$LBSC^{+}$: Learning-based Cost-aware Caching with Performance Optimization for Cloud Databases

Zhongle Xie, Kehan Liu, Zhaoxuan Ji *Member, IEEE,* Yuncheng Wu, Meihui Zhang *Senior Member, IEEE,*

***Abstract*—Caching is an essential technique for mitigating the high latency and low bandwidth of cloud databases. However, existing caching algorithms are often unsuitable for such environments as 1) they provide limited adaptability to changing workloads, and 2) most of them do not explicitly incorporate data fetching costs into their decision-making process. Integrating learning-based models with cost-aware caching algorithms is natural for better performance. However, most existing learning-based methods focus solely on eviction, assuming that all new items should be admitted into the cache by default. This oversight leads to inefficient cache utilization, as low-frequency or even one-time access data items are unnecessarily stored and quickly evicted. Extending learning to jointly optimize admission and eviction is further complicated by the absence of the oracle algorithm to guide the model. Moreover, current learning models incur significant computation overheads, potentially worsening the performance of cloud databases.**

In this paper, we propose a learning-based cost-aware caching framework called $LBSC^{+}$ for cloud databases, ensuring faster query execution and robust performance in dynamic workloads. We first introduce an approximately optimal oracle algorithm called BeladySizeCost for joint admission and eviction, which retains data items with high cost per byte that are likely to be accessed in near future. Then, we present a lightweight supervised learning-based model that learns from BeladySizeCost to jointly predict the admission probability of incoming requests and the eviction probability of cached data. Moreover, we design effective optimizations to significantly reduce the computation overheads of the learning-based algorithm. Extensive experiments in both simulations and real-world cloud databases demonstrate that the proposed framework significantly outperforms the state-of-the-art baselines.

***Index Terms*—cloud database, caching, machine learning**

I. Introduction

CLOUD databases [1]–[4] typically adopt a storage-disaggregation architecture that divides computation and storage into separate server nodes connected through the network, which has attracted growing attention from both academia and industry. To reduce query latency, these systems usually cache data at the computation nodes. However, traditional caching frameworks are not well-suited to such systems as they fail to satisfy the following three requirements:

Zhongle Xie is with the Zhejiang University, Hangzhou 310000, China (e-mail: xiezl@zju.edu.cn)

Kehan Liu is with the Zhejiang University, Ningbo 315000, China (e-mail: 22551107@zju.edu.cn)

Zhaoxuan Ji is with the Beijing Institute of Technology, Beijing 100081, China (e-mail: jizhaoxuan@bit.edu.cn)

Yuncheng Wu is with the Renmin University of China, Beijing 100872, China (e-mail: wuyuncheng@ruc.edu.cn

Meihui Zhang is with the Beijing Institute of Technology, Beijing 100081, China (e-mail: meihui_zhang@bit.edu.cn)

Corresponding author: Zhaoxuan Ji.

(1) Adaptability to dynamic workloads. Cloud databases experience highly dynamic workloads compared to conventional databases [5]. Typically, real-world workloads include various access patterns, such as random access and scan [6]. Traditional heuristic-based caching algorithms like LRU (Least Recently Used) [7] may suffer performance degradation under these workloads as they can only work well for certain access patterns. Consequently, systems relying on these heuristic-based algorithms cannot adapt to changing access patterns of the workload.

(2) Awareness of data fetching cost. The data fetching cost in cloud databases comprises both the transfer and computation costs, which require special consideration. First, since network traffic is the primary bottleneck in cloud databases [8], [9], the time cost of transferring data from the storage node to the local cache cannot be ignored. Second, cloud storage systems typically offer predicate push-down as a native capability, such as AWS S3 Select [10], allowing users to send predicate operations to remote storage and thus yields computation costs at the storage nodes. Besides, a burst of slow queries may last for several minutes in commercial cloud databases [11], making performance optimization in cloud environments even more critical. Without explicitly considering these two types of costs, the caching frameworks can provide limited performance in the cloud environment.

(3) Coordinated admission and eviction. Existing caching algorithms typically focus only on eviction. For example, LBSC [12] determines which items to remove once the cache becomes full, while admitting all new items by default. This strategy leads to inefficient cache utilization, i.e., low-frequency or even one-time access data can occupy valuable cache space, causing frequent evictions and wasted I/O. In contrast, cloud databases require coordinated control over admission and eviction to ensure that only high-value data are retained in the cache.

Recently, numerous learning-based algorithms have been developed to enhance the adaptability of caching in traditional databases [6], [13]–[15], typically by mimicking the Belady [16] oracle algorithm. However, most of these methods still lack cost-awareness and consider only evictions. Effective caching in cloud databases entails making two interdependent decisions simultaneously: *what to admit* (i.e., whether an incoming item should enter the cache) and *what to evict* (i.e., which cached item should be replaced when the cache is full). These decisions influence each other: overly conservative admission underutilizes the cache space, leading to more frequent evictions. Thus, a unified learning-based framework that jointly optimizes admission and eviction while considering the data cost is essential. However, such integration is non-trivial due to the following challenges:

(1) The absence of an oracle algorithm for joint admission and eviction. Similar to previous learning-based caching algorithms [14], [15], [17], incorporating an oracle algorithm is essential for cloud-native databases. However, it is difficult to find an optimal oracle algorithm when considering data fetching costs with joint admission and eviction. In fact, finding the optimal caching policy in such case is known to be an NP-hard problem [18].

(2) The efficiency in utilizing learning models. Although current learning-based algorithms perform well with changing workloads, they often require continuous retraining and extensive model inferences [14], [17], resulting in significant computation overheads, i.e., consuming a significant amount of CPU resources. Consequently, the extra cost of introducing learning-based models may lead to an even worse performance for the database systems.

To address these challenges, we propose $LBSC^+$, an enhanced learning-based cost-aware caching framework built upon our previous LBSC design. This framework utilizes a lightweight supervised learning-based model that learns from our proposed oracle algorithm to jointly predict the admission probability of incoming requests and the eviction probability of cached data items, achieving globally optimized cost-aware caching. Specifically, we propose an approximately optimal oracle algorithm called BeladySizeCost, to tackle the first challenge. The basic idea is to retain data items with high cost per byte and potential near-future accesses, thereby adapting to changing workloads while considering the costs. To address the second challenge and enhance the efficiency of our framework in practical applications, we employ a single shared model for both admission and eviction predictions. We further propose an adaptive inference-reusing technique that dynamically determines which previous predictions to reuse, along with lightweight optimizations such as data sampling and model reuse, effectively reducing the number of model inferences without compromising accuracy.

To summarize, we make the following key contributions in this work:

- We update a learning-based cost-aware caching framework named $LBSC^+$, which jointly optimizes admission and eviction policies for cloud databases, achieving faster query execution and robust performance under dynamic workloads.
- We propose BeladySizeCost, a cost-aware oracle algorithm that guides both admission and eviction decisions in our learning-based framework. We demonstrate both theoretically and experimentally that it is optimal under the assumption that the cache is full upon request arrival.
- To further enhance efficiency, $LBSC^+$ adopts a single shared model for both admission and eviction predictions. On top of this unified design, we develop an adaptive inference-reusing mechanism that dynamically determines which previous predictions to reuse, significantly cutting down redundant inferences without sacrificing accuracy. Additionally, lightweight techniques such as adaptive sampling and model reuse are incorporated to make $LBSC^+$ more efficient.
- We implement an $LBSC^+$ simulator and integrate $LBSC^+$ into an underlying cloud database. Our evaluation in real-world cloud databases shows that $LBSC^+$ outperforms the state-of-the-art (SOTA) algorithms by up to 25%, which is a significant improvement in terms of caching compared to similar works [8]. Additionally, our proposed optimizations achieve a computation overhead reduction of up to 50%.

TABLE I
PERFORMANCE METRICS USED IN CACHE REPLACEMENT POLICIES

Metrics	Definition
Miss Rate	$\frac{\sum_{i \in Hit} 0 + \sum_{j \in Miss} 1}{\sum_{i \in Hit} 1 + \sum_{j \in Miss} 1}$
Byte Miss Rate	$\frac{\sum_{i \in Hit} 0 + \sum_{j \in Miss} s_j}{\sum_{i \in Hit} s_i + \sum_{j \in Miss} s_j}$
Total Cost	$\sum_{j \in Miss} \mathcal{C}_j$

TODO: The rest of the paper is organized as follows. Section II explains why we need a new metric for caching algorithms. Section III introduces BeladySizeCost algorithm and gives a theoretical proof. Section IV and V detail the $LBSC^+$ framework. System integration is discussed in Section VI. Section VII evaluates the performance of our framework compared to the baselines. We review the related work in Section VIII and conclude the paper in Section IX.

II. Cost-aware Metric in Caching

In this section, We review existing cache performance metrics (Table I) and explore the need for a new cost-aware metric for cloud environments. The miss rate metric, used in traditional databases like MySQL [19] and PostgreSQL [20], ignores variable item sizes and transferring costs, making it unsuitable for cloud databases. Another metric is the byte miss rate, which improves by considering size but neglects computation costs. To consider both transfer and computation costs, we define a new metric, named total cost (Eq 1), to better measure the performance of caching algorithms in cloud databases. The detailed discussion can be found in the technical report [**?**].

$$\mathcal{C}_i = \mathcal{CC}_i + \mathcal{TC}_i \tag{1}$$

In the equation, $\mathcal{CC}_i$ presents the computation cost, which signifies the time required for calculating the cached data (e.g., intermediate results of the query). $\mathcal{TC}_i$ denotes the transfer cost, which reflects the time to transfer data between two storage media, and it is proportional to the data size. For ease of representation, we use $\mathcal{C}$ to denote the total cost, with i referring to the data index. It is worth noting that the total cost $\mathcal{C}$ refers to the time of fetching data from remote storage to the local cache.

