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

Zhongle Xie*, Zhaoxuan Ji*, Kehan Liu *Member, IEEE,* Quanqing Xu, Meihui Zhang *Senior Member, IEEE,*

***Abstract*—Cloud databases rely on caching to mitigate high latency and limited bandwidth introduced by storage-disaggregation architectures. However, existing caching mechanisms struggle to handle dynamic workloads, ignore data-fetching costs, and optimize only eviction, leading to inefficient cache utilization. Learning-based approaches offer adaptability but incur high inference overhead and typically assume unconditional item admission, resulting in unnecessary evictions and wasted computation. In this paper, we propose $LBSC^+$, a learning-based cost-aware caching framework that jointly optimizes admission and eviction for cloud databases. First, we introduce BeladySizeCost, an approximately optimal cost-aware oracle for joint admission and eviction, which prioritizes data items with high cost per byte that are likely to be accessed in the near future. Using this oracle, we then train a lightweight supervised model that predicts item utility for both cache admission and eviction. To reduce inference overhead during online execution, we further incorporate three performance optimizations for $LBSC^+$: adaptive inference reuse, cost-aware sampling, and model reuse. Experiments using synthetic traces and real-world cloud database deployments show that $LBSC^+$ reduces total data-fetching cost by up to 17% relative to state-of-the-art methods. Additionally, the proposed performance optimization techniques reduce computation overhead by up to 90%, making $LBSC^+$ practical for large-scale cloud systems.**

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

I. Introduction

CLOUD databases [1]–[3] increasingly adopt storage-disaggregation architectures, where compute and storage nodes communicate over the network. While this architecture provides elasticity and operational flexibility, it also introduces high data-access latency and limited bandwidth between compute and storage. To mitigate these bottlenecks, cloud systems often cache data at compute nodes. However, traditional caching techniques are ill-suited for cloud environments because they fail to satisfy three essential requirements:

The first requirement is **Adaptability to dynamic workloads**. Cloud databases experience highly dynamic workloads compared to conventional databases [4]. Real-world traces often exhibit mixed patterns, such as random lookups, large scans, and skewed access bursts that shift over time [5]. Heuristic policies like Least Recently Used (LRU) [6] perform well only on specific access patterns and degrade significantly when the workload changes [5]. Consequently, caching strategies relying on those static heuristics cannot consistently adapt to evolving request behaviors.

The second is **Awareness of data-fetching cost**. The data fetching cost in cloud databases comprises both the transfer costs, dominated by bandwidth limitations [7], [8], and computation costs, introduced by predicate push-down or object processing in storage nodes [9], [10]. These costs may vary by data items and queries. Without explicit cost awareness, cache management decisions may retain low-utility items and evict expensive ones, causing repeated high-cost data fetching, resulting in suboptimal performance.

The last requirement is **Coordinated admission and eviction**. Caches for cloud databases must jointly determine *what to admit* and *what to evict*. Existing approaches, including LBSC [11], focus primarily on eviction and implicitly admit every incoming item. This strategy may lead to cache pollution: a burst of low-utility or one-time requests evicts valuable items unnecessarily. Since admission and eviction decisions directly affect one another, cloud systems require a unified mechanism that simultaneously optimizes both.

Recently, numerous learning-based algorithms have been developed to enhance the adaptability of caching in traditional databases [5], [12], [13], typically by mimicking the Belady [14] oracle algorithm. Yet, most of these methods still lack cost-awareness and consider only evictions. In addition, learning-based systems introduce non-negligible inference and retraining overhead, which may even outweigh their benefits in latency-sensitive settings. These observations motivate the need for a cost-aware, learning-based caching framework that jointly optimizes admission and eviction while remaining efficient for production cloud databases. Designing such a framework, however, faces two challenges:

Challenge 1: The absence of an oracle algorithm for joint admission and eviction. As in prior learning-based caching algorithms [13], [15], [16], incorporating an oracle algorithm is essential for cloud-native databases. However, deriving an optimal oracle becomes challenging when considering data fetching costs with joint admission and eviction. In fact, designing the optimal caching policy in such a case is known to be an NP-hard problem [17].

Challenge 2: The efficiency in utilizing learning models. Although existing learning-based algorithms adapt well to changing workloads, they typically require continuous retraining and extensive model inferences [13], [16], which in-

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

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

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

Quanqing Xu is with OceanBase and the Research Institute of Ant Group, Hangzhou 310000, China (e-mail: xuquanqing.xqq@oceanbase.com)

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

Corresponding author: Meihui Zhang.

* The authors contributed equally

troduces substantial computational overhead. Without careful optimization, such overheads may outweigh the benefits of improved cache decisions, limiting applicability in latency-sensitive systems.

To address these challenges, we propose $LBSC^+$, a unified learning-based cost-aware caching framework for cloud databases. $LBSC^+$ jointly predicts admission and eviction decisions using a lightweight supervised model trained on a new oracle algorithm we introduce, called BeladySizeCost, which approximates the optimal joint policy while incorporating data-fetching cost. The oracle prioritizes items with high cost per byte and high likelihood of near-future access, enabling principled cost-aware caching. To make $LBSC^+$ practical in real systems, we further introduce three complementary performance optimizations, i.e., adaptive inference reuse, cost-aware sampling, and model reuse, which together reduce online computation overhead by up to 90%. Through extensive experiments on synthetic traces and real cloud databases, we show that $LBSC^+$ reduces total data-fetching cost by up to 17% and delivers robust performance under diverse workloads.

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

- We present $LBSC^+$, a unified learning-based framework, which jointly optimizes admission and eviction for cost-aware caching against dynamic workloads in cloud databases.
- 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.
- We design a lightweight shared model for both admission and eviction predictions and introduce inference reuse, sampling, and model reuse to reduce computation overhead without sacrificing accuracy of the model.
- We implement an $LBSC^+$ simulator and integrate $LBSC^+$ into underlying cloud databases. Experiments on real cloud workloads demonstrate that $LBSC^+$ improves performance by up to 17% compared with state-of-the-art algorithms, and reduces computation overhead by up to 90%.

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

II. BeladySizeCost: Cost-aware oracle algorithm in caching

In this section, we first motivate the need for a cost-aware metric in cloud database caching and then introduce BeladySizeCost, an approximately optimal oracle algorithm designed to minimize total data-fetching cost. We formally describe its ranking function, theoretical basis, and practical estimation strategy.

TABLE I
Performance metrics used in caching 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	$\sum_{j\in Miss} \mathcal{C}_j$

A. Cost-aware Metric in Caching

In cloud databases, the data-fetching costs consist of transfer cost, dominated by bandwidth constraints and large data volume, and computation cost, inccurred by operations on storage nodes. Traditional caching systems evaluate performance using metrics such as miss rate or byte miss rate as summarized in Table I. The former, used in databases like MySQL [18] and PostgreSQL [19], counts the fraction of cache misses but neglects the data-fetching costs. The latter improves upon this by accounting for cache item sizes, thereby reflecting transfer costs. However, it still assumes a uniform retrieval cost across cached items and overlooks computation costs. The assumptions do not always hold for cutting-edge cloud databases, especially those built on storage-disaggregated architectures. To more accurately represent data-fetching cost in such environments, we define the *total cost* of a cached item i, denoted as $\mathcal{C}_i$, as follows:

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

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 nodes, and it is proportional to the data size.

B. BeladySizeCost Algorithm

The Belady algorithm [14] is widely recognized as the optimal eviction policy in traditional caching systems and has served as the foundation for many recent learning-based caching approaches [13], [20]. Belady evicts the item whose next access lies farthest in the future and achieves optimality under the assumptions of uniform object size, uniform data-fetching cost, and unconditional admission.

However, Belady becomes inadequate for cloud databases because these assumptions no longer hold. In storage-disaggregated architectures, cache items vary in size and incur heterogeneous *total cost* defined in Eq. 1. Moreover, unconditional admission may introduce cache pollution as mentioned in Section I. Motivated by this, we summarize the desired properties of a practical oracle for cloud databases as follows:

P1. **Cost-awareness.** The algorithm should incorporate the *total cost* with admission control.

P2. **Online applicability.** The algorithm should be computationally efficient and capable of online tracking in cloud settings.

P3. **Belady reduction.** The algorithm should revert to Belady-style eviction behavior when all items have the identical *total cost* and data size.

To meet these properties, we propose BeladySizeCost, an approximately optimal cost-aware oracle algorithm for cloud databases. The key idea is to quantify the benefit of caching each item based on its cost, its probability of near-future access, and its size. For the cached item i, BeladySizeCost computes a utility score:

$$U_i = \frac{\mathcal{C}_i \cdot p_i}{s_i}$$

where $\mathcal{C}_i$ denotes the *total cost* defined in Eq. 1, p_i represents the probability that data item i will be accessed in the future (hit probability), and s_i is the data size.

