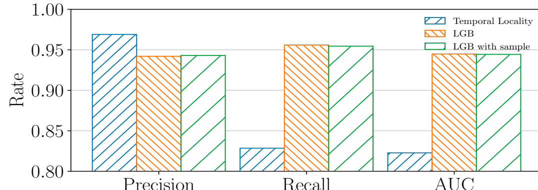


Figure 6: Results of prediction and baseline and influence of data sampling.

Figure 6 shows the precision, recall scores and the AUC of the temporal locality baseline (blue bars) and the LightGBM model used in Leaper (orange bars).

To conclude, our model performs much better than the baseline. With similar precision, our model achieves 15.39% higher recall scores than the baseline. The recall achieved by the baseline implies that 82.85% data has strong temporal localities and a cache large enough to store all of them could potentially perform well. The 15.39% increase of recall achieved by our model translates to higher cache hit rates given the same cache capacity. The 14.84% increase of AUC achieved by our model over the baseline shows that our model has a better predictive capability which gives more robust results.

7.2.2 *Influence of data sampling*

We now evaluate the impacts of data sampling in the Collector. The green bars in Figure 6 show the precision, recall scores and AUC achieved when we adopt sampling on the key range collection. The sampling rate we use is 0.01, as we use in the Collector. As introduced in Section 4, we draw into sampling errors trading for less overhead of the Collector. Despite such errors, our model with sampled inputs still achieves similar results as the accurate statistics. This is mainly because our model is a binary classifier, so that such sampling errors have negligible impacts on the binary outputs of the model, as confirmed by the above results.

7.2.3 *Features*

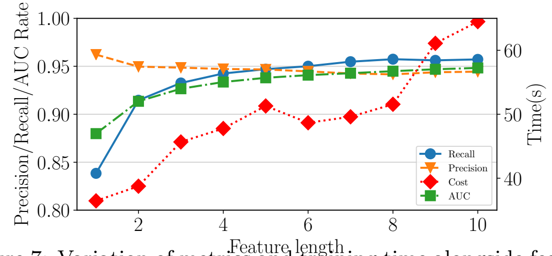


Figure 7: Variation of metrics and training time alongside feature length.

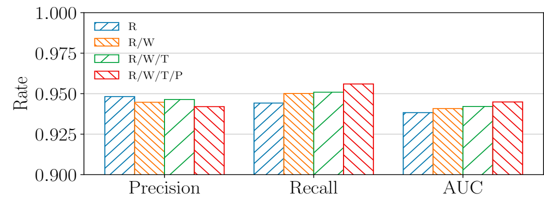


Figure 8: Variation of metrics using different feature types.

As discussed in Section 4.3, feature selection includes two aspects, length and types. Figure 7 shows the precision, recall and AUC metrics on the left y-axis, and the training time on the right y-axis, for different feature lengths. With increasing feature lengths, both recall and AUC increase significantly until the length larger than six. The precision has a minor decrease and stabilizes at around 0.95. On the other hand, the cost rises proportionally with increasing length, which may be influenced by different early stopping rounds. These results show that the increasing feature lengths pay off up until around a length of four to six.

When it comes to feature types, Figure 8 gives out the precision, recall scores and AUC of different feature types. We use the same prediction results provided by the LightGBM model in this experiment, and only changes the feature type: R (read arrive rate), R/W (write arrival rate), R/W/T (prediction time) and R/W/T/P (precursor arrival rate). Comparing these experiments, read arrival rate, write arrival rate and precursor arrival rate contribute to the improvement of the recall score, while the prediction time has relatively minor impacts. The importance of features generated by the LightGBM model shown in Table 3 also achieves the same conclusion. The AUC increases when adding more features, 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 including Linear models, Naive Bayesian models, Decision Tree models and Neural Network models, and select six best-performing ones to compare. Figure 9 presents

Table 2: Detailed information of different workloads

Workload Type	Point lookups	Range lookups	Updates	Inserts	Read-write Ratio[1]	Table size[2]
Default synthetic workload[3]	75%	0%	20%	5%	3:1	20m
E-commerce workload	75%	10%	10%	5%	6:1	10m
Instant messaging workload	40%	0%	35%	25%	2:3	8m

[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 primary keys.
[3] The synthetic workload uses the default mixture. The default Zipf factor we use is 0.5.