III. BeladySizeCost Algorithm

The Belady algorithm [16] serves as the canonical oracle for traditional learning-based caching systems [14], [21]–[23]. To adapt to cloud databases, we propose BeladySizeCost, an approximately optimal algorithm designed to minimize the total cost $\mathcal{C}$ through coordinated admission and eviction decisions under dynamic workloads.

The core component of a caching algorithm is the ranking function, prioritizing which data items to cache and which to evict [21]. For BeladySizeCost, we aim to design a ranking function with the following desired properties:

P1. The function should naturally incorporate the data cost $\mathcal{C}_i$. Retaining data with high cost per byte is more effective.

P2. The function should be computationally efficient and capable of online tracking, without requiring the entire workload in advance.

P3. When the total cost $\mathcal{C}_i$ of each data item is uniform (i.e., all items have the same fetching costs), the algorithm should reduce to a variant of the Belady policy that jointly handles admission and eviction. In this case, when the cache is full, the policy evicts the data item with the farthest future access among candidates, including all data items and the newly requested one. If the new request itself corresponds to the farthest future access, it is bypassed and not admitted into the cache.

Given these properties, the BeladySizeCost algorithm evicts the data item with the lowest value $\frac{\mathcal{C}_i \cdot p_i}{s_i}$ when the cache is full. The eviction candidates are chosen from all items in the cache together with the newly requested one. Note that p_i is the probability that data item i will be accessed in the future (hit probability), s_i is the data size.

In BeladySizeCost, the ranking function considers the data cost, and it makes decisions without requiring the knowledge of the entire workload, satisfying properties P1 and P2. Next, we perform an asymptotic analysis of BeladySizeCost to establish its correctness.

The theoretical optimality of BeladySizeCost can be established by considering a dynamic programming formulation that minimizes the total cost. Let $\mathcal{T}_t(S_t)$ represent the set of possible cache states at time t, given the current cache state S_t and a new data request α_{t+1}. The set of feasible cache states after eviction can be written as:

$$\mathcal{T}_t(S_t) = \{S \subseteq (S_t \cup \alpha_{t+1})\}, \tag{2}$$

where S denotes any cache state in $\mathcal{T}_t(S_t)$. When the cache is full, the algorithm must decide whether the newly requested item α_{t+1} should be admitted into the cache, which is denoted as the cache admission problem. If $S = S_t$, it indicates that α_{t+1} is not admitted; otherwise, α_{t+1} replaces one or more items in the cache.

In cloud databases, it is common that the sizes of cached items are relatively small compared to the total cache size. Therefore, we have the following lemma.

Lemma 1. *The exchange between the evicted item(s) and item α_{t+1} does not affect the size of the entire cache, i.e., $Size(S) = Size(S_t)$.*

The problem of minimizing the total cost for joint cache admission and eviction can be formulated as an optimization problem that resembles the knapsack problem, which is known to be NP-complete. The objective is to maximize the overall cost benefit of the items retained in the cache while staying the capacity constraint:

$$\begin{aligned} &\max_{i \in \mathcal{T}_t(S_t)} \mathcal{C}_i \cdot p_i \\ s.t. &\sum_{i \in \mathcal{T}_t(S_t)} s_i \leq CS \end{aligned} \tag{3}$$

where CS denotes the cache capacity. According to Lemma 1, BeladySizeCost guarantee that the items retained in the cache are those with the highest $\mathcal{C}_i \cdot p_i$, thereby minimizing the total cost over the entire workload. The detailed proof of optimality is omitted here due to space limitations, and the complete proof can be found in the technical report [**?**].

To obtain $\frac{\mathcal{C}_i \cdot p_i}{s_i}$ for each data item in the cache, we must estimate its hit probability p_i. Since the size s_i and cost $\mathcal{C}_i$ of each item are known and fixed, p_i can be inferred from the data's locality, i.e., the likelihood of being re-accessed within a short period [21]. Thus, we approximate p_i by the next-accessed probability within a sliding window of future requests:

$$p_i \propto \sum_{x=1}^{N} \frac{1}{x} \cdot \mathbb{I}(age_i^t = t + x) \tag{4}$$

where I(.) is an indicator function, age_i^t stands for the re-access index of the item i under request index t. We use $\frac{1}{x} \cdot \mathbb{I}(true)$ to denote the next reuse probability under the index t. In practice, using only the first next-accessed term provides a sufficiently accurate estimation while satisfying the desired properties P3 and consuming fewer CPU resources. Consequently, we rank the cached data items (including the newly added request) based on $\frac{\mathcal{C}_i \cdot \frac{1}{x} \cdot \mathbb{I}(.)}{s_i}$, and evict the item(s) with the lowest value to restore the cache within the capacity limit.

IV. Design of $LBSC^+$

In this section, we shall present our caching framework $LBSC^+$. We first give an overview of the proposed framework, and then detail main components of $LBSC^+$, respectively.

A. Overview

Figure 1 illustrates an overview of $LBSC^+$, which comprises three main components: Cache Manager, Predictor, and Efficiency Optimizer. The primary objective of $LBSC^+$ is to predict whether a newly requested item should be admitted into the cache and which data items should be evicted when the cache exceeds its capacity.

Cache Manager orchestrates the entire caching workflow. When a new request arrives, this component determines whether it should be admitted into the cache and identifies which cached item(s) should be replaced once the cache exceeds its capacity. The complete process will be detailed in Section IV-B.

Predictor serves as the learning component of $LBSC^+$. It continuously generates training data based on access histories, learns from the oracle policy (BeladySizeCost) using

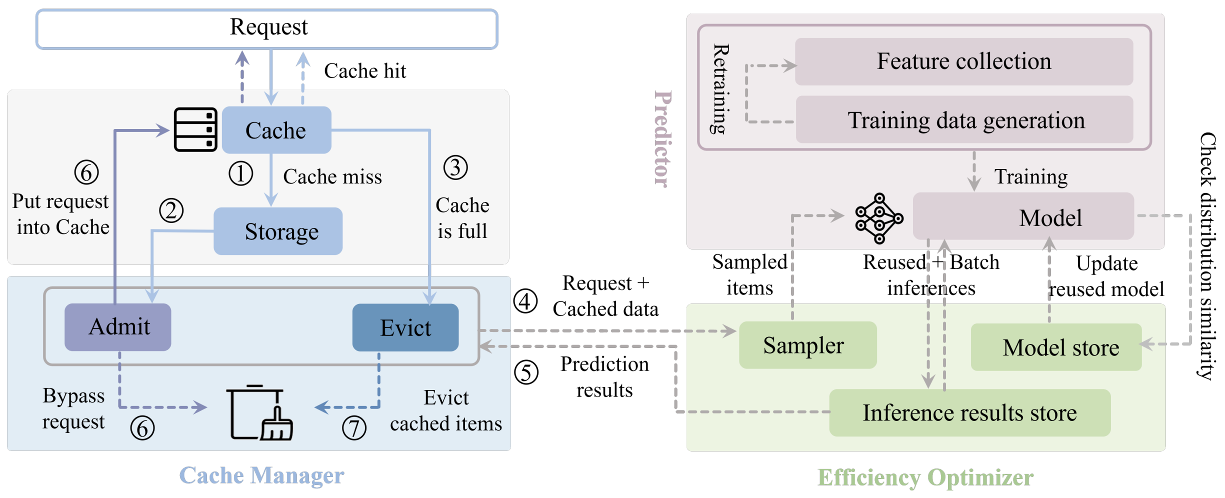


Fig. 1. An overview of the $LBSC^+$ framework.

supervised signals. After training, it jointly predicts admission and eviction probabilities. The model aims to capture varying data-access patterns in cloud environments. We will detail the model training and inference in Section IV-C.

Efficiency Optimizer aims to minimize the computational overhead of the learning-based model. It employs several lightweight yet effective techniques, including inference reuse, adaptive sampling, and model reuse. This component enables $LBSC^+$ to maintain high prediction accuracy while significantly reducing CPU consumption during online decision-makings. We will elaborate on these optimizations in Section V.

B. Cache Manager

This component is responsible for orchestrating the entire cache pipeline, encompassing both admission and eviction decisions. Previous caching algorithms primarily focus on eviction [6], [14], [21], determining which items to remove once the cache becomes full while admitting all new items by default. This strategy underutilizes the limited cache space: if an incoming item has a very low access probability in the future or even requested only once, caching it brings little benefit and may trigger unnecessary evictions. By jointly optimizing admission and eviction, $LBSC^+$ effectively filters out such low-frequency data before it enters the cache, improving space utilization and reducing eviction frequencies.

The detailed workflow is illustrated in Algorithm 1. When a new request α_{t+1} arrives, the $LBSC^+$ framework first checks whether the requested data item already exists in the cache. If it results in a cache hit (line 3), the corresponding item's features are updated to generate the new training data (as described later in Section IV-C), and the cached data returned to the user (lines 4-6). If it is a cache miss, the framework enters the joint admission-eviction stage.