Upon each request arrival, BeladySizeCost performs joint admission and eviction through a unified ranking process. Specifically, it forms a candidate set consisting of all cached items and the newly requested item, computes their utility scores, and removes the lowest-utility items until the cache capacity constraint is satisfied.

Before explaining how BeladySizeCost satisfies the three desired properties. We first conduct an asymptotic analysis of BeladySizeCost to establish its theoretical optimality and introduce how to estimate hit probability in practice.

Theoretical Optimality. We consider a dynamic programming formulation that minimizes the *total cost*. Let $\mathcal{T}_{t+1}(S_{t+1})$ represent the set of possible cache states at time $t+1$, given the current cache state S_t and a new data request α_{t+1}. The set of feasible cache states upon α_{t+1} can be written as:

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

where S denotes any cache state in $\mathcal{T}_{t+1}(S_{t+1})$. If $S = S_t$, it indicates that α_{t+1} is not admitted into the cache; otherwise, α_{t+1} replaces one or more items in the cache. In cloud database, individual cached items are typically much smaller than the overall cache capacity. Therefore, we have the following lemma:

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

Theorem 1. *The BeladySizeCost algorithm finds the optimal eviction solution.*

Proof. We first define a dynamic programming formulation $DP_t(S_t)$ in Eq 3, which denotes the possible minimum total cost $\mathcal{C}$ under the cache state S_t.

$$DP_t(S_t) = \mathcal{C}_t \cdot \mathbb{I}(\alpha_{t+1} \in S_t) + \min_{S_{t+1} \in \mathcal{T}_t(S_t)} DP_{t+1}(S_{t+1}) \tag{3}$$

In the equation, $\mathcal{C}_t$ indicates the data cost of the item at request t. $\mathbb{I}()$ represents the indicator function, i.e., $\mathbb{I}(true) = 1$ while $\mathbb{I}(false) = 0$. Particularly, the final cache state $DP_T(S_T) = 0$ at the last index T. For S_t, the cache state $S_{t+1} \in \mathcal{T}_t(S_t)$ is an optimal choice for the next state if and only if it preserves the dynamic programming recursion. Thus, the dynamic programming formulation is equivalent to minimizing the total cost $\mathcal{C}$ of the entire workload. Consequently, we can transform the minimization of $\mathcal{C}$ into an optimization problem formalized by Eq 4. For ease of representation, we denote the total cache size as CS in the equation. It is worth noting that the problem is equivalent to the knapsack problem, which is known as NP-complete [21].

$$\begin{aligned} &\max_{i \in \mathcal{T}_t(S_t)} \mathcal{C}_i \cdot p_i \\ s.t. &\sum_{i \in \mathcal{T}_t(S_t)} s_i \ \leq \ CS \end{aligned} \tag{4}$$

Specifically, we now import Lemma 1 for cloud databases and we could find the optimal solution. We proceed to demonstrate that this solution is optimal. Let $\mathcal{T}^{'} \neq \mathcal{T}$ be an arbitrary subset of $\mathcal{T}_t(S_t)$ satisfying Eq 4. To establish the optimality of the solution, we need to show that $\sum_{i \in \mathcal{T}'} \mathcal{C}_i \cdot p_i \ \leq \ \sum_{j \in \mathcal{T}} \mathcal{C}_j \cdot p_j$. Since the BeladySizeCost holds items in the cache with maximal $rank_i = \frac{\mathcal{C}_i \cdot p_i}{s_i}$, it follows Eq 5, indicating that the total rank of the cache state, as determined by BeladySizeCost decisions, is higher than or equal to any decisions because the BeladySizeCost retains maximal $rank_i$ in the cache. Furthermore, as BeladySizeCost always holds items with maximal $rank_i$, the total rank of the cache states maintained by BeladySizeCost is higher than or equal to the total rank of the cache state maintained by other algorithms. That is, we have the following inequality:

$$\sum_{i \in \mathcal{T}'} \frac{\mathcal{C}_i \cdot p_i}{s_i} \ \leq \ \sum_{j \in \mathcal{T}} \frac{\mathcal{C}_j \cdot p_j}{s_j}. \tag{5}$$

We define:

$$\begin{aligned} rank^{\mathcal{T}}_{min} &= \min_{i \in \mathcal{T}} \frac{\mathcal{C}_i \cdot p_i}{s_i} \\ rank^{\mathcal{T}'}_{max} &= \max_{i \in \mathcal{T}'} \frac{\mathcal{C}_i \cdot p_i}{s_i}. \end{aligned} \tag{6}$$

From the definition, we can deduce that $rank^{\mathcal{T}}_{min} \geq rank^{\mathcal{T}'}_{max}$ since BeladySizeCost always evicts the lowest $rank_i$. Consequently, Eq 5 holds as follows:

$$\begin{aligned} \sum\nolimits_{i \in \mathcal{T}} \mathcal{C}_i \cdot p_i &\geq rank^{\mathcal{T}}_{min} \cdot CS \\ &\geq rank^{\mathcal{T}'}_{max} \cdot CS \\ &\geq \ \sum\nolimits_{i \in \mathcal{T}'} \mathcal{C}_i \cdot p_i \end{aligned} \tag{7}$$

Therefore, we show that the solution of the BeladySizeCost indeed maximizes Eq 4, which means it is optimal. □

Hit Probability Estimation. To obtain the utility score U_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 item i are known and fixed, p_i is the only unknown factor in U_i, which can be inferred from the data's locality, i.e., the likelihood of being re-accessed within a short period [20]. Thus, we approximate p_i by the next-accessed probability of future requests:

$$p_i \propto \sum_{x=1}^{\infty} \frac{1}{x} \cdot \mathbb{I}(age_i^t = t + x) \tag{8}$$

where $\mathbb{I}(\cdot)$ 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 consuming fewer CPU resources. Consequently, we rank the cached data items (including the newly added request) based on U_i, where $p_i = \frac{1}{x} \cdot \mathbb{I}(true)$, and evict the item(s) with the lowest value to restore the cache within the capacity limit.

To summarize, we discuss how BeladySizeCost satisfies the proposed properties, i.e., P1-P3. For P1, it explicitly incorporates *total cost* in the utility score. For P2, it relies only on per-item statistics, computing with low overhead. For P3, if all items share identical cost and size, the factor $\frac{\mathcal{C}_i}{s_i}$ is constant across items. The utility score $U_i = \frac{\mathcal{C}_i}{s_i} \cdot \frac{\mathbb{I}(true)}{x}$ is therefore equivalent to ranking by $\frac{1}{x}$ for items with future access and 0 otherwise. Hence, items with larger next-access distance receive smaller utility and are evicted first, which matches the Belady algorithm.

III. Design of $LBSC^+$

We present the design of $LBSC^+$, including an overview of the framework, the workflow of the Cache Manager, and the learning-based Predictor responsible for generating admission and eviction decisions, in this section.

A. Overview

$LBSC^+$, as depicted in Figure 1, consists of three tightly integrated components: Cache Manager, Predictor, and Performance Optimizer. The framework receives a continuous stream of requests, and for each request, $LBSC^+$ determines whether the item should be admitted into the cache and, if necessary, which existing items should be evicted when the cache becomes full. The design is centered on a single learning-based model trained to approximate the optimal behavior of BeladySizeCost. The functionality of each component is as follows:

Cache Manager governs online decision-making and coordinates admission and eviction. When a new request arrives, this component determines whether it should be admitted into the cache and identifies which cached item(s) should be evicted once the cache exceeds its capacity based on the prediction results. The complete process will be detailed in Section III-B.

Predictor serves as the learning component of $LBSC^+$. It continuously generates training data based on access histories, and learns from BeladySizeCost using supervised signals. After training, it jointly predicts admission and eviction probabilities for items. The model aims to capture varying data-access patterns in cloud environments. We will detail the model training and inference in Section III-C.