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%

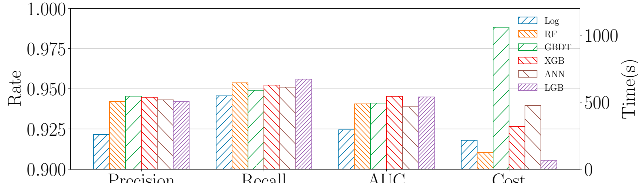


Figure 9: Variation of metrics using different models.

the results. The blue, orange, green, red, brown and purple bars represent Logistic Regression (LR), Random Forest (RF), GBDT (implemented by scikit-learn [34]), 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. Thus, 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. The report interval for intermediate statistics is set to 1 second.

7.3.1 Cache invalidation in Flush

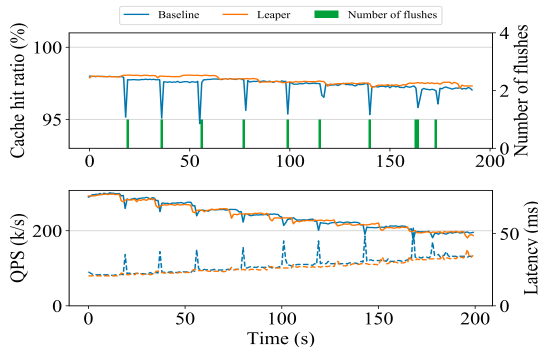


Figure 10: Cache hit ratio, QPS and latency of synthetic workload for flush operations over 200 seconds among baseline and Leaper.

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 steep descent of the QPS and step ascent of latency shown in the lower figure of Figure 10, it does not interfere with our evaluation on flush. The upper figure indicates that our method removes almost all the cache invalidations (reflected as cache misses) caused by flush. And, the lower figure indicates that the QPS increases and latency decreases along with the reduced cache misses.

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 same problem caused by compactions. We first use two real-world applications and then use synthetic workloads with varying configurations.

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

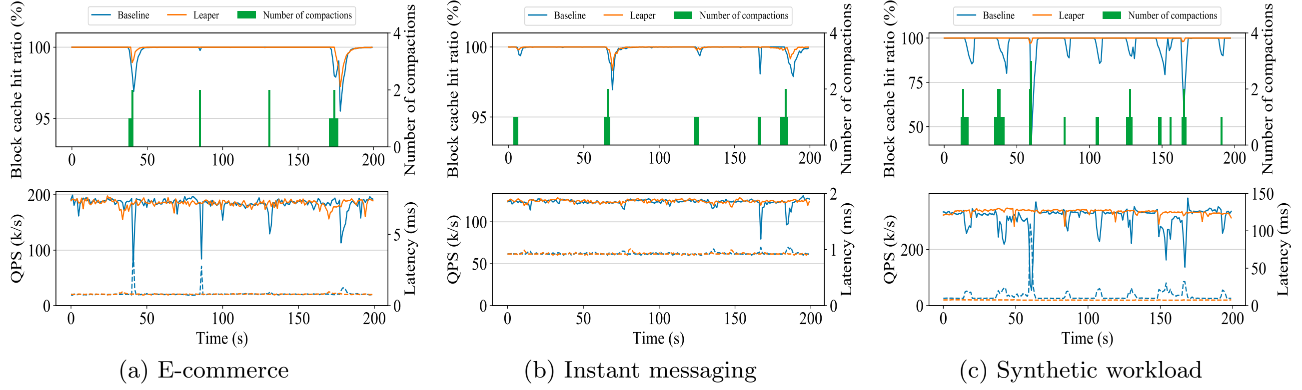
Figure 11(b) shows the cache hit ratio, QPS and latency of the instant messaging workload over 200 seconds. The upper figure also indicates that Leaper reduces about half of the cache invalidations. Despite that the instant messaging workload has more write operations and more compactions than the e-commerce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

We now evaluate the efficiency of *Key Range Selection* in the e-commerce workload. The initial value we choose is 10 and the merge threshold we choose is 0.6. The experimental results are shown in Figure 12. The key range with * (i.e., 10^4) is the most suitable range size calculated. From the upper figure, we find it performs stabler cache hit ratio than other sizes. The lower figure shows that 10^4 is nearly optimal for the system performance.