In this stage, if the cache is not full, the new request α_{t+1} is directly inserted into the cache and return (lines 8-11). To adapt to the changing workloads, we continuously generate training data and retrain the model once accumulated data

Algorithm 1 $LBSC^+$

```
 1: Initialize corresponding thresholds: $T_{\mathcal{R}}, T_1, T_2, T_{JS}$
 2: for data requests $\alpha_1, \alpha_2, \ldots, \alpha_{t+1}, \ldots$ do
 3:   if $\alpha_{t+1}$ is in cache then
 4:     update_features($\alpha_{t+1}$)
 5:     $D^{t+1}$ = generate_training_data($\alpha_{t+1}$)
 6:     return
 7:   else
 8:     if not is_cache_full($\alpha_{t+1}$) then
 9:       put $\alpha_{t+1}$ into the cache
10:       return
11:     end if
12:     if calculate_all_JS() $> T_{JS}$ then
13:       re_train()
14:     end if
15:     put $\alpha_{t+1}$ into a pool $G_{tmp}$ with cached data
16:     while is_cache_full() do
17:       inference_reuse()
18:       sample data using dynamic_splitting($T_{\mathcal{R}}, T_1, T_2$)
19:       batch_model_inference()
20:       admission_eviction($G_{tmp}$)
21:     end while
22:     return
23:   end if
24: end for
```

exceeds a predefined size. To reduce unnecessary retraining, we introduce a strategy that reuses previous models without compromising accuracy (lines 12-14). The model training and the model reuse strategy will be detailed in Sections IV-C2 and V-C separately. When the cache is full, the new request α_{t+1} is temporarily added to the candidate pool together with all existing cached data items (line 15). The trained model then evaluates every item in this pool and outputs a priority score, representing the predicted long-term utility of each item based on both cost and access probability learned from the oracle BeladySizeCost policy. To make the learning process more lightweight, we introduce adaptive sampling and inference reusing techniques to reduce the number of model inferences (lines 17-18). The details of these methods are presented in Sections V-A and V-B. Instead of evicting only existing cached items, $LBSC^+$ considers both the new request and cached data items in a unified ranking list. The item(s) with the lowest score are evicted to restore the cache within its capacity limit. Consequently, if the new request α_{t+1} has the lowest predicted utility, it is bypassed and never admitted the cache; otherwise, one or more existing items are evicted to make room for the new request (lines 19-20). Note that the other procedures in Algorithm 1 will be introduced in the subsequent sections.

Importantly, in $LBSC^+$, both admission and eviction decisions share the same model because their objectives are inherently consistent: both aim to maximize the overall cost-benefit of the cached items under dynamic workloads. Using a unified model ensures that admission and eviction are guided by the same learned utility function, achieving globally optimized, cost-aware cache management.

C. Predictor

Model is the learning component of $LBSC^+$, responsible for training and maintaining the shared model that supports joint admission and eviction decisions. It encompasses three major stages: feature extraction, model training, and model inference. In addition, an outlier filtering mechanism is integrated to enhance training stability and accuracy.

1) Feature Extraction: Since features play a crucial role in machine learning [24]–[27], we carefully select the following features, inspired by the recent state of the arts [6], [14].

Frequency. The number of re-accesses for each item.

Delta. The amount of access between two consecutive requests for a specific item. Specifically, $Delta_1$ represents the number of access since the last request, $Delta_2$ indicates the number of access between the item's previous two requests, and so forth. Figure 2(a) shows an example[1], where the items colored green indicate the access of the same item, and their $Delta$ features are $Delta_1 = I_3 - I_2$, $Delta_2 = I_2 - I_1$, and so on, where I_1, I_2, I_3 denote the request indexes.

Domain Features. To tailor the cache for specific systems, such as cloud databases, we incorporate domain knowledge as additional features to enhance the learning model. These features remain unchanged once the item is loaded into the cache. We focus on these static features because they often capture workload characteristics. For instance, in cloud databases, the cached items have varying sizes, and each item has its own metadata, including table name, column name, partition index, etc.

2) Model Training: After extracting features, we need to generate training data and train a supervised learning model.

Training data generation. We first introduce how to generate labels for the training data. Once a new request arrives, a

[1]The detailed description of the components in Figure 2 will be introduced in Section IV-C2.

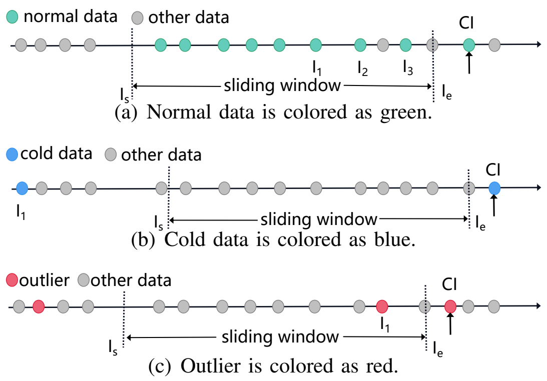


Fig. 2. Generation of three types of training data using a sliding window.

sliding window[2] is employed to generate the corresponding label. Figure 2 shows the concept of sliding window $[I_s, I_e]$. The length (or size) of the sliding window represents the maximum number of requests it covers, i.e., $I_e - I_s$, denoted as L_e^s for simplicity. Initially, when the number of requests exceeds L_e^s, we start generating training data for each new request. After generation, the sliding window moves forward so that I_e points to the current request (i.e., CI shown in Figure 2). Additionally, we maintain a separate history list to retain features of the items that appear in the sliding window but do not exist in the cache. Therefore, if the requested item is in the history list or the cache, the difference between the current index and its last access index becomes the label. Otherwise, if the item's last request falls outside the sliding window, it is labeled as cold data and assigned a large value ($2 * L_e^s$ in our implementation) to indicate infrequent access. For instance, in Figure 2(a), the label of the data colored green is "$CI - I_3$" when a new request comes. At CI in Figure 2(b), we can see the item colored blue comes in, however, its last access I_1 exceeds the L_e^s, so we label it as "$2 * L_e^s$". After labeling, the sliding window moves forward to generate the next training data. To further improve the model performance, we propose a simple but efficient method to filter abnormal training data shown as follows:

Outlier detection on training data. We continuously generate new training data, yet not every data is effective for the model. Such data are outliers and can be pruned to further improve the model performance. For example, in Figure 2(c), we observe that the red-colored data is not accessed for a long time before I_1, indicating that it belongs to "cold" data. However, we erroneously label it as $CI - I_1$, following the previously introduced method, which suggests that it will be accessed after a short time. We treat such data as outliers, and the detecting procedure is formally defined as follows:

Definition 1. *Outlier Detection. Given L_e^s, the training data D^i with label y^i and corresponding last reuse $Delta_1^i$. For every generated training data D^i, it is an outlier if: $y^i < L_e^s$ and $Delta_1^i > L_e^s$. When we detect the training data as an outlier, we remove it from the training data set.*

[2]The concept of sliding window has been proposed in previous works [14], [17], [28]. We refer interested readers to these works for the effect of sliding window size on model accuracy and efficiency.

Model training. After generating sufficient training data, we proceed to train a prediction model. $LBSC^+$ uses gradient boosting machines (GBM) [29] because tree models do not require feature normalization and can train concurrently, resulting in reduced CPU resource consumption. Also, GBM has proven to work effectively in previous works [14], [22], [30]. To adapt to the changing workload, $LBSC^+$ retrains the model once collecting enough training data by default. However, continuous training may produce unnecessary computation overheads, hence we propose to reuse previously-trained models in Section V-C.

3) Model Inference: As noted in Section IV-B, when the cache reaches its capacity limit, $LBSC^+$ performs model inference to determine whether the new item to admit into the cache and which cached items to evict. It is worth mentioning that some previous work [15] makes a prediction upon a new request, treating it as a probability that measures the likelihood of future re-access when eviction. While this approach reduces computation overheads, the probability becomes less accurate over time as the system runs. In contrast, our method improves both admission and eviction accuracy but incurs higher computation overheads [22].

Since the major computation overhead comes from the inferences of the newly requested data and cached data items, we employ several techniques to reduce the total number of model inferences. First, we introduce an adaptive inference-reusing mechanism that dynamically determines which previous predictions to reuse across multiple model inference cycles. We design a data structure to store previous prediction results, allowing $LBSC^+$ to efficiently reuse them when triggering multiple evictions. Furthermore, we utilize a sampling-with-replacement strategy on the cache items before the model inference. The motivation of the sampling is majorly due to the large amounts of the caching items – as reported in [9], the cache usually consists of hundreds of thousands of items in production environments. After the sampling, we conduct the inference on the sampled items and select the items to evict based on the inference results. Finally, we find a specific property under the cost-aware setting, which allows us to further reduce the computation overheads when making model inferences. We illustrate the details in Section V.

V. Efficiency Optimizations

In this section, we present several optimizations applied during the model training and inference phases. These optimizations aim to reduce the computational overhead of $LBSC^+$.

A. Inference reusing

In the learning-based caching, model inference must be invoked frequently to support admission and eviction decisions, which causes it to dominate the overall computational overhead. To mitigate this issue, we propose an inference reusing mechanism in this section. Across consecutive cache decisions, most cached items remain unchanged, and corresponding predictions remain valid because the relative ordering among cached items does not immediately become stale.

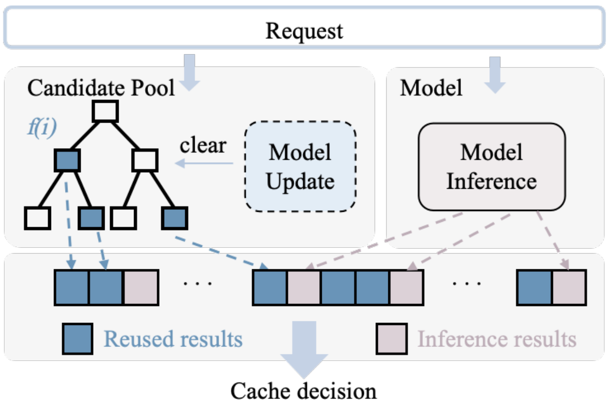


Fig. 3. The inference reusing process of $LBSC^+$.

This observation provides an opportunity to reuse previously computed inference results rather than recomputing them for every cache decision.