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

Algorithm 1 $LBSC^+$

1: $D_t = list()$
2: **for** each incoming request α_t **do**
3: update_features(α_t)
4: d_t = generate_training_data(α_t)
5: $D_t.add(d_t)$
6: **if** α_t is in cache **then**
7: **return**
8: **else**
9: **if** not is_cache_full(α_t) **then**
10: put α_t into the cache
11: **return**
12: **end if**
13: put metadata of α_t into a pool G_{tmp}
14: **while** is_cache_full(α_t) **do**
15: R = inference_reuse()
16: S = adaptive_sample(C)
17: U = batch_model_inference(S, M_t)
18: e = joint_admission_eviction(G_{tmp}, R, U)
19: apply_eviction(e).
20: **end while**
21: **return**
22: **end if**
23: **end for**

Algorithm 2 Background Model Refresh in $LBSC^+$

1: Initialize the corresponding thresholds: T_{JS}, δ_D
2: **while** len(D_t)$\geq \delta_D$ **do**
3: B_t = sample_recent_batch(D_t)
4: **if** calculate_all_JS(B_t) $> T_{JS}$ **then**
5: M_t = retrain(D_t)
6: **else**
7: M_t = find_similar_model(D_t)
8: **end if**
9: **end while**

B. Cache Manager

This component coordinates the end-to-end workflow of $LBSC^+$, encompassing both admission and eviction decisions under dynamic cloud workloads. Prior caching algorithms typically focus exclusively on eviction, removing items after the cache becomes full while admitting every new request unconditionally [5], [13], [20]. This strategy underutilizes the cache space: if an incoming item has a low access probability in the future, caching it brings little benefit and may trigger unnecessary evictions. A unified treatment of admission and eviction is therefore essential to ensure that only high-utility items occupy limited cache space. Preserving this motivation, $LBSC^+$ integrates both decisions into a single workflow guided by a shared model with the same learning objective.

The detailed workflow of the cache manager is illustrated in Algorithm 1. When a new request α_t arrives, the $LBSC^+$ framework first updates features of α_t to assist training data generation (as described later in Section III-C), and the generated training data is put into D_t (lines 3-5). Then, $LBSC^+$ checks whether the requested data item already exists in the

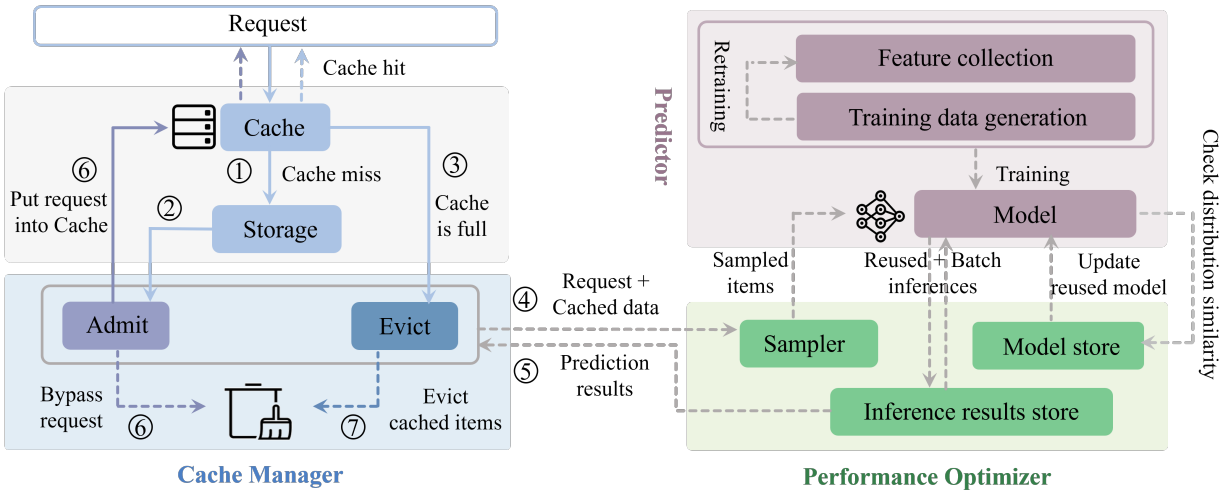


Fig. 1. An overview of the $LBSC^+$ framework.

cache. If a cache hit occurs, the cached item is then returned to the user (lines 6-7). If a cache miss occurs, the framework enters the joint admission-eviction stage.

In this stage, if the cache is not full, the new request α_t is directly inserted into the cache and returned (lines 9-12). Otherwise, the new request α_t is inserted into a temporary candidate pool G_{tmp} along with all cached items (line 13). This pool forms the set of items jointly considered for admission and eviction. Noting that the pool occupies little space, as it maintains only metadata, not the actual data content. $LBSC^+$ then iteratively restores the cache to its capacity limit. During each iteration (lines 14–20), the system applies the inference-reuse mechanism to avoid recomputing predictions for items whose prior inference results are still reliable. An adaptive sampling is then performed using a splitting strategy, which reduces the number of model inferences by focusing inference on items with cost-to-size ratios. Here, R denotes the reused item predictions, C the cached items, and S the sampled items. The details of the inference-reuse mechanism and the adaptive sampling are described in Sections IV-A and IV-B, respectively.

In line 17 of Algorithm 1, the cache manager calls the predictor to perform batch model inference on the sampled items and uses the resulting predicted utility scores to determine which items to evict. Items with the lowest predicted utility are removed until the cache state satisfies the capacity constraint. That is, when the new request receives the lowest score, it is bypassed entirely by the cache manager. Otherwise, one or more cached items are evicted, and the new request is inserted into the cache.

When the count of generated training data in D_t is larger than a predefined threshold δ_D, $LBSC^+$ would check whether the current workload distribution has shifted to invoke model retraining, as shown in Algorithm 2. The Jensen–Shannon (JS) divergence [22] between the most recent training batch and all previously stored batches is compared here, and the retraining of the model occurs only if every divergence exceeds the threshold T_{JS}. Otherwise, the most similar model is reused, thereby avoiding unnecessary training overhead (lines 4–8). The updated model is denoted as M_t in the algorithm, and this process is performed in the background to avoid blocking the processing of requests. The details on this process are elaborated in Section IV-C.

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

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

1) Feature Collection: Since features play a crucial role in machine learning [23], we carefully select the following features, inspired by the recent state of the arts [5], [13].

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

Delta. The number of accesses between two consecutive requests of the same item. Specifically, $Delta_1$ denotes the number of accesses since the most recent request, $Delta_2$ denotes the number of accesses between the two previous requests of that item, and so forth. Figure 2 illustrates an example, where the items colored green correspond to repeated accesses of the same item. In this case, $Delta_1 = I_3 - I_2$ and $Delta_2 = I_2 - I_1$, where I_1, I_2, I_3 denote the request indexes.

Domain Features. To customize the cache for different systems, domain knowledge is incorporated as auxiliary features to strengthen the learning model. For instance, in cloud databases, the cached items have varying sizes, and each item has its own metadata, including table name, column name, etc.

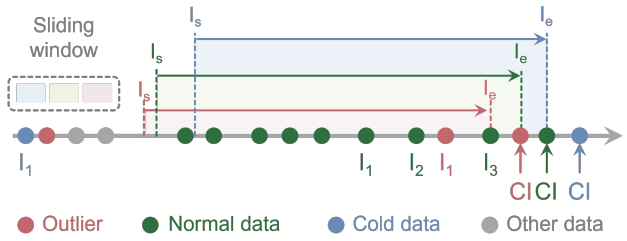


Fig. 2. Generation of three types of training data using a sliding window.

2) Model Training: We first introduce how to generate labels for the training data. Once a new request arrives, a sliding window[1] 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 ($2 \cdot L_e^s$ in our implementation) to indicate infrequent access. For instance, in Figure 2, the label of the data colored green is "$CI - I_3$" when a new request comes. Additionally, the last access of the item colored blue, i.e, I_1, exceeds the L_e^s, so we label it as "$2 \cdot L_e^s$". After labeling, the sliding window moves forward to generate the next training data. To further improve the model performance, we propose a simple yet efficient method to filter the outliers in the training data.

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, 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 $CI - I_1$, which suggests that it will be accessed after a short time. We formally define the outlier detection procedure 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.*

After generating sufficient training data, we proceed to train a prediction model. $LBSC^+$ adopts gradient boosting machines (GBM) [25], since tree models do not require feature normalization and support concurrent training, thereby reducing CPU resource consumption. In addition, GBM has demonstrated strong effectiveness in prior works [13], [26]. To adapt to evolving workloads, $LBSC^+$ retrains the model by default once enough new training data has been collected. However, continuous retraining may introduce unnecessary computation overhead. To address this issue, we propose a model reuse strategy, which is described in Section IV-C.

[1]The concept of sliding window has been proposed in previous works [13], [16], [24]. We refer interested readers to these works for the impact of sliding window size on model accuracy and efficiency.

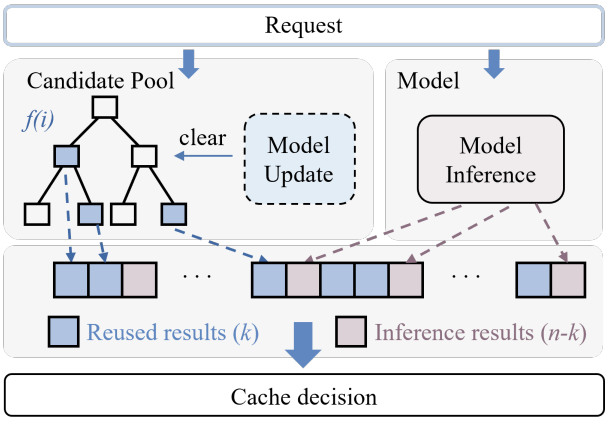


Fig. 3. The adaptive inference reusing process of $LBSC^+$.