Flexible Benchmark

A. *Overall results*

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



(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.

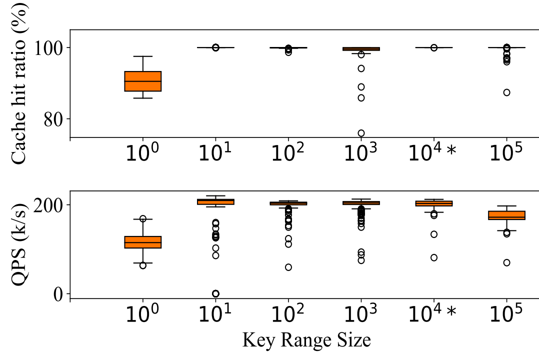


Figure 12: Cache hit ratio, QPS and latency over different key range sizes ranging from 1 to 10^5. The key ranges with * means calculated effective ranges in the offline part.

synthetic workload contain more blocks than that in real-world applications. QPS and latency performance in the lower 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. It is important to 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. It is shown that during and after compactions, the QPS speedup, latency smooth, and cache miss elimination can achieve the increase of about 50%, 40% and 70% on average, respectively, which indicates Leaper's effectiveness for cache invalidation problem in LSM-tree based storage engines.

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.

B. *Range query*

In this part, we study the influence of the range query. Its 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 figure of Figure 13 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 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 the range query accelerates the collection of key range statistics for hot key ranges and then helps to make more accurate predictions. Overall, Leaper outperforms the baseline for range-intensive workloads.

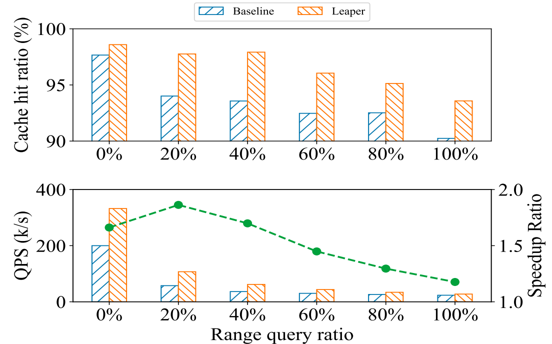


Figure 13: Cache hit ratio and QPS over different range query ratios ranging from 0% to 100% among baseline and Leaper.

C. *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 are uniformly distributed at random, Leaper achieves no speedup, and even slows down because of the computational and storage overhead. With skewed accesses, some records become hot, allowing Leaper to achieve higher 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.

D. *Different mixtures*

Figure 15 shows the performance of Leaper and baseline 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%

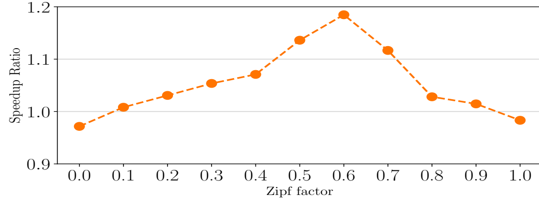


Figure 14: Speedup Ratio of QPS between Leaper and baseline with different zipf factors ranging from 0 to 1.0.

and 90% while keeping the same percents of updates and inserts, Leaper always outperforms the baseline.

