NeurBench: A Benchmark Suite for Learned Database Components with Drift Modeling: [Experiments & Analysis]

ZHANHAO ZHAO, National University of Singapore, Singapore
HAOTIAN GAO, National University of Singapore, Singapore
NAILI XING*, National University of Singapore, Singapore
LINGZE ZENG, National University of Singapore, Singapore
MEIHUI ZHANG, Beijing Institute of Technology, China
GANG CHEN, Zhejiang University, China
MANUEL RIGGER, National University of Singapore, Singapore
BENG CHIN OOI, Zhejiang University, China

Learned database components, which deeply integrate machine learning into their design, have been extensively studied in recent years. Given the dynamism of databases, where data and workloads continuously drift, it is crucial for learned database components to remain effective and efficient in the face of data and workload drift. Robustness, therefore, is a key factor in assessing their practical applicability. Although recent works examine learned database components under specific drift, they fail to enable systematic performance evaluations across a broad range of drift or under customized drift as needed. This paper presents NeurBench, a new benchmark suite that supports evaluating learned database components under measurable and controllable data and workload drift. We quantify diverse types of drift by introducing a key concept called the drift factor. Building on this formulation, we propose a drift-aware data and workload generation framework that effectively simulates real-world drift while preserving inherent correlations. Experimental results demonstrate the effectiveness of NeurBench in generating realistic data and workload drift, while providing insights into the performance of representative learned database components under different drift scenarios.

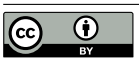
CCS Concepts: • **Information systems → Database performance evaluation**; **Database management system engines**.

Additional Key Words and Phrases: Learned database; data and workload drift; benchmark; AIxDB

ACM Reference Format:
Zhanhao Zhao, Haotian Gao, Naili Xing, Lingze Zeng, Meihui Zhang, Gang Chen, Manuel Rigger, and Beng Chin Ooi. 2026. NeurBench: A Benchmark Suite for Learned Database Components with Drift Modeling: [Experiments & Analysis]. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 214 (June 2026), 27 pages. https://doi.org/10.1145/3802091

*Corresponding author.

Authors' Contact Information: Zhanhao Zhao, National University of Singapore, Singapore, zhanhao@nus.edu.sg; Haotian Gao, National University of Singapore, Singapore, gaohaotian@comp.nus.edu.sg; Naili Xing, National University of Singapore, Singapore, xingnl@comp.nus.edu.sg; Lingze Zeng, National University of Singapore, Singapore, lingze@comp.nus.edu.sg; Meihui Zhang, Beijing Institute of Technology, China, meihui_zhang@bit.edu.cn; Gang Chen, Zhejiang University, China, cg@zju.edu.cn; Manuel Rigger, National University of Singapore, Singapore, rigger@comp.nus.edu.sg; Beng Chin Ooi, Zhejiang University, China, ooibc@zju.edu.cn.



ACM 2836-6573/2026/6-ART214
https://doi.org/10.1145/3802091

1 Introduction

Database systems are increasingly embracing machine learning (ML) techniques in their design. Key database components, such as query optimizers, indexes, and concurrency control (CC), are being upgraded with ML models to boost performance. For example, learned query optimizers [40, 41, 64, 65, 71] treat query optimization as an ML problem to obtain more efficient query plans. Learned indexes [12, 26, 53, 60] leverage ML models to predict the positions of query keys, enabling faster key lookups than traditional indexes. Similarly, learned CC algorithms [56] predict optimal CC actions (*e.g.*, locking strategies and blocking timeouts) for concurrent reads and writes to improve transaction throughput.

One major challenge faced by learned database components is the dynamic nature of databases, commonly referred to as *data and workload drift* [27, 28, 61]. Specifically, the data stored in databases constantly changes over time due to inserts, updates, and deletions, while workloads also evolve, with drift in query patterns [44, 61] and query/transaction arrival rates [34, 54, 55]. In contrast, ML models typically derive insights from static datasets. This fundamental difference between the paradigms of databases and ML can cause learned database components to be ineffective and outdated in