3) Model Inference: As noted in Section III-B, when the cache reaches its capacity limit, $LBSC^+$ performs model inference to determine whether to admit the new item into the cache and which cached items to evict. Since the major computation overhead comes from the inferences of the newly requested data and cached data items, we employ several optimizations 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 inferences. We use a red-black tree [27] 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 [8], 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 the newly requested item, 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 IV.

IV. Performance Optimizations

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

A. Inference reusing

In learning-based caching systems, model inference is invoked frequently to support admission and eviction decisions, and often dominates the overall computation overhead.

However, across consecutive cache decisions, the majority of cached items remain unchanged, and their relative utility ordering does not immediately become stale. This temporal stability creates an opportunity to reuse previously computed inference results instead of recomputing them at every decision point. Nonetheless, directly reusing a fixed number of predictions per decision may cause limited robustness under dynamic workloads. Excessive reuse may apply stale predictions and degrade cache decisions, while conservative reuse forfeits potential efficiency gains. Therefore, an effective reuse mechanism must adaptively balance prediction freshness and inference reduction.

$LBSC^+$ introduces an adaptive inference reusing mechanism that selectively reuses prior predictions based on their estimated freshness. The key idea is to maintain a candidate pool $\mathcal{P}$ that stores predicted utility scores of cached items from previous decisions, and to determine reuse eligibility using a freshness metric defined as:

$$f(i) = \frac{score_i}{\log(1 + pred_i)} \tag{9}$$

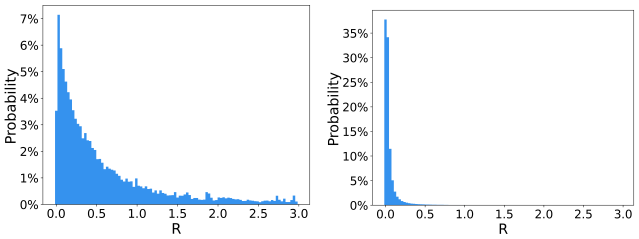
where $score_i$ is the model's predicted score for item i, and $pred_i$ denotes the elapsed time since item i's last model inference, indicating how stale its previous prediction has become. The logarithmic term normalizes temporal staleness, stabilizing the scale of the freshness metric under varying reuse intervals. A larger $f(i)$ indicates that the prediction remains reliable 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 in $\mathcal{P}$ are sorted in descending order of their freshness scores $f(i)$, and items with $f(i) > \theta$ are selected for reuse across consecutive decisions, where θ is a user-defined threshold. During this process, previously reused predictions are applied directly to admission and eviction decisions without invoking the model. Only the remaining items require fresh inference, thereby reducing the total inference count. Whenever the model is retrained, all cached predictions in $\mathcal{P}$ are invalidated to ensure consistency. In addition, if any item in $\mathcal{P}$ is accessed again, its score and freshness must be immediately updated. To efficiently manage freshness metadata and reuse decisions, $LBSC^+$ employs a red–black tree structure indexed by $f(i)$. When triggering multiple evictions, the structure 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, denoted as $\mathcal{R}$, as a metric. Formally, for every item i, we define the ratio R_i as follows:

$$\mathcal{R}_i = \frac{\mathcal{C}_i}{s_i}. \tag{10}$$



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

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

Figure 4 shows the distribution of $\mathcal{R}$ for a representative workload (see Section VI-A), along with the corresponding evictions made by BeladySizeCost. Although $\mathcal{R}$ spans a wide range across the entire workload (from 0.0 to 3.0 in Figure 4(a)), BeladySizeCost predominantly evicts items with relatively small $\mathcal{R}$ values. As illustrated in Figure 4(b), more than 99% of the evicted items have $\mathcal{R}$ between 0.0 and 1.0. This observation is consistent with prior works [28], [29]: under typical workloads, items with higher $\mathcal{R}$ (i.e., higher cost per byte) are more likely to be retained in the cache. Therefore, when model inference is required for cache decisions, it is desirable to only sample items with lower $\mathcal{R}$ values, which can further reduce inference overhead. Based on this insight, we propose the following two methods to narrow the sampling range.

Static Splitting Method (SSM). We partition the cached data items into two groups according to a predefined cost-size ratio threshold (denoted as $T_{\mathcal{R}}$). Sampling is then performed exclusively from the subset whose $\mathcal{R}$ values are lower than $T_{\mathcal{R}}$. We formally define SSM in Definition 2, where $baseline$ denotes sampling from the entire cached items while keeping other settings unchanged.

Definition 2. *Given a threshold $T_{\mathcal{R}}$ and a cache containing N_C items, the $baseline$ samples $\mathcal{U}_{ori}$ items whenever the cache is full. We partition the cached items into two disjoint groups $\mathcal{R}^l$ and $\mathcal{R}^h$, such that:*

$$\begin{aligned} &\mathcal{R}_i < T_{\mathcal{R}}, \forall i \in \mathcal{R}^l \\ &\mathcal{R}_j \geq T_{\mathcal{R}}, \forall j \in \mathcal{R}^h \end{aligned} \tag{11}$$

Under the SSM scheme, only $\frac{Num(\mathcal{R}^l)\cdot\mathcal{U}_{ori}}{N_C}$ items are sampled from $\mathcal{R}^l$ whenever the cache reaches capacity, while the total cost $\mathcal{C}$ of SSM still (approximately) converges to the $baseline$. Here, $Num(\mathcal{R}^l)$ indicates the number of items contained in $\mathcal{R}^l$.

The SSM method aims to keep the total cost $\mathcal{C}$ close to that of the $baseline$. Therefore, the choice of $T_{\mathcal{R}}$ is crucial. If $T_{\mathcal{R}}$ is set too small, some items that should be evicted may never be sampled, which can lead to an increase in the total cost $\mathcal{C}$. In contrast, if $T_{\mathcal{R}}$ is too large, $\mathcal{R}^l$ will include more items, limiting the reduction in computation overhead. Consequently, the effectiveness of this method depends on the prior knowledge of $\mathcal{R}$'s distribution.

Algorithm 3 dynamic_splitting($T_{\mathcal{R}}, T_1, T_2$)

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

Dynamic splitting method (DSM). In production environments, the distribution of $\mathcal{R}$ over the entire workload is typically unknown, which makes the SSM difficult to apply. To address this limitation, we propose a dynamic splitting method (DSM), as outlined in Algorithm 3. At the beginning of the workload, DSM samples items proportionally from both $\mathcal{R}^l$ and $\mathcal{R}^h$, and increase the threshold $T_{\mathcal{R}}$ whenever the evicted item's ratio $\mathcal{R}$ (denoted as $E_{\mathcal{R}}$) exceeds $T_{\mathcal{R}}$ (lines 4-9). This design avoids the adverse impact of an overly small initial $T_{\mathcal{R}}$ on the cost $\mathcal{C}$. If, over a sustained period, all evicted items have $\mathcal{R} < T_{\mathcal{R}}$ (i.e., when $\hat{T_1} \geq T_1$), DSM assumes that restricting sampling to $\mathcal{R}^l$ will not noticeably affect the total cost $\mathcal{C}$. Accordingly, the algorithm switches to sampling exclusively from $\mathcal{R}^l$ using the same sampling budget as in SSM (line 11). Since the distribution of $\mathcal{R}$ may evolve over time, DSM periodically resumes sampling from the entire cache after a fixed interval T_2. During this phase, $T_{\mathcal{R}}$ is further increased whenever an evicted item's $\mathcal{R}$ exceeds $T_{\mathcal{R}}$ (lines 12–18). 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

Although continuous retraining enables the model to adapt to evolving workloads, doing so at high frequency is often unnecessary. When the distribution of the current training batch closely resembles that of a previous batch, the corresponding model can be reused to effectively reduce retraining overhead. In this work, we define a batch as a sequence of consecutively generated training data whose count equals a predefined batch size. [2]

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

Prior research [16] has shown that the access patterns in production workloads typically follow a standalone Zipfian distribution. Accordingly, we represent the access patterns of different batches as standalone Zipfian distributions. To quantify the similarity of two batches, i.e., the similarity between their underlying distributions, we adopt the Jensen-Shannon (JS) divergence [30], 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:*

$$JS(\vec{p}, \vec{q}) = \frac{1}{2}(KL(\vec{p}\|\frac{\vec{p}+\vec{q}}{2}) + KL(\vec{q}\|\frac{\vec{p}+\vec{q}}{2})), \tag{12}$$

where

$$KL(\vec{p}\|\vec{q}) = \sum_{i=1}^{m} p_i \log(\frac{p_i}{q_i}). \tag{13}$$

The Zipfian distribution is characterized by $p_i = A/i^{\alpha}$, where A denotes the normalization constant and α is the Zipfian parameter. In practice, these parameters can be estimated using a Least-Squared-based approach [16]. However, this estimation procedure incurs extra computation overheads and may yield inaccurate parameter values. To avoid these issues, our implementation directly samples training items together with their access frequencies for subsequent computation.