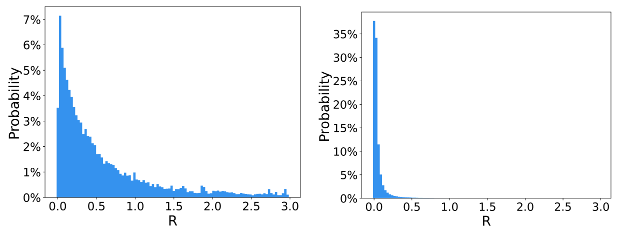
The static reuse strategy, i.e., reusing fixed number of predictions in every cache decision, performs poorly under dynamic workloads. When the number of reused predictions is too large, the system may reuse stale results and make suboptimal cache decisions; when the number is too small, many reusing opportunities are lost and the inference overhead remains unnecessarily high.

To this end, $LBSC^+$ selectively reuses previously computed prediction results, dynamically determining when and how much to reuse. The central idea is to maintain a candidate pool $\mathcal{P}$ that stores the predicted scores of cached items from previous cache decisions. Whether a prediction can be reused is determined by its freshness, which is computed as follows:

$$f(i) = \frac{score_i}{\log(1 + pred_i)} \tag{5}$$

where $score_i$ is the model's predicted score for item i, and $pred_i$ denotes the elapsed time since item i's last model inference, indicating how stale its previous prediction has become. Note that the denominator applies a logarithmic transformation. Because the next-access prediction may vary across several orders of magnitude, we apply a $log(.)$ function to narrow this variation in the model. The logarithmic transformation in Equation 5 therefore stabilizes the scale of the freshness metric. A large $f(i)$ suggests that the prediction result of cached item i remains sufficiently fresh and can be safely reused.

With the freshness metric, $LBSC^+$ performs inference reusing as illustrated in Figure 3: when a new request arrives, $LBSC^+$ first computes the freshness score $f(i)$ for each item in the candidate pool $\mathcal{P}$ and determines how many predictions can be reused. Specifically, the items $\mathcal{P}$ are sorted in descending order of their freshness scores $f(i)$, and items with $f(i) > 1$ are selected for reuse across consecutive decisions. During this process, previously reused predictions are applied directly to admission and eviction decisions without invoking the model. In the original model, each cache decision would require n model inferences. If $LBSC^+$ identifies k items to be reused, only the remaining $n - k$ items require



(a) Total workload. (b) BeladySizeCost evictions.

Fig. 4. Distribution of $\mathcal{R}$ between the total workload and BeladySizeCost evictions.

model inferences. Whenever the model is retrained, all reused predictions in $\mathcal{P}$ become invalid and need to be rebuilt. In addition, if any item in $\mathcal{P}$ is accessed again, its score and freshness must be immediately updated.

To efficiently manage the reuse metadata (i.e., freshness with corresponding elapsed time), $LBSC^{+}$ employs a red–black tree structure [31] that stores this information, indexed by $f(i)$. When triggering multiple evictions, the tree enables fast lookup in $O(log\ n)$ and incremental updating of low-utility items.

B. Sampling with cost-size threshold

To further reduce the number of samples for model inferences, narrowing the scope of sampling is a reasonable way. We first introduce the ratio of cost to the size of every item as a metric. Formally, for every item i, we define the ratio R_i as follows:

$$\mathcal{R}_i = \frac{\mathcal{C}_i}{s_i}. \tag{6}$$

Figure 4 displays the distribution of $\mathcal{R}$ for a specific workload (see Section VII-A), as well as the corresponding evictions conducted by BeladySizeCost. While the ratio $\mathcal{R}$ for the entire workload spans a wide range (ranging from 0.0 to 3.0 in Figure 4(a)), BeladySizeCost typically evicts data with small $\mathcal{R}$. As shown in Figure 4(b), over 99% of data's $\mathcal{R}$ falls between 0.0 and 1.0. This observation aligns with prior works [32], [33] – for general workloads, the item with higher $\mathcal{R}$ (i.e., more cost per byte) leads to a higher likelihood of retention in the cache. Consequently, it is desirable to only sample items with lower $\mathcal{R}$ values when needing model inferences for eviction, which could further save the overhead of the inference. Therefore, we propose the following two methods to narrow the sampling range.

Static Splitting Method (SSM). We divide the cached data into two parts based on a predefined cost-size ratio threshold (denoted by $T_{\mathcal{R}}$) and exclusively sample from the portion whose $\mathcal{R}$ is lower than $T_{\mathcal{R}}$. We formally define SSM in Definition 2, where *baseline* indicates sampling from the entire cached data while keeping other settings unchanged.

Definition 2. *Given $T_{\mathcal{R}}$ and the total number of items in the cache (N_C), the baseline samples $\mathcal{U}_{ori}$ items whenever the cache is full. We split the cached data into two sets $\mathcal{R}^l$ and $\mathcal{R}^h$, which maintain:*

$$\begin{aligned} &\forall \mathcal{R}_i < T_{\mathcal{R}}, i \in \mathcal{R}^l \\ &\forall \mathcal{R}_j \geq T_{\mathcal{R}}, j \in \mathcal{R}^h \end{aligned} \tag{7}$$

Algorithm 2 dynamic_splitting($T_{\mathcal{R}}, T_1, T_2$)

```
1:  $\hat{T_1} = 0$, $\hat{T_2} = 0$.
2:  for data requests $\alpha_1, \alpha_2, \ldots, \alpha_{t+1}, \ldots$ do
3:      if $\hat{T_1} < T_1$ then
4:          sample $\frac{Num(\mathcal{R}^l)*\mathcal{U}_{ori}}{N_C}$ items from $\mathcal{R}^l$.
5:          sample $\frac{Num(\mathcal{R}^h)*\mathcal{U}_{ori}}{N_C}$ items from $\mathcal{R}^h$.
6:          if $E_{\mathcal{R}} > T_{\mathcal{R}}$ then
7:              $T_{\mathcal{R}} = T_{\mathcal{R}} + 0.1$, $\hat{T_1} = 0$.
8:          end if
9:          $\hat{T_1} = \hat{T_1} + 1$.
10:     else
11:         sample data from $\mathcal{R}^l$ only, $\hat{T_2} = \hat{T_2} + 1$.
12:         if $\hat{T_2} > T_2$ then
13:             $\hat{T_2} = 0$.
14:             sample data from $\mathcal{R}^l$ and $\mathcal{R}^h$ separately.
15:             if $E_{\mathcal{R}} > T_{\mathcal{R}}$ then
16:                 $T_{\mathcal{R}} = T_{\mathcal{R}} + 0.1$.
17:             end if
18:         end if
19:         $\hat{T_2} = \hat{T_2} + 1$.
20:     end if
21: end for
```

Our SSM only samples $\frac{Num(\mathcal{R}^l)*\mathcal{U}_{ori}}{N_C}$ items from $\mathcal{R}^l$ whenever the cache is full, while the total cost $\mathcal{C}$ of SSM still (approximately) converges to the baseline. Note that $Num(\mathcal{R}^l)$ indicates the number of items in the set $\mathcal{R}^l$.

Our method needs to ensure that the total cost $\mathcal{C}$ is almost the same as the *baseline*. Thus, the choice of $T_{\mathcal{R}}$ is crucial. If $T_{\mathcal{R}}$ is too small, some items that should be evicted will never be sampled, potentially increasing the total cost $\mathcal{C}$. Conversely, if $T_{\mathcal{R}}$ is too large, $\mathcal{R}^l$ will include more items, which may not help reduce the computation overheads. Thus, this method's effectiveness relies on knowing $\mathcal{R}$'s distribution of the workload in advance.

Dynamic splitting method (DSM). In production systems, we can hardly know $\mathcal{R}$'s distribution of the entire workload, making the SSM method impractical. To overcome this challenge, we further propose a dynamic splitting method described in Algorithm 2. At the beginning of the workload, we sample data proportionally from $\mathcal{R}^l$ and $\mathcal{R}^h$ and increase the value of $T_{\mathcal{R}}$ if the evicted item's $\mathcal{R}$ (i.e., $E_{\mathcal{R}}$) is greater than $T_{\mathcal{R}}$ (lines 4-9). The rationale is to prevent a small initial $T_{\mathcal{R}}$ from significantly affecting the workload's $\mathcal{C}$. For a continuous period, if all evicted items are less than the $T_{\mathcal{R}}$ (i.e., when $\hat{T_1} \geq T_1$), we argue that sampling only from $\mathcal{R}^l$ will not impact on the total cost $\mathcal{C}$. Consequently, we start to exclusively sample from $\mathcal{R}^l$ (line 11), using the same sampling number as in SSM. In real-world environments, $\mathcal{R}$'s distribution may vary within a workload. To adapt to the changing workload, we need to sample data from the entire cache after a period of time (i.e., T_2), and continuously increase the value of $T_{\mathcal{R}}$ if the evicted data's $\mathcal{R}$ is greater than $T_{\mathcal{R}}$ (lines 12-18). This procedure prevents starvation since it enables items from $\mathcal{R}^h$ have the chance to be predicted by the model. Notably,

$T_{\mathcal{R}}$ monotonically increases in our algorithm, ensuring robust performance w.r.t. the total cost $\mathcal{C}$. This is because our method can hold $\mathcal{R}^l$ with almost all data that should be evicted.

C. Model reusing

While continuous training can adapt well to changing workloads, it is unnecessary to frequently retrain the model. If the distribution of the current batch of training data is similar to one of the previous batches, we can the corresponding model to reduce the retraining overheads. In this paper, we define a batch as a set of consecutive generated training data whose count matches the predefined batch size.[3]

From [17], we know that the access pattern of typical workloads follows a standalone Zipfian distribution in production environments. Therefore, we model the access patterns of different batches as standalone Zipfian distributions. To measure the similarity of two batches, i.e., the similarity of their distributions, we use Jensen-Shannon (JS) divergence [34] defined as follows.