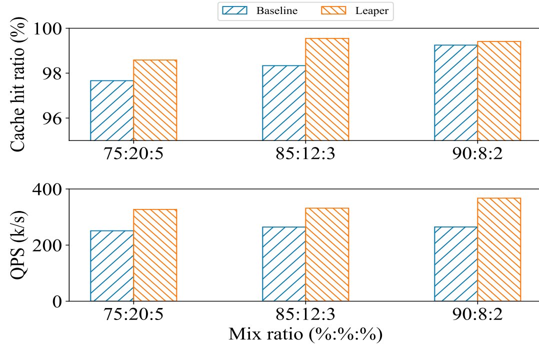


Figure 15: Cache hit ratio and QPS over different mixtures of point lookups, updates and inserts.

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 overall overhead expressed by QPS.

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

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 reasonable 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, which is reasonable for system performance.

8. RELATED WORKS

We have discussed the existing solutions that try to solve the cache invalidation issue in Section 2. Here we summarize other works using Machine Learning methods to solve problems in the database systems.

Prior works have studied the use of machine learning to assist the database administrators to manage the database. OtterTune [43] introduces a machine learning pipeline to recommend the optimal knobs configuration across different workloads. CDBTune [45] and Qtune [21] model the knobs tuning process as decision-making steps and use reinforcement learning algorithms to learn this process. QueryBot 5000 [24] proposed a forecasting framework that predicts the future arrival rate of database queries based on historical data. iBTune [41] uses a pairwise DNN model to predict the upper bounds of the response time which saves memory usage of the database. DBSeer [30] performs statistical performance modeling and prediction to help DBA understanding resource usage and performance.

The other category of works integrate machine learning techniques into the database kernel to optimize different individual modules of the 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 [26], Neo [25] have been proposed to optimize the join order of the planner using reinforcement learning methods. Some approaches are proposed to intelligently schedule transactions in DBMS kernel [46, 40]. Other work like [12] uses LSTM to predict the memory access pattern, for the purpose of prefetching future memory address accesses.

Self-driving relational database systems [33] 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 pipeline to predict records that would be requested in near future from moved records. And then prefetch them into caches accordingly. We have optimized the implementation of Leaper to keep its overhead below 0.95% in real-world workloads. Our evaluations using both synthetic and real-world workloads show that Leaper eliminates about 70% cache invalidations and 99% latency spikes compared with the state-of-the-art baseline.

Future work. In this work, DBMS is decoupled from offline model training. Although such design reduces the negative impact for online performance, it introduces complexity for DBMS deployment. Especially for traditional on-premise environment, the offline training requires extra hardware resources that are not easily provided. We will explore more light-wight and efficient machine learning pipeline coupled with DBMS in the future.

10. REFERENCES

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

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

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

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

[5] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[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.

[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. Jagadishl, P. Narayan2t4, S. Seshadri, R. Kanneganti, and S. Sudarshan3t5. Incremental organization for data recording and warehousing. In *VLDB*, volume 97, pages 16–25. Citeseer, 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. *Proceedings of the VLDB Endowment*, 12(12):2118–2130, 2019.

[22] H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte. Cardinality estimation using neural networks. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, pages 53–59. IBM Corp., 2015.

[23] 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.

[24] 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.

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

[26] 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.

[27] Q. Meng, G. Ke, T. Wang, W. Chen, Q. Ye, Z.-M. Ma, and T.-Y. Liu. A communication-efficient parallel algorithm for decision tree. In *Advances in Neural Information Processing Systems*, pages 1279–1287, 2016.

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

[29] R. Mitchell and E. Frank. Accelerating the xgboost algorithm using gpu computing. *PeerJ Computer Science*, 3:e127, 2017.

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

[31] 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.

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

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] 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. *Proceedings of the VLDB Endowment*, 12(10):1221–1234, 2019.

[42] 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.

[43] 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.

[44] H. Zhang, S. Si, and C.-J. Hsieh. Gpu-acceleration for large-scale tree boosting. *arXiv preprint arXiv:1706.08359*, 2017.

[45] 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.

[46] T. Zhang, A. Tomasic, Y. Sheng, and A. Pavlo. Performance of oltp via intelligent scheduling. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1288–1291, April 2018.