Overall, when model retraining is required, we first sample a subset of items from the current batch and compute their JS divergence with respect to previous batches. If the JS divergence between the current batch and all historical batches exceeds a predefined threshold (i.e., $T_{\mathcal{R}}$ in Algorithm 2), we retrain the model and store it. Otherwise, we reuse the previously trained model associated with the minimum JS divergence. More details can be found in our technical report [31].

We maintain an extra small cache regarding the prefix of each trace (e.g., the first one million requests) to set the proper similarity threshold $\mathcal{J}$. Within the small cache, we use the current model and stored historical models to conduct inferences separately and calculate their total cost. The model with the closest total cost to the current model is then selected to calculate the corresponding JS-divergence as our final $\mathcal{J}$ for the entire trace. We assume that the threshold remains fixed for each trace, and demonstrate a significant computation overhead reduction achieved by this method in Section VI.

Using historical models to save retraining costs necessitates keeping the models and sampling data in memory. Nevertheless, the storage overhead is negligible compared to the large cache size. In the implementation, we use LightGBM library [32] to learn the model, and each model consumes only about 6KB of memory, while the entire cache may consume two orders of magnitude larger space.

Further Optimizing Model Storage Overhead. Although one model consumes a small amount of memory, continuously storing models still increases the model utilization space and computation overheads (more JS divergence calculations). Hence, we aim to restrict the number of stored models, and evict models when the model count exceeds a certain threshold (denoted as M). Here, we argue that storing very similar models in memory is redundant and wasteful. To diversify

the stored models, we compare the JS divergence of each two stored models, and the two models corresponding to the minimum similarity value (i.e., models M_i and M_j) are candidates for removal. We remove the model with the smaller average JS divergence (shown in Eq 14) between the two models. The rationale behind this is that the model with a higher average JS divergence is more diverse from other stored models and more valuable than the other one.

$$AVG(M_i) = \frac{1}{M}\sum_{m=1}^{M} JS(\vec{p_i}, \vec{p_m}). \tag{14}$$

V. Integration with cloud databases

$LBSC^+$ is designed as a modular caching framework that can be integrated into a wide range of cloud database systems with minimal architectural changes. In this section, we first demonstrate a concrete integration with FPDB to illustrate the end-to-end deployment workflow. We then discuss how $LBSC^+$ can be extended to commercial cloud databases and the practical considerations involved.

A. Integration with FPDB

FPDB [7] is a cloud analytical database that adopts a storage-disaggregation architecture. It stores original data on the storage node using AWS S3 [9] or MinIO [33]. FPDB supports predicate push-down to reduce network traffic, and caches intermediate query results, referred to as segments, at compute nodes to mitigate data-fetching overhead. Each segment corresponds to a column within a table partition, and segment sizes vary depending on schema and query patterns.

The default 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 $LBSC^+$ at the segment cache layer. Since $LBSC^+$ operates on cache items with associated costs and features, this replacement does not require changes to FPDB's query execution engine.

A key challenge in FPDB integration is that the effective cost of a segment may change across accesses, as different queries may consume only a subset of the segment. To address this, we dynamically estimate the effective cost of a segment as follows:

$$\mathcal{C}_{new} = \frac{\mathcal{C}_{seg} + \mathcal{C}_{old} \cdot (num - 1)}{num}. \tag{15}$$

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 VI.

B. Extension to Commercial Cloud Databases

The modular design of $LBSC^+$ enables its integration into mainstream cloud databases such as OceanBase [34], TiDB [35], or PolarDB [36]. However, several practical challenges must be addressed for effective deployment:

TABLE II
Datasets used in the evaluation.

Dataset	Wiki_1	Wiki_2	SSB
Unique Requests (Million)	11.1	7.43	0.1
Max Data Size (MB)	557.89	674.25	209
Min Data Size (Byte)	42	10	1
Mean Data Size (MB)	0.09	0.12	21.72

Heterogeneous Caching Granularity. Commercial cloud databases differ in what they cache: some systems store final query results, while others cache intermediate results or raw table data. This heterogeneity affects both feature extraction and cost modeling. To ensure accurate predictions, $LBSC^+$ must adapt its cost definition and feature set to match the caching granularity of the target system – just like what we have done in the integration with FPDB.

Feature Collection Overheads. Some systems do not explicitly maintain fine-grained access metadata required by the model (e.g., per-item hit frequency or cost). In such cases, it is necessary to design lightweight feature extraction methods that capture the required features while minimizing the additional overhead introduced.

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

VI. Evaluation

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

A. Experimental Setup

Testbed: We evaluate on Alibaba Cloud ECS [37] with real cloud database deployments. Compute and storage nodes are provisioned with 16 vCPUs and 64GB of memory, and they communicate over a 5Gbps network. All servers run on the Ubuntu 20.04 operating system. For experiments using synthetic datasets, we use a dedicated server with an Intel Xeon W-2275 3.30GHz 14-core CPUs and 512GB of main memory.

Datasets and Benchmark: We evaluate $LBSC^+$ on synthetic traces and real cloud databases, and the key statistics of these datasets are summarized in Table II.

Synthetic Datasets. We first use two real-world traces [13] to evaluate the performance of $LBSC^+$ in a simulated environment. Both traces are collected from a west-coast node that serves photos and other media content for Wikipedia pages. To simulate the cloud environment, we generate the data cost for the items from the original datasets, including the computation cost and transfer cost mentioned in Eq. 1. In detail, we rewrite the transfer cost $\mathcal{TC}_i$ as $k \cdot s_i$ 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 i.

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.

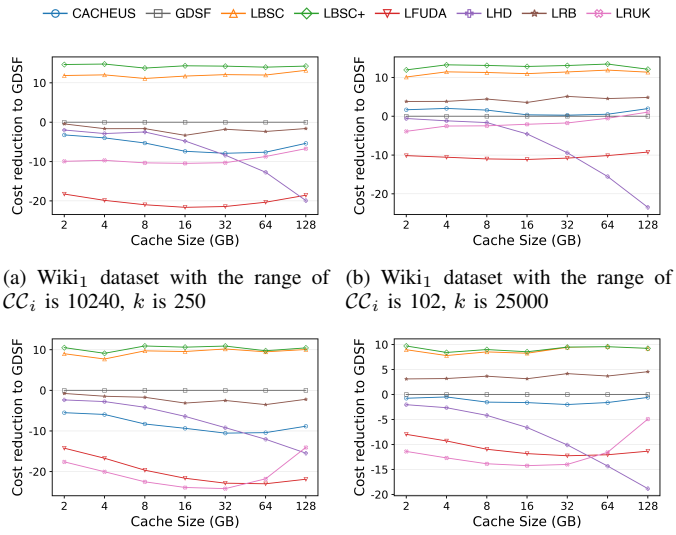
Star Schema Benchmark (SSB) [38]. This benchmark is a variant of TPC-H [39], which consists of one fact table and four dimension tables, and naturally exhibits skewed access patterns and heterogeneous data-fetching costs. SQL workloads are generated from predefined SSB query templates. Predicate parameters are instantiated following Zipfian distributions [40] with skew factors θ ranging from 0.0 to 2.0, and the first 50 queries are used for cache warm-up.

Implementation. We integrate $LBSC^+$ in FPDB [7] and OceanBase [34] for end-to-end system evaluation using SSB. For FPDB, cached units correspond to intermediate query results, and $LBSC^+$ is integrated into FPDB's segment cache layer. We construct analytical queries involving multiple joins and aggregations, and each cache item is associated with the data-fetching cost of the corresponding intermediate result. For OceanBase, we adopt a non-intrusive design by attaching an external cache layer that stores the final query results. Under this setting, we generate relatively simple SQL queries with short per-query execution times. The cost of each cache item is defined as the end-to-end query execution latency.

Baselines: We compare $LBSC^+$ against a set of baselines, including both heuristic-based caching algorithms and machine learning-based methods. These baselines are described as follows.