Definition 3. *The JS-divergence distance between two probability distributions $\vec{p} = (p_1, ..., p_n)^T \geq 0$ and $\vec{q} = (q_1, ..., q_n)^T \geq 0$ is defined as:*

$$JS(\vec{p}, \vec{q}) = \frac{1}{2}(KL(\vec{p}\|\frac{\vec{p}+\vec{q}}{2}) + KL(\vec{q}\|\frac{\vec{p}+\vec{q}}{2})), \tag{8}$$

where

$$KL(\vec{p}\|\vec{q}) = \sum_{i=1}^{m} p_i \log(\frac{p_i}{q_i}). \tag{9}$$

The Zipfian distribution follows $p_i = A/i^\alpha$, with A being the normalization factor and α being the Zipfian parameter. Therefore, it is natural to directly estimate the two parameters to compute JS divergence. For instance, a Least-Squared-Method based model could be employed to estimate the parameters [17]. However, such a method introduces extra computation overheads in practice, and the estimated parameters often have deviations. Consequently, we directly sample the training data along with their corresponding frequencies to perform the computation in our implementation.

In summary, when needing to retrain the model, we first sample some data items from current batch and calculate the corresponding JS divergence based on the previous batches. If all values of the JS divergence are greater than a specific threshold (i.e., T_{JS} in Algorithm 1), we retrain the model and store it. Conversely, we reuse the model whose corresponding JS divergence is the lowest. More details can be found in our technical report [?].

VI. Integration with Cloud Databases

Our framework is general for integrating into any cloud database systems. To demonstrate the feasibility of integration, we first illustrate the process using the FPDB system as an example and then discuss its extension to current commercial databases.

[3]We leave the research to better split the training data into batches and set a proper batch size for our future work.

A. Integration with FPDB

FPDB [8] is a cloud analytical database that adopts a storage-disaggregation architecture. It stores original data on the storage node using AWS S3 [10] or MinIO [35] and supports predicate push-down to reduce the network traffic. The basic caching unit, called a segment, corresponds to a specific column within a table partition, and its size varies depending on the schema. The cache algorithm in FPDB is called WLFU. It assigns weights to each segment based on transfer and computation costs and evicts the one with the lowest weight.

We replace WLFU with our $LBSC^+$ framework. Because a segment's access cost may change when only part of it is used, we estimate its effective cost dynamically as follows:

$$\mathcal{C}_{new} = \frac{\mathcal{C}_{seg} + \mathcal{C}_{old} * (num - 1)}{num}. \tag{10}$$

where $\mathcal{C}_{seg}$ is the cost of the current access, $\mathcal{C}_{old}$ is the previous estimation, and num denotes the number of accesses. Intuitively, the estimated cost of an item is calculated as the average of its history data fetching costs, which yields accurate cost estimation with negligible runtime overhead, as validated in Section VII.

B. Extension to Commercial Cloud Databases

The modular design of $LBSC^+$ enables its integration into mainstream cloud databases such as OceanBase [36], TiDB [37], or PolarDB [38]. However, several practical challenges must be addressed for effective deployment:

Heterogeneous Caching Granularity. Different cloud databases cache data at varying levels of granularity: some store final query results, others cache intermediate results or raw table data. Such diversity makes direct integration of $LBSC^+$ potentially inaccurate. Therefore, its cost modeling must be adapted according to the cached data type. For example, in systems with uniform block sizes, the size factor can be ignored from the model.

Feature Collection Overheads. Some systems do not explicitly maintain fine-grained access metadata required by the model (e.g., per-segment hit frequency or cost). In such cases, lightweight monitoring hooks can be embedded into the query executor to capture essential features.

Overall, integrating $LBSC^+$ into commercial cloud databases requires modest architectural changes, preserving compatibility with existing components. Once deployed, $LBSC^+$ provides an adaptive, cost-aware caching capability, effectively improving throughput and reducing latency in real-world cloud environments.

VII. Evaluation

In this section, we shall present the experimental evaluation of $LBSC^+$ and compare its performance against state-of-the-art baselines.

TABLE II
TWO SYNTHETIC DATASETS USED IN THE EVALUATION.

Dataset		$Wiki_1$	$Wiki_2$
Unique Requests(million)		11.1	7.43
Total Bytes(TB)		3.79	3.15
Unique Bytes(GB)		1071	854.95
Data Size	Max(MB)	557.89	674.25
	Min(Byte)	42	10
	Mean(MB)	0.09	0.12

A. Experimental Setup

Testbed: We conduct experiments with real cloud databases on Alibaba Cloud ECS [39] compute-optimized instance, computation and storage node with 16vCPU 64GB memory, and the network bandwidth is 5Gbps. All servers run on the Ubuntu 20.04 operating system. Experiments with synthetic datasets are on a server featuring Intel Xeon W-2275 3.30GHz 14-core CPUs with 512GB main memory.

Datasets and Benchmark: *Synthetic Datasets*. We first utilize two traces [14] to evaluate the performance of $LBSC^+$ in simulator environments. They are collected from a west-coast node, serving photos and other media content for Wikipedia pages. Table II provides the statistical details of these workloads.

Based on the original datasets, we generate the data cost for the items, including the computation cost and transfer cost mentioned in Eq 1, to simulate the cloud environment. In detail, we rewrite the transfer cost $\mathcal{TC}_i$ as $k * s_i$ for each data item i. Recall that k presents network transmission speed calculated by real network bandwidths and s_i denotes the size of the data item. As for the computation cost $\mathcal{CC}_i$, we uniformly sample from a predefined range and assign $\mathcal{CC}_i$ according to the sampling results. In our experiments, we conduct thorough comparisons under different settings and signify the specific settings in each use case.

Star Schema Benchmark (SSB) [40]. We then implement $LBSC^+$ in FPDB and evaluate the performance using the same benchmark, Star Schema Benchmark (SSB), as implemented in FPDB [8], which is similar to TPC-H [41]. The SSB dataset consists of five tables (four dimension tables and one fact table). The fact table is much larger than the dimension tables. Queries are generated based on three randomly selected query templates with parameters in the filter predicates chosen from a specified range of values following a Zipfian distribution [42]. In our experiments, we generate seven types of queries with skew factors ranging from 0.0 to 2.0, and randomly concatenate these queries to simulate the changing workload. Other configurations remain consistent with FPDB.

Baselines: We compare $LBSC^+$ with eight baselines, including both heuristic-based caching algorithms and machine learning-based methods. The descriptions of baselines are as follows.

- *LFU* [43]. This method evicts the item with the least access frequency.
- *LFUDA* [44]. LFUDA is an extension of LFU, where an item's priority key is determined by its frequency plus the cache age.
- *LRUK* [7]. Note that LRU evicts the item that is the least recently used. LRUK is an extension of LRU that utilizes k LRU stacks.
- *Belady* [16]. It evicts the item with the furthest access.
- *GDSF* [32]. It ranks the cached data by $rank_i = f_i * \mathcal{C}_i/s_i + L_i$, where f_i represents access frequency, $\mathcal{C}_i$ is the cost, L_i is the rank of the last evicted item. In FPDB, the caching algorithm WLFU is similar to GDSF, but it does not consider the L_i in its ranking function.
- *LHD* [21]. It models the hit density based on age and size and evicts the item with the lowest hit density.
- *CACHEUS* [6]. It uses regret minimization [45], allowing the dynamic selection of one of two expert policies upon a cache miss. The original algorithm is only suitable for data with uniform size. We re-implement CACHEUS to adapt variable-size data.
- *LRB* [14]. It is a learning-based caching algorithm that mimics the offline Belady algorithm. It uses an ML model to predict the next request time of the cached data.

Metrics: Our main metric is the total cost $\mathcal{C}$ defined in Section II. Due to the diversity of the workloads, the total cost $\mathcal{C}$ can vary widely. Hence, for ease of visual presentation, we calculate every caching algorithm's $\mathcal{C}$ and report them compared to GDSF (for its popularity with cost-aware consideration) using the metric: $\frac{\mathcal{C}_{GDSF}-\mathcal{C}_{algorithm}}{\mathcal{C}_{GDSF}}$. Each experiment starts with a cache warm-up phase where no metrics are recorded. For FPDB datasets, the metric is the end-to-end execution time of all queries. Furthermore, the metric in Section VII-B3 is the computation overhead (or cost) reduction compared to the $baseline$. Note that the $baseline$ in this section indicates the default $LBSC^+$ without optimizations.

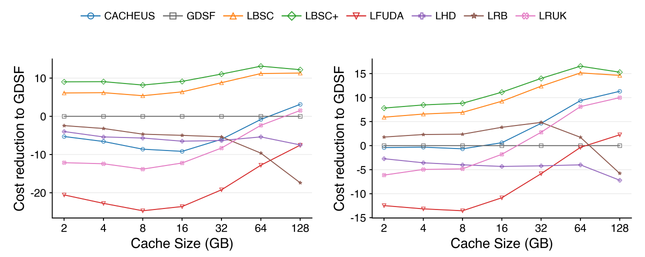
B. Performance Evaluation

We present the following experimental studies in this subsection. First, we compare the overall performance of $LBSC^+$ with the state-of-the-art designs under various workloads and cache sizes using synthetic datasets. Then, we demonstrate the significant performance improvement achieved by $LBSC^+$ when used in cloud databases (i.e., FPDB). Finally, we conduct a detailed analysis of the effectiveness of optimizations.

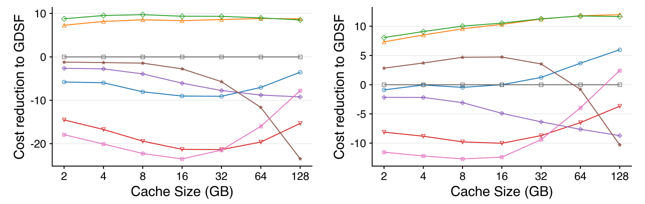
1) Performance on Synthetic Datasets:

We generate $\mathcal{C}_i$ for each item according to Eq 1. To thoroughly compare the performance of different algorithms, we adjust each parameter in Eq 1 to imitate various scenarios. Figure 5 illustrates the cost reduction of the evaluated algorithms to GDSF, with different cache sizes using two types of cost settings. Note that the transfer costs are calculated based on real network traffic.