- *Belady* [14] evicts the item with the furthest access.
- *LFUDA* [41] is an extension of LFU [42], where an item's priority key is determined by its frequency plus the cache age. Note that LFU evicts the item with the least access frequency.
- *LRUK* [6] is an extension of LRU [6] that utilizes k LRU stacks. Note that LRU evicts the item that is the least recently used.
- *GDSF* [28] ranks the cached data by $rank_i = \frac{f_i \cdot \mathcal{C}_i}{s_i} + L_i$, where f_i represents access frequency, $\mathcal{C}_i$ is the *total 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* [20] models the hit density based on age and size, and evicts the item with the lowest hit density.
- *CACHEUS* [5] employs a regret-minimization framework [43] to dynamically choose between two expert eviction policies on each cache miss. Since the original algorithm assumes uniform-sized objects, we re-implement and adapt it to support variable-size data.
- *LRB* [13] 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.
- *LBSC* [11] is a cost-aware learning-based caching framework, which optimizes only eviction while admitting all new items by default. Moreover, it does not incorporate the inference reusing strategy.

Metrics: Our main metric is the total cost $\mathcal{C}$ defined in Section II-A. Due to the diversity of the workloads, the total cost $\mathcal{C}$ can vary widely. Hence, to facilitate visual presentation, we calculate the $\mathcal{C}$ values of every caching algorithm 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 the SSB datasets, the metric is the end-to-end execution time of all queries. Furthermore, the metric in Section VI-B3 is the computation overhead (or cost) reduction compared to the $baseline$, which indicates $LBSC^+$ without the corresponding optimization in Section IV.



(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

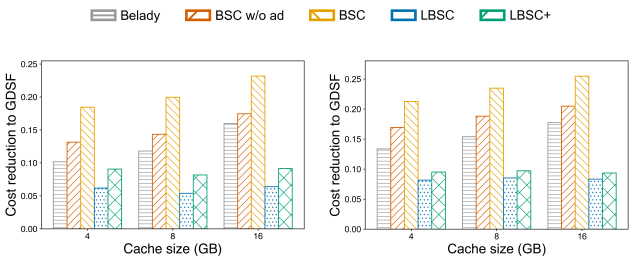
Fig. 5. The cost reduction to GDSF w.r.t varying cache sizes for $LBSC^+$ and the seven baselines on two datasets. On each dataset, the computation cost dominates in the sub-figures (a) and (c), and the transfer cost dominates in the sub-figures (b) and (d).

B. Performance Evaluation

In this subsection, we evaluate the effectiveness of $LBSC^+$ from three perspectives. First, we compare its overall performance against state-of-the-art caching algorithms using synthetic datasets under diverse cost settings and cache sizes. Second, we examine its effectiveness when integrated into real cloud database systems, namely FPDB and OceanBase. Finally, we analyze the impact of the proposed optimization techniques on computational overhead.

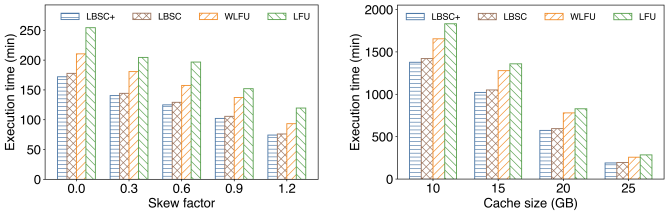
1) Performance on Synthetic Datasets: We evaluate $LBSC^+$ on two synthetic workloads derived from real-world traces under varying cache sizes and cost configurations. For each item, the total cost $\mathcal{C}$ is generated according to Eq. 1, and we vary the relative dominance of computation cost and transfer cost to emulate different cloud environments.

Figure 5 presents the cost reduction of $LBSC^+$ and seven baselines relative to GDSF under different cache sizes. Across all traces and cache sizes, $LBSC^+$ **consistently achieves the lowest total cost**, outperforming all baselines. Overall, $LBSC^+$ reduces the *total cost* by 10%-17% compared to GDSF, regardless of whether computation cost in Figures 5(a) and 5(c) or transfer cost in Figures 5(b) and 5(d) dominates.



(a) Workload-1 (b) Workload-2

Fig. 6. *Total cost* comparison among Belady, BSC w/o ad, BSC, LBSC and $LBSC^+$. Note that "BSC" denotes BeladySizeCost, and "BSC w/o ad" means BeladySizeCost without admission control.



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

Fig. 7. *Total cost* comparison among $LBSC^+$, LBSC, WLFU, and LFU in FPDB.

Figure 5 also highlights the impact of coordinated admission and eviction. Compared to LBSC, which admits all items by default, $LBSC^+$ achieves noticeably lower cost, especially under small cache sizes. This gap is most pronounced when cache capacity is limited (e.g., 2–16 GB in Figure 5), where unconditional admission in LBSC causes frequent eviction of high-utility items. As cache size increases, the performance gap between $LBSC^+$ and LBSC gradually narrows, indicating that admission control is particularly critical under tight memory constraints.

The results in Figure 5 reveal that, except for $LBSC^+$ and LBSC, no existing baseline consistently improves performance as cache size grows. When transfer cost dominates, the *total cost* trend closely follows byte miss rate, allowing LRB and CACHEUS to outperform other heuristics shown in Figures 5(b) and 5(d). In contrast, when computation cost dominates, byte miss rate becomes a poor proxy for *total cost*, and GDSF performs best among heuristics, evidenced in Figures 5(a) and 5(c). Nevertheless, due to limited adaptability to workload dynamics, all baselines remain inferior to $LBSC^+$ across configurations.

So far, we have demonstrated that $LBSC^+$ achieves significant improvements over state-of-the-art baselines. To evaluate the quality of the learning target, i.e., the BeladySizeCost algorithm proposed in Section II-B, we compare $LBSC^+$, LBSC, BeladySizeCost, BeladySizeCost without admission control, and the classical Belady algorithm in Figure 6. As shown in the figure, **BeladySizeCost consistently achieves the lowest total cost** across all workloads and cache sizes, validating its effectiveness as a cost-aware oracle. Moreover, BeladySizeCost clearly outperforms its admission-free variant, confirming that joint admission and eviction are essential while providing a strong upper bound for cost-aware caching.

In contrast, the classical Belady algorithm performs substantially worse once heterogeneous data-fetching costs are introduced in Figure 6, demonstrating that it might be ill-suited as an oracle for cloud databases. Although a performance gap remains between $LBSC^+$ and BeladySizeCost, it is expected due to practical factors such as finite training data and constrained inference budgets. Despite these limitations, $LBSC^+$ remains superior to all learning-based and heuristic baselines.

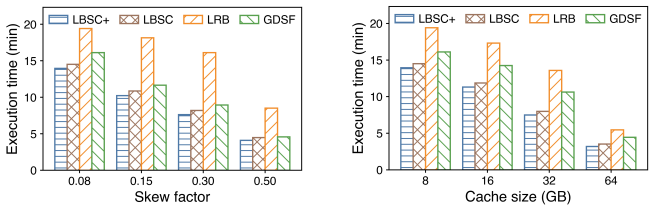
2) Performance under FPDB and OceanBase: We further evaluate $LBSC^+$ in real cloud database systems to demonstrate its practical effectiveness beyond synthetic traces. Specifically, we integrate $LBSC^+$ into FPDB and OceanBase and compare it with representative baseline caching policies.

FPDB. Figure 7 compares $LBSC^+$ with LBSC, LFU and WLFU [7] in FPDB. Specifically, Figure 7(a) presents the execution time of five types of batch queries with a 20GB cache. $LBSC^+$ consistently outperforms LBSC, LFU and WLFU. The most significant cost reduction occurs when θ = 0.3, with $LBSC^+$ outperforming LFU and WLFU by 31.35% and 22.33%, respectively. Overall, $LBSC^+$ outperforms the WLFU algorithm by 21.26% on average in FPDB. To simulate the changing workload, we concatenate seven types of queries, and Figure 7(b) illustrates the execution time of the changing workload with four different cache sizes. We can find that $LBSC^+$ performs best in the changing workload, which demonstrates $LBSC^+$ can adapt well to the changing workload. For example, $LBSC^+$ outperforms WLFU by 26.40% when the cache size is 20GB.

OceanBase. Figure 8 presents the evaluation results on OceanBase. Figure 8(a) shows the execution time of batch queries generated with different skew factors under an 8 GB cache. $LBSC^+$ consistently outperforms GDSF, LRB, and LBSC across all skew settings, highlighting its robustness to workload skew even when caching final query results. Figure 8(b) further examines the impact of cache size on a representative workload. Across all cache sizes, $LBSC^+$ achieves the lowest execution time, reducing execution time by up to 18% compared to GDSF and by 5% compared to LBSC. These results indicate that admission control and cost-aware learning remain beneficial even when the caching granularity shifts from intermediate results in FPDB to final query results in OceanBase.

Taken together, Figures 7 and 8 demonstrate that $LBSC^+$ delivers stable and superior performance in real cloud database systems. Its advantages persist across different caching granularities, workload skews, and cache sizes, underscoring the practicality and generality of the proposed framework.