We find that $LBSC^+$ consistently outperforms all state-of-the-art algorithms, achieving the lowest $\mathcal{C}$ for all trace and cache size combinations. Overall, $LBSC^+$ reduces the total cost $\mathcal{C}$ by 8%-13%. Notice that no prior caching algorithm can consistently improve the performance when the cache size increases. GDSF performs best when the computation cost dominates (i.e., Figures 5(a) and 5(c)), while LRB demonstrates optimal performance when the transfer cost dominates.

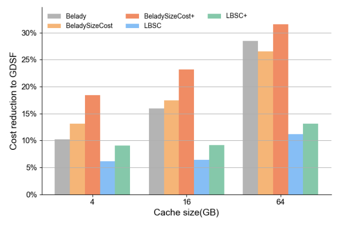


(a) $Wiki_1$ dataset with the range of $\mathcal{CC}_i$ is 10240, k is 250
(b) $Wiki_1$ dataset with the range of $\mathcal{CC}_i$ is 102, k is 25000

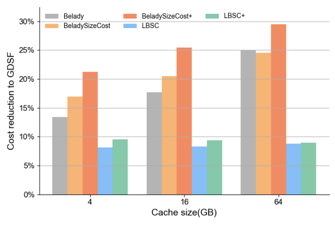


(c) $Wiki_2$ dataset with the range of $\mathcal{CC}_i$ is 10240, k is 250
(d) $Wiki_2$ dataset with the range of $\mathcal{CC}_i$ is 102, k is 25000

Fig. 5. The cost reduction to GDSF w.r.t varying cache sizes for $LBSC^+$ and the six algorithms on two datasets. On each dataset, the computation cost dominates in the sub-figures (a) and (c), and the transfer cost dominates in the sub-figures (b) and (d).



(a) Workload-1



(b) Workload-2

Fig. 6. Performance comparison among $LBSC^+$, BeladySizeCost and Belady.

Specifically, when the transfer cost dominates, the trend of the byte miss rate closely aligns with the total cost $\mathcal{C}$. LRB and CACHEUS opt for byte miss rate, so they have a relatively better performance compared to other algorithms. For instance, LRB improves cost reduction by around 5% over GDSF, as shown in Figure 5(b) and 5(d). Additionally, LRB consistently outperforms CACHEUS with a varying advantage between 1% and 9%, indicating that a learning-based algorithm is superior to a heuristic-based algorithm with diverse access patterns. On the other hand, when the computation cost dominates, the trends of byte miss rate deviate from $\mathcal{C}$. In such cases, GDSF is the best because it considers the cost. Nonetheless, it is still worse than $LBSC^+$ due to its inability to adapt to changing workloads. Among the evaluated algorithms, we observe that LRUK and LFUDA perform the worst in almost all cases. In the worst case, their cost reduction is even below -20%. In addition, we find that LHD is sensitive to cache size. While LHD behaves well in small cache size, its performance decreases as the cache size increases, and becomes even the worst at the cache size of 64G and 128G, as illustrated in Figure 5(b) and 5(d). The evaluations on the other traces show similar results.

So far, we have demonstrated that $LBSC^+$ achieves significant improvements over state-of-the-art algorithms. Since $LBSC^+$ learns from BeladySizeCost, we now evaluate the performance of BeladySizeCost. Figure 6 shows a comparison of the cost reduction among $LBSC^+$, BeladySizeCost, and Belady under two workloads and three kinds of cache size combinations. Similar to Figure 5, the computation cost dominates in Figure 6(a), and the transfer cost dominates in Figure 6(b).

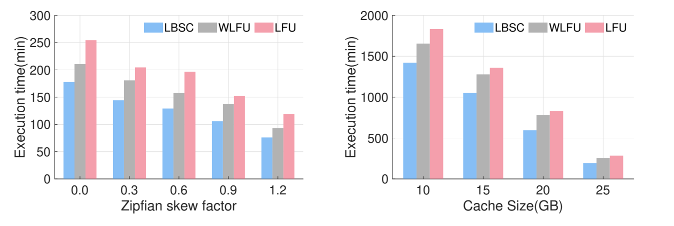
As shown in Figure 6, BeladySizeCost is superior to $LBSC^+$ and Belady in all cases, reducing costs over GDSF by 20% on average. These results confirm that BeladySizeCost is a robust oracle algorithm and is approximately optimal under the assumptions described in Section III. It also shows a desirable upper bound for cost-aware caching algorithms. Moreover, Figure 6 suggests that Belady is unsuitable as an oracle when considering the cost, which is essential in cloud databases.

A noteworthy aspect is that there is a gap between $LBSC^+$ and BeladySizeCost, though $LBSC^+$ learns from BeladySizeCost. This is because the learned model is also affected by other factors, such as generated training data, the number of model inferences, and so on. Nevertheless, we observe from Figure 6(a) that $LBSC^+$ outperforms Belady in terms of cost reduction. This further demonstrates Belady's inferiority as an oracle algorithm, not to mention that it requires prior knowledge of the workload.

2) *Performance of FPDB:*

To evaluate the performance of $LBSC^+$ with real-world cloud databases, we integrate it into FPDB. Figure 7 compares $LBSC^+$ with LFU and WLFU [8]. Specifically, Figure 7(a) presents the execution time of five types of batch queries with a 20GB cache. Each batch size is 600, and they are generated with different Zipfian skew factors (θ). To simulate the changing workload, we concatenate seven types of queries (the corresponding zipfian factors are 0.0, 0.3, 0.6, 0.9, 1.2, 1.5, 2.0, respectively), and the total number of queries is 3900. Figure 7(b) illustrates the execution time of the changing workload with four different cache sizes. It is important to note that the first 50 queries are used to warm up the cache and are excluded from the figure. We generate query datasets using the same setting as [8]. We include table name, column name, and partition id as extra features.

As shown in Figure 7(a), $LBSC^+$ consistently outperforms LFU and WLFU. The most significant cost reduction occurs when θ = 0.3, with $LBSC^+$ outperforming LFU and WLFU by 28% and 19%, respectively. Overall, $LBSC^+$ outperforms the WLFU algorithm by 15% on average in FPDB. We also observe that the performance benefit of $LBSC^+$ decreases as θ increases. When θ is small, there is little access skewness, and the cost difference between different segments is more pronounced due to the high miss rate, leading to the higher effectiveness of $LBSC^+$. However, when θ is large, the access skewness overwhelms the difference in cost among segments. Furthermore, Figure 7(b) shows that $LBSC^+$ performs best in the changing workload, which demonstrates $LBSC^+$ can adapt well to the changing workload. For example, $LBSC^+$ outperforms WLFU by 20% when the cache size is 20GB.



(a) Comparison with different zipfian factors in 2GB cache. (b) Comparison with different cache using the changing workload.

Fig. 7. The total cost comparison among $LBSC^{+}$, WLFU, and LFU in FPDB.

TABLE III
PERFORMANCE COMPARISON BETWEEN BELADYSIZECOST AND BELADY IN FPDB

Execution Time(min)	Zipfian skew factor				Cache Size			
	0.0	0.6	1.2	2.0	10G	15G	20G	25G
Belady	141	116	68	29	1426	1037	580	232
BeladySizeCost	112	89	53	20	1188	1027	477	148

Moreover, the execution time of all algorithms decreases as the cache size becomes larger because of more cached data.

Finally, we evaluate the performance of BeladySizeCost in FPDB. Table III compares their performance across four different queries and a changing workload in various caches. The dataset settings are the same as those in Figure 7. We observe that BeladySizeCost outperforms Belady in all cases, demonstrating the effectiveness of our proposed cost function.

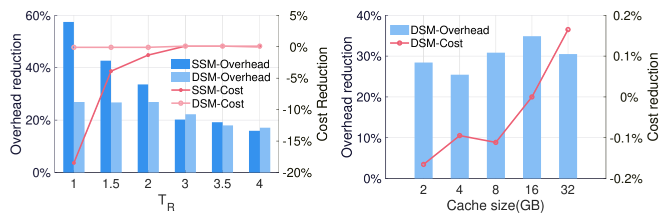
3) Optimizations on $LBSC^{+}$:
In this subsection, we evaluate the effectiveness of the three optimizations proposed in Section IV, i.e., sampling with cost-size threshold, model reusing, and outlier detection (also with the bypass strategy used in cache insertion). Note that the workload used in this subsection is the same as the Figure 5(a).

Inference Reusing.

Sampling with Cost-size Threshold. In Section V-B, we propose the two sampling methods: SSM and DSM. Figure 8 compares the two methods in a specific workload.

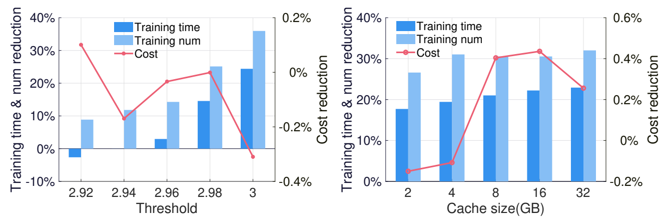
We have three key observations. *First*. We can achieve a positive outcome in computation overhead reduction while maintaining the same cost reduction if we know the $\mathcal{R}$'s distribution of the entire workload. In Figure 8(a), when we set $T_{\mathcal{R}}$ to 3, the computation overhead reduction is up to 20%, and the cost reduction is the same as the $baseline$ meanwhile. We can find that $T_{\mathcal{R}}$ covers more than 99% of the evictions in BeladySizeCostif it is greater than 3. This indicates the vast majority of items that should be evicted from the cache are in the set $\mathcal{R}^{l}$, so sampling data only from $\mathcal{R}^{l}$ indeed has the same $\mathcal{C}$ as the $baseline$, while the number of model inferences is much less than the $baseline$.

Second. As $T_{\mathcal{R}}$ becomes larger, the computation overhead reduction decreases, and the cost reduction becomes better. This is because the larger $T_{\mathcal{R}}$, the more items $\mathcal{R}^{l}$ contains. Consequently, the number of model inferences increases, which is the bottleneck affecting the computation overhead. At the same time, the items in $\mathcal{R}^{h}$ prefer to retain in the cache, so the cost is closer to the $baseline$. Figure 8(a) demonstrates that the cost reduction is up to -18% (the red lines in the figure) when $T_{\mathcal{R}}$ is set to 1, but the corresponding computation



(a) Comparison between the SSM and DSM methods. (b) Comparison of DSM with varying cache sizes.

Fig. 8. In (a), the histogram presents the computation overheads reduction to the $baseline$ w.r.t. different $T_{\mathcal{R}}$ values. The line chart shows the cost reduction to the $baseline$ using SSM and DSM. In (b), we compare the DSM with varying cache sizes of a specific workload and the initial $T_{\mathcal{R}}$ is set to 0.5.



(a) Impact of threshold. (b) Impact of cache size.

Fig. 9. In (a), the histogram presents the training time and the training number reduction to the $baseline$ among different thresholds, and the line chart presents the corresponding cost reduction to the $baseline$. In (b), we compare the performance of model reusing at varying cache sizes, and the default similarity threshold is set as 2.99.

overhead reduction is more than 55% (the blue bars in the figure). When $T_{\mathcal{R}}$ is set to 4, the cost is the same as the $baseline$, but the computation overhead reduction is only 17%.

Third. The DSM method is more robust. In real-world systems, it is impossible to obtain the true $\mathcal{R}$'s distribution of the entire workload. Therefore, if we do not set the $T_{\mathcal{R}}$ properly, the cost performance will be severely degraded (e.g., when the threshold is set as 1, as shown in Figure 8(a)). However, our DSM method is robust for arbitrary initial $T_{\mathcal{R}}$ configurations and has a significant improvement in throughput. As shown in Figure 8(b), the computation overhead reduction is around 30% on average. Meanwhile, the loss of the cost is no more than 0.7%. However, if we use SSM and set the $T_{\mathcal{R}}$ as 0.5, the $\mathcal{R}^{l}$ has no data in some cases, which may shut down the system.

Model Reusing. To reduce the computation overhead of model training, we store the previously trained models in the memory or local storage.

Firstly, we investigate how the threshold impacts the performance (i.e., $\mathcal{C}$ and computation overhead). As shown in Figure 9(a), the training time and the training number reduction both decrease when the threshold becomes small, but the corresponding cost reduction is increased. This is because the smaller the threshold, the fewer reusable models. Conversely, the reused model may not work well when the threshold is significantly big. To determine an appropriate threshold, we use the prefix of the trace as described in Section **??**.

Additionally, we find the training number reduction is always bigger than the training time reduction. The reason is that computing JS divergence also produces extra overheads. When the threshold is set to 2.92, the training time reduction is even less than 0, indicating the overheads of calculating JS divergence cannot be negligible.

Figure 9(b) depicts the performance of model reusing w.r.t. varying cache sizes. We find that the training time reduction is around 20%, the training number reduction is around 30%, and the corresponding cost reduction is even by up to 0.4% when the cache size is 16GB. This suggests that reusing the previous model is sometimes even better than continuously retraining the model. In summary, these results demonstrate the superior performance of our method.

VIII. Related Work

In this section, we review the relevant research works in three broad lines: cloud databases, caching algorithms and learning-based systems.

Cloud Databases. A cloud database is built to run in a public or hybrid cloud environment to help organize, store, and manage data within an organization [46], [47]. Modern cloud databases adopt an architecture with storage disaggregation. Examples include traditional data warehousing systems adapting to the cloud as well as databases natively developed for the cloud, such as Snowflake [2], Alluxio [48], PolarDB [1], AWS Aurora [49], and FPDB [8] that is used in this work.

Caching Algorithms. The design of the cache has been studied for more than 50 years, we only focus on software cache designs here, especially with variable-sized cached data. They can be divided into two major lines of work: heuristic-based and learning-based. GDSF is the heuristic-based algorithm that is the most relevant to our work. Furthermore, there are other recent researches about learning-based algorithms such as Hawkeye [50], RL-Belady [15] and PARROT [13]. While they can adapt to changing workloads, most are impractical because of high computation overheads. They all imitate Belady or some other approximate bounds that measure performance using miss rate. However, none of them takes the cost into consideration; therefore, they are unsuitable for cloud databases.

Learning Systems. Many system optimizations can be done by machine learning and deep learning models [51]–[55]. For instance, in the area of databases, such models can be applied in index optimization [56]–[60], cardinality estimation [61], [62], query optimizer [63]–[65], configuration tuning [66], [67] and concurrency control [68]. Besides databases, ML-based works have been done in other areas of the system, such as sorting algorithms [69], bloom filter [70], and CPU scheduling [71]. Especially, Polyjuice [68] proposes to use reinforcement learning to solve the concurrency control in database systems, which performs better than the state-of-the-art solutions. Although these works try to leverage machine learning to make systems self-driving, none of them targets cost-aware caching algorithms which have different models.

IX. Conclusion

In this paper, we propose a machine learning-based cost-aware caching framework primarily aimed at improving the performance of cloud databases. Firstly, we present an approximately optimal oracle algorithm called BeladySizeCost for $LBSC^+$ to learn. We prove that it is optimal, assuming the cache is full when every new request comes. Next, to make the learning-based algorithm more lightweight, we introduce several optimizations. Finally, we implement a simulator and integrate the framework into FPDB to evaluate the performance.

Acknowledgment

TODO: This work is supported by National Natural Science Foundation of China (62072033). Yuncheng Wu's work is supported by the National Research Foundation, Singapore under its Emerging Areas Research Projects (EARP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

References

[1] W. Cao, Y. Liu, Z. Cheng, N. Zheng, W. Li, W. Wu, L. Ouyang, P. Wang, Y. Wang, R. Kuan *et al.*, "Polardb meets computational storage: Efficiently support analytical workloads in cloud-native relational database." in *18th USENIX Conference on File and Storage Technologies, FAST 20*, 2020, pp. 29–41.

[2] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang *et al.*, "The snowflake elastic data warehouse," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 215–226.

[3] X. Yu, M. Youill, M. Woicik, A. Ghanem, M. Serafini, A. Aboulnaga, and M. Stonebraker, "Pushdowndb: Accelerating a dbms using s3 computation," in *2020 IEEE 36th International Conference on Data Engineering*, 2020, pp. 1802–1805.

[4] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi, "Logbase: A scalable log-structured database system in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, 2012.

[5] J. Wang, T. Li, A. Wang, X. Liu, L. Chen, J. Chen, J. Liu, J. Wu, F. Li, and Y. Gao, "Real-time workload pattern analysis for large-scale cloud databases," *Proceedings of the VLDB Endowment*, vol. 16, no. 12, pp. 3689–3701, 2023.

[6] L. V. Rodriguez, F. Yusuf, S. Lyons, E. Paz, R. Rangaswami, J. Liu, M. Zhao, and G. Narasimhan, "Learning cache replacement with {CACHEUS}," in *19th USENIX Conference on File and Storage Technologies, FAST 21*, 2021, pp. 341–354.

[7] E. J. O'neil, P. E. O'neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," *ACM SIGMOD Record*, vol. 22, no. 2, pp. 297–306, 1993.

[8] Y. Yang, M. Youill, M. Woicik, Y. Liu, X. Yu, M. Serafini, A. Aboulnaga, and M. Stonebraker, "Flexpushdowndb: Hybrid pushdown and caching in a cloud dbms," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, 2021.

[9] D. Durner, B. Chandramouli, and Y. Li, "Crystal: a unified cache storage system for analytical databases," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2432–2444, 2021.

[10] "Amazon s3 select," 2018, https://aws.amazon.com/blogs/aws/s3-glacierselect/.

[11] M. Ma, Z. Yin, S. Zhang, S. Wang, C. Zheng, X. Jiang, H. Hu, C. Luo, Y. Li, N. Qiu *et al.*, "Diagnosing root causes of intermittent slow queries in cloud databases," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1176–1189, 2020.

[12] Z. Ji, Z. Xie, Y. Wu, and M. Zhang, "Lbsc: A cost-aware caching framework for cloud databases," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2024, pp. 4911–4924.

[13] E. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," in *International Conference on Machine Learning*. PMLR, 2020, pp. 6237–6247.

[14] Z. Song, D. S. Berger, K. Li, A. Shaikh, W. Lloyd, S. Ghorbani, C. Kim, A. Akella, A. Krishnamurthy, E. Witchel *et al.*, "Learning relaxed belady for content distribution network caching," in *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 20*, 2020, pp. 529–544.

[15] G. Yan and J. Li, "Rl-bélády: A unified learning framework for content caching," in *Proceedings of the 28th ACM International Conference on Multimedia*, 2020, pp. 1009–1017.

[16] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[17] G. Yan, J. Li, and D. Towsley, "Learning from optimal caching for content delivery," in *Proceedings of the 17th International Conference on Emerging Networking Experiments and Technologies*, 2021, pp. 344–358.

[18] D. S. Berger, N. Beckmann, and M. Harchol-Balter, "Practical bounds on optimal caching with variable object sizes," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 2, pp. 1–38, 2018.

[19] A. MySQL, "Mysql," 2001, https://www.mysql.com/.

[20] B. Momjian, *PostgreSQL: introduction and concepts*. Addison-Wesley New York, 2001, vol. 192.