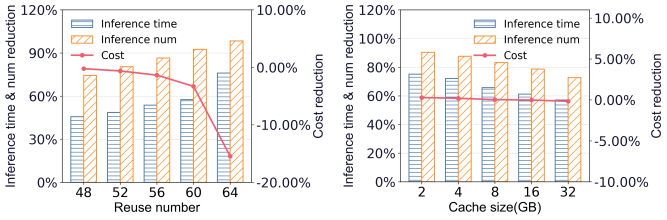
3) Optimizations on $LBSC^+$:
In this subsection, we evaluate the effectiveness of the three optimizations proposed in Section IV, i.e., adaptive inference reusing, sampling with cost-size threshold, and model reusing. Unless otherwise stated, all experiments use the same workload configuration as Figure 5(a), and the $baseline$ refers to



(a) Comparison with different factors in 8GB cache.

(b) Comparison with different cache sizes under one workload.

Fig. 8. *Total cost* comparison among $LBSC^+$, LBSC, LRB, and GDSF in OceanBase.



(a) Impact of reuse number.

(b) Impact of cache size.

Fig. 9. In (a), the histogram shows the reductions in inference time and inference number compared to the $baseline$ under different reuse numbers (i.e., static inference reusing), and the line chart illustrates the corresponding cost reduction. Note that the sample number is 64 by default. In (b), we evaluate the performance of our dynamic inference reusing strategy across different cache sizes.

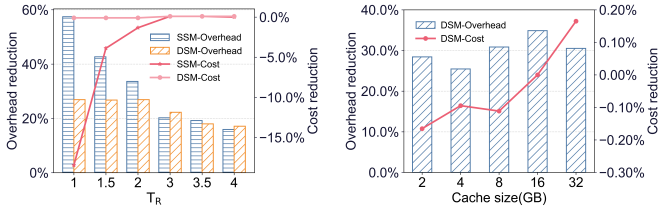
$LBSC^+$ without the corresponding optimization.

Inference Reusing. Figure 9 evaluates the effectiveness of the inference reusing strategy in Section IV-A.

We first examine the static reusing strategy, where a fixed number of previously computed predictions are reused across consecutive cache decisions. As shown in Figure 9(a), when the reuse number increases, both inference time and inference count are reduced, since fewer fresh model inferences are required. However, excessive reuse leads to stale predictions and degrades cache decisions. For instance, with a reuse number of 64, the cost increases by up to 15%, indicating that invalid reuse significantly harms cache decisions. On the other hand, when the reuse number is small, the reductions in inference time and inference number are marginal, resulting in limited improvement in computation overhead. Thus, the static reuse strategy is not robust, as determining an appropriate reuse number is sensitive to different workloads.

Figure 9(b) evaluates the dynamic reusing strategy, where it automatically determines the number of predictions to reuse at each decision based on the freshness metric defined in Eq. 9. Across different cache sizes, adaptive reuse consistently reduces the number of model inferences to about 85% and inference time to about 73% of the baseline, while maintaining comparable or slightly improved cost performance. Therefore, we can find that the dynamic strategy is robust for different workloads: when access patterns becomes stable, it reuses a larger portion of past predictions; when the workload becomes volatile, the strategy automatically reduces the reuse number to refresh inference results.

Overall, Figure 9 demonstrates that adaptive inference reuse



(a) Comparison between the SSM and DSM methods.

(b) Comparison of DSM with varying cache sizes.

Fig. 10. 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.

achieves a robust trade-off between computational efficiency and cache decision quality.

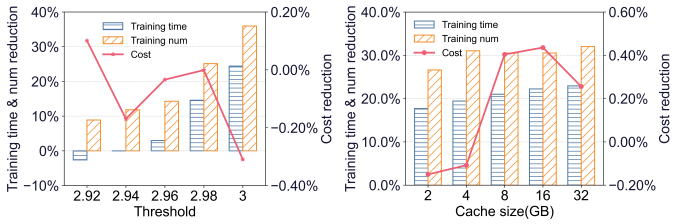
Sampling with Cost-size Threshold. Figure 10 compares the two sampling strategies: SSM and DSM.

Figure 10(a) shows that when the cost–size ratio threshold $T_{\mathcal{R}}$ is properly selected, SSM can substantially reduce computation overhead while preserving cost reduction close to the $baseline$. For example, when $T_{\mathcal{R}}$ is set to 3, the computation overhead reduction reaches about 20%, while the total cost remains nearly unchanged. This is because such a threshold already covers more than 99% of the evictions under BeladySizeCost, indicating that most evicted items lie in the subset $\mathcal{R}^l$. Therefore, restricting sampling to this subset maintains eviction quality while lowering inference overhead.

Figure 10(a) also illustrates a clear trade-off as $T_{\mathcal{R}}$ changes. A larger $T_{\mathcal{R}}$ expands the subset $\mathcal{R}^l$, and the items in $\mathcal{R}^h$ prefer to retain in the cache, so the cost is closer to the $baseline$. But this weakens the overhead reduction effect. For instance, when $T_{\mathcal{R}}$ is set to 1, the computation overhead reduction exceeds 55%, but the total cost degrades by about 18%. When $T_{\mathcal{R}} = 4$, the total cost is almost identical to the $baseline$, while the overhead reduction decreases to only 17%.

Figure 10(b) demonstrates that DSM is more robust when the workload distribution is unknown. Unlike SSM, which is sensitive to the initial threshold choice, DSM dynamically adjusts $T_{\mathcal{R}}$ according to observed eviction behavior. Across different cache sizes, DSM consistently achieves around 30% computation overhead reduction on average, while the corresponding cost loss remains below 0.7%. This indicates that DSM provides stable cost performance together with meaningful efficiency gains, without requiring distribution of the entire workload.

Model Reusing. Next, we evaluate the model reusing strategy in Section IV-C. Figure 11(a) investigates the impact of the threshold on performance, including the total cost $\mathcal{C}$ and the computation overhead. We observe that 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. Additionally, we find the training number reduction is always bigger than the training time reduction. The reason is that computing JS



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

Fig. 11. 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.

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 11(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 can sometimes outperform continuous retraining. In summary, these results demonstrate the superior performance of our method.

VII. Related Work

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

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 [28] 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 [44], RL-Belady [15] and PARROT [12]. More recent learning-based caching systems exploit richer runtime features to guide admission, prefetching, and eviction under various workloads [45], [46]. For modern cloud databases, such as Snowflake [2], Alluxio [47], and PolarDB [1], they typically adopt heuristic caching algorithms (e.g., LRU and LFU), and do not consider data-fetching cost.

Learning Systems. Many system optimizations can be enabled by machine learning and deep learning models [48]–[50]. For instance, in the area of databases, such models can be applied in index optimization [51], [52], cardinality estimation [53], [54], query optimization [55], [56], configuration tuning [57]–[59] and concurrency control [60]. Besides databases, ML-based works have been done in other areas of systems, such as sorting algorithms [61], bloom filter [62], and LSM-tree optimizations [63]–[65]. Specifically, RusKey [64] is an reinforcement learning enhanced key-value store that supports dynamic workloads in an online manner, and they also propose a policy propagation method across levels, reducing the demand for training samples. Despite these advances toward self-driving systems, existing works do not explicitly address cost-aware caching with joint admission and eviction in cloud databases.

VIII. Conclusion

We proposed $LBSC^+$, a learning-based cost-aware caching framework for cloud databases that jointly optimizes admission and eviction under heterogeneous data-fetching costs. At its core, BeladySizeCost generalizes Belady's policy by incorporating cost, size, and near-future reuse, providing principled supervision for a lightweight unified model. To ensure practical efficiency, $LBSC^+$ integrates adaptive inference reuse, cost-aware sampling, and model reuse. Experiments on synthetic traces and real cloud databases, i.e., FPDB and OceanBase, show that $LBSC^+$ reduces total data-fetching cost by up to 17% while cutting computation overhead by up to 90%. In future work, we plan to explore the integration of prefetching techniques into $LBSC^+$. Leveraging the access probability predictions, such an extension may further reduce data-fetching cost and latency.

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 *FAST*, 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 *SIGMOD*, 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 *ICDE*, 2020, pp. 1802–1805.

[4] 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," *PVLDB*, vol. 16, no. 12, pp. 3689–3701, 2023.

[5] L. V. Rodriguez, F. Yusuf, S. Lyons, E. Paz, R. Rangaswami, J. Liu, M. Zhao, and G. Narasimhan, "Learning cache replacement with cacheus," in *FAST*, 2021, pp. 341–354.

[6] 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.

[7] 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," *PVLDB*, vol. 14, no. 11, 2021.

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

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

[10] 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," *PVLDB*, vol. 13, no. 8, pp. 1176–1189, 2020.

[11] Z. Ji, Z. Xie, Y. Wu, and M. Zhang, "Lbsc: A cost-aware caching framework for cloud databases," in *ICDE*, 2024, pp. 4911–4924.