[21] N. Beckmann, H. Chen, and A. Cidon, "{LHD}: Improving cache hit rate by maximizing hit density," in *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 18*, 2018, pp. 389–403.

[22] J. Yang, Z. Mao, Y. Yue, and K. Rashmi, "{GL-Cache}: Group-level learning for efficient and high-performance caching," in *21st USENIX Conference on File and Storage Technologies, FAST 23*, 2023, pp. 115–134.

[23] W. Zhou, Z. Niu, Y. Xiong, J. Fang, and Q. Wang, "{3L-Cache}: Low overhead and precise learning-based eviction policy for caches," in *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, 2025, pp. 237–254.

[24] P. Domingos, "A few useful things to know about machine learning," *Communications of the ACM*, vol. 55, no. 10, pp. 78–87, 2012.

[25] S. Cai, K. Zheng, G. Chen, H. Jagadish, B. C. Ooi, and M. Zhang, "Arm-net: Adaptive relation modeling network for structured data," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 207–220.

[26] R. Fu, Y. Wu, Q. Xu, and M. Zhang, "Feast: A communication-efficient federated feature selection framework for relational data," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–28, 2023.

[27] J. Zhang, Z. Luo, Q. Xu, and M. Zhang, "Pa-feat: Fast feature selection for structured data via progress-aware multi-task deep reinforcement learning," in *2023 IEEE 39th International Conference on Data Engineering*, 2023, pp. 394–407.

[28] T. Zhang, J. Tan, X. Cai, J. Wang, F. Li, and J. Sun, "Sa-lsm: optimize data layout for lsm-tree based storage using survival analysis," *Proceedings of the VLDB Endowment*, vol. 15, no. 10, pp. 2161–2174, 2022.

[29] A. Natekin and A. Knoll, "Gradient boosting machines, a tutorial," *Frontiers in neurorobotics*, vol. 7, p. 21, 2013.

[30] D. S. Berger, "Towards lightweight and robust machine learning for cdn caching," in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, 2018, pp. 134–140.

[31] P. Black, "Red black tree," 2006.

[32] P. Cao and S. Irani, "Cost-aware www proxy caching algorithms." in *Usenix Symposium on Internet Technologies and Systems*, vol. 12, no. 97, 1997, pp. 193–206.

[33] C. Li and A. L. Cox, "Gd-wheel: a cost-aware replacement policy for key-value stores," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, pp. 1–15.

[34] M. Menéndez, J. Pardo, L. Pardo, and M. Pardo, "The jensen-shannon divergence," *Journal of the Franklin Institute*, vol. 334, no. 2, pp. 307–318, 1997.

[35] "Minio," 2016, https://min.io/.

[36] Z. Yang, C. Yang, F. Han, M. Zhuang, B. Yang, Z. Yang, X. Cheng, Y. Zhao, W. Shi, H. Xi *et al.*, "Oceanbase: a 707 million tpmc distributed relational database system," *Proceedings of the VLDB Endowment*, vol. 15, no. 12, pp. 3385–3397, 2022.

[37] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang *et al.*, "Tidb: a raft-based htap database," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3072–3084, 2020.

[38] W. Cao, Y. Zhang, X. Yang, F. Li, S. Wang, Q. Hu, X. Cheng, Z. Chen, Z. Liu, J. Fang *et al.*, "Polardb serverless: A cloud native database for disaggregated data centers," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2477–2489.

[39] F. Li, "Cloud-native database systems at alibaba: Opportunities and challenges," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2263–2272, 2019.

[40] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak, "The star schema benchmark and augmented fact table indexing," in *Performance Evaluation and Benchmarking: First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers 1*. Springer, 2009, pp. 237–252.

[41] "Tpc-h," 1988, https://www.tpc.org/tpch/.

[42] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," in *Proceedings of the 1994 International Conference on Management of Data*, 1994, pp. 243–252.

[43] E. G. Coffman and P. J. Denning, *Operating systems theory*. prentice-Hall Englewood Cliffs, NJ, 1973, vol. 973.

[44] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin, "Evaluating content management techniques for web proxy caches," *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, no. 4, pp. 3–11, 2000.

[45] N. Littlestone and M. K. Warmuth, "The weighted majority algorithm," *Information and Computation*, vol. 108, no. 2, pp. 212–261, 1994.

[46] Y. Wu, T. T. A. Dinh, G. Hu, M. Zhang, Y. M. Chee, and B. C. Ooi, "Serverless data science-are we there yet? a case study of model serving," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1866–1875.

[47] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K.-L. Tan, Y. M. Teo, and S. Wang, "Efficient distributed memory management with rdma and caching," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1604–1617, 2018.

[48] "Alluxio," 2021, https://www.alluxio.io/.

[49] A. Verbitski, A. Gupta, D. Saha, J. Corey, K. Gupta, M. Brahmadesam, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvilli *et al.*, "Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 789–796.

[50] A. Jain and C. Lin, "Back to the future: Leveraging belady's algorithm for improved cache replacement," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture*. IEEE, 2016, pp. 78–89.

[51] W. Wang, M. Zhang, G. Chen, H. Jagadish, B. C. Ooi, and K.-L. Tan, "Database meets deep learning: Challenges and opportunities," *ACM SIGMOD Record*, vol. 45, no. 2, pp. 17–22, 2016.

[52] W. Wang, J. Gao, M. Zhang, S. Wang, G. Chen, T. K. Ng, B. C. Ooi, J. Shao, and M. Reyad, "Rafiki: Machine learning as an analytics service system," *Proceedings of the VLDB Endowment*, vol. 12, no. 2.

[53] W. Wang, G. Chen, A. T. T. Dinh, J. Gao, B. C. Ooi, K.-L. Tan, and S. Wang, "Singa: Putting deep learning in the hands of multimedia users," in *Proceedings of the 23rd ACM international conference on Multimedia*, 2015, pp. 25–34.

[54] B. C. Ooi, K.-L. Tan, S. Wang, W. Wang, Q. Cai, G. Chen, J. Gao, Z. Luo, A. K. Tung, Y. Wang *et al.*, "Singa: A distributed deep learning platform," in *Proceedings of the 23rd ACM international conference on Multimedia*, 2015, pp. 685–688.

[55] N. Xing, S. H. Yeung, C.-H. Cai, T. K. Ng, W. Wang, K. Yang, N. Yang, M. Zhang, G. Chen, and B. C. Ooi, "Singa-easy: An easy-to-use framework for multimodal analysis," in *Proceedings of the 29th ACM International Conference on Multimedia*, 2021, pp. 1293–1302.

[56] Y. Dai, Y. Xu, A. Ganesan, R. Alagappan, B. Kroth, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "From {WiscKey} to bourbon: A learned index for {Log-Structured} merge trees," in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 20*, 2020, pp. 155–171.

[57] 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*, 2018, pp. 489–504.

[58] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann *et al.*, "Alex: an updatable adaptive learned index," in *Proceedings of the 2020 International Conference on Management of Data*, 2020, pp. 969–984.

[59] C. Yue, Z. Xie, M. Zhang, G. Chen, B. C. Ooi, S. Wang, and X. Xiao, "Analysis of indexing structures for immutable data," in *Proceedings of the 2020 International Conference on Management of Data*, 2020, pp. 925–935.

[60] S. Wu, Y. Li, H. Zhu, J. Zhao, and G. Chen, "Dynamic index construction with deep reinforcement learning," *Data Science and Engineering*, vol. 7, no. 2, pp. 87–101, 2022.

[61] K. Kim, J. Jung, I. Seo, W.-S. Han, K. Choi, and J. Chong, "Learned cardinality estimation: An in-depth study," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1214–1227.

[62] J. Sun, J. Zhang, Z. Sun, G. Li, and N. Tang, "Learned cardinality estimation: A design space exploration and a comparative evaluation," *Proceedings of the VLDB Endowment*, vol. 15, no. 1, pp. 85–97, 2021.

[63] 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*, vol. 12, no. 11, pp. 1705–1718, 2019.

[64] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making learned query optimization practical," *ACM SIGMOD Record*, vol. 51, no. 1, pp. 6–13, 2022.

[65] Q. Cai, C. Cui, Y. Xiong, W. Wang, Z. Xie, and M. Zhang, "A survey on deep reinforcement learning for data processing and analytics," *IEEE Transactions on Knowledge and Data Engineering*, 2022.

[66] C. Lin, J. Zhuang, J. Feng, H. Li, X. Zhou, and G. Li, "Adaptive code learning for spark configuration tuning," in *2022 IEEE 38th International Conference on Data Engineering*, 2022, pp. 1995–2007.

[67] D. Van Aken, D. Yang, S. Brillard, A. Fiorino, B. Zhang, C. Bilien, and A. Pavlo, "An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems," *Proceedings of the VLDB Endowment*, vol. 14, no. 7, pp. 1241–1253, 2021.

[68] J.-C. Wang, D. Ding, H. Wang, C. Christensen, Z. Wang, H. Chen, and J. Li, "Polyjuice: High-performance transactions via learned concurrency control." in *15th USENIX Symposium on Operating Systems Design and Implementation*, 2021, pp. 198–216.

[69] A. Kristo, K. Vaidya, U. Çetintemel, S. Misra, and T. Kraska, "The case for a learned sorting algorithm," in *Proceedings of the 2020 International Conference on Management of Data*, 2020, pp. 1001–1016.

[70] H. Chen, Z. Wang, Y. Li, R. Yang, Y. Zhao, R. Zhou, and K. Zheng, "Deep learning-based bloom filter for efficient multi-key membership testing," *Data Science and Engineering*, vol. 8, no. 3, pp. 234–246, 2023.

[71] Y. Sheng, A. Tomasic, T. Zhang, and A. Pavlo, "Scheduling oltp transactions via machine learning," *arXiv preprint arXiv:1903.02990*, 2019.