[12] E. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," in *ICML*, 2020, pp. 6237–6247.

[13] 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 *NSDI*, 2020, pp. 529–544.

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

[15] G. Yan and J. Li, "Rl-bélády: A unified learning framework for content caching," in *ACM MM*, 2020, pp. 1009–1017.

[16] G. Yan, J. Li, and D. Towsley, "Learning from optimal caching for content delivery," in *CoNEXT*, 2021, pp. 344–358.

[17] D. S. Berger, N. Beckmann, and M. Harchol-Balter, "Practical bounds on optimal caching with variable object sizes," *POMACS*, vol. 2, no. 2, pp. 1–38, 2018.

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

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

[20] N. Beckmann, H. Chen, and A. Cidon, "Lhd: Improving cache hit rate by maximizing hit density," in *NSDI*, 2018, pp. 389–403.

[21] D. Dobkin and R. J. Lipton, "A lower bound of 12n2 on linear search programs for the knapsack problem," *Journal of Computer and System Sciences*, vol. 16, no. 3, pp. 413–417, 1978.

[22] H.-V. Nguyen and J. Vreeken, "Non-parametric jensen-shannon divergence," in *ECML PKDD*, 2015, pp. 173–189.

[23] 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 *ICDE*, 2023, pp. 394–407.

[24] 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," *PVLDB*, vol. 15, no. 10, pp. 2161–2174, 2022.

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

[26] D. S. Berger, "Towards lightweight and robust machine learning for cdn caching," in *HotNet*, 2018, pp. 134–140.

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

[28] P. Cao and S. Irani, "Cost-aware www proxy caching algorithms." in *USITS*, vol. 12, no. 97, 1997, pp. 193–206.

[29] C. Li and A. L. Cox, "Gd-wheel: a cost-aware replacement policy for key-value stores," in *EuroSys*, 2015, pp. 1–15.

[30] 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.

[31] "Technical report," https://github.com/jzx-bitdb/LBSC/blob/main/technical_report.pdf, 2025.

[32] 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," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

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

[34] 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," *PVLDB*, vol. 15, no. 12, pp. 3385–3397, 2022.

[35] 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," *PVLDB*, vol. 13, no. 12, pp. 3072–3084, 2020.

[36] 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 *SIGMOD*, 2021, pp. 2477–2489.

[37] F. Li, "Cloud-native database systems at alibaba: Opportunities and challenges," *PVLDB*, vol. 12, no. 12, pp. 2263–2272, 2019.

[38] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak, "The star schema benchmark and augmented fact table indexing," in *TPCTC*, 2009, pp. 237–252.

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

[40] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," in *SIGMOD*, 1994, pp. 243–252.

[41] 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.

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

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

[44] A. Jain and C. Lin, "Back to the future: Leveraging belady's algorithm for improved cache replacement," in *ISCA*, 2016, pp. 78–89.

[45] T. Kong, H. Li, Y. Zhao, L. Li, X. Gao, Q. Wu, and J. Cui, "Stscache: An efficient semantic caching scheme for time-series data workloads based on hybrid storage," *PVLDB*, vol. 18, no. 9, pp. 2964–2977, 2025.

[46] F. Zirak, F. Choudhury, and R. Borovica-Gajic, "Selep: Learning based semantic prefetching for exploratory database workloads," *PVLDB*, vol. 17, no. 8, pp. 2064–2076, 2024.

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

[48] R. Zhu, L. Weng, W. Wei, D. Wu, J. Peng, Y. Wang, B. Ding, D. Lian, B. Zheng, and J. Zhou, "Pilotscope: Steering databases with machine learning drivers," *PVLDB*, vol. 17, no. 5, pp. 980–993, 2024.

[49] 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 *ACM MM*, 2015, pp. 25–34.

[50] 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 *ACM MM*, 2015, pp. 685–688.

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

[52] T. Wang, L. Liang, G. Yang, T. Heinis, and E. Yoneki, "A new paradigm in tuning learned indexes: A reinforcement learning enhanced approach," *SIGMOD*, vol. 3, no. 3, pp. 1–26, 2025.

[53] K. Kim, J. Jung, I. Seo, W.-S. Han, K. Choi, and J. Chong, "Learned cardinality estimation: An in-depth study," in *SIGMOD*, 2022, pp. 1214–1227.

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

[55] Q. Cai, C. Cui, Y. Xiong, W. Wang, Z. Xie, and M. Zhang, "A survey on deep reinforcement learning for data processing and analytics," *TKDE*, 2022.

[56] G. X. Yu, Z. Wu, F. Kossmann, T. Li, M. Markakis, A. Ngom, S. Madden, and T. Kraska, "Blueprinting the cloud: Unifying and automatically optimizing cloud data infrastructures with brad," *PVLDB*, vol. 17, no. 11, pp. 3629–3643, 2024.

[57] C. Lin, J. Zhuang, J. Feng, H. Li, X. Zhou, and G. Li, "Adaptive code learning for spark configuration tuning," in *ICDE*, 2022, pp. 1995–2007.

[58] 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," *PVLDB*, vol. 14, no. 7, pp. 1241–1253, 2021.

[59] J. Lao, Y. Wang, Y. Li, J. Wang, Y. Zhang, Z. Cheng, W. Chen, M. Tang, and J. Wang, "Gptuner: An llm-based database tuning system," *ACM SIGMOD Record*, vol. 54, no. 1, pp. 101–110, 2025.
[60] 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 *OSDI*, 2021, pp. 198–216.
[61] A. Kristo, K. Vaidya, U. Çetintemel, S. Misra, and T. Kraska, "The case for a learned sorting algorithm," in *SIGMOD*, 2020, pp. 1001–1016.
[62] 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," *DSE*, vol. 8, no. 3, pp. 234–246, 2023.
[63] W. Yu, S. Luo, Z. Yu, and G. Cong, "Camal: Optimizing lsm-trees via active learning," *SIGMOD*, vol. 2, no. 4, pp. 1–26, 2024.
[64] D. Mo, F. Chen, S. Luo, and C. Shan, "Learning to optimize lsm-trees: Towards a reinforcement learning based key-value store for dynamic workloads," *SIGMOD*, vol. 1, no. 3, pp. 1–25, 2023.
[65] M. Li, H. Chai, S. Luo, H. Dai, R. Gu, J. Zheng, and G. Chen, "Vega: An active-tuning learned index with group-wise learning granularity," *SIGMOD*, vol. 3, no. 1, pp. 1–26, 2025.

Zhongle Xie (Member, IEEE) received the BSc degree from Shanghai Jiao Tong University, in 2014 and the PhD degree from the National University of Singapore, in 2020. He is currently a ZJU100 researcher with School of Software Technology, Zhejiang University. His research interests include AI4DB, vector database, and AI systems. He has served as a PC member for VLDB' 22-24, IEEE ICDE' 23, CIKM' 25, VLDB' 26, EDBT' 26, etc.

Zhaoxuan Ji received the BSc degree from Nankai University, in 2017 and the MSc degree from the University of Chinese Academy of Sciences, in 2020. He is currently pursuing the PhD degree in the School of Computer Science & Technology at Beijing Institute of Technology, under the supervision of Prof. Meihui Zhang. His research interests include AI4DB, LLMs, and data management.

Kehan Liu (Member, IEEE) is currently pursuing an MSc degree in the School of Software Technology at Zhejiang University, under the supervision of Prof. Being Chin Ooi and Dr. Zhongle Xie. Prior to joining Zhejiang University, she earned her BSc degree from China University of Petroleum (Beijing), in 2025. Her research interests focus on caching mechanisms and performance optimization for cloud databases.

Quanqing Xu (Senior Member, IEEE) received the Ph.D. degree from Peking University, Beijing, China, in 2009. He is currently with OceanBase and the Research Institute of Ant Group, Hangzhou, China. He was a research scientist with the Agency for Science, Technology and Research, Singapore, and an adjunct faculty with the Singapore Institute of Technology, Singapore. His research interests mainly include database systems and distributed systems.

Meihui Zhang (Senior Member, IEEE) received the BEng in computer science from the Harbin Institute of Technology, China, in 2008, and the PhD in computer science from the National University of Singapore, Singapore, in 2013. She is currently a professor with the Beijing Institute of Technology, Beijing, China and was an assistant professor with the Singapore University of Technology and Design (SUTD), Singapore from 2014 to 2017. Her research interests include big data management and analytics, large-scale data integration, blockchain systems and AI system. She has served as a vice PC chair/associate editor for TKDE, VLDB'18, ICDE'18, VLDB'19, VLDB'20, SIGMOD'21, ICDE'22, VLDB'23, SIGMOD'23, ICDE'23, VLDB'24, and VLDB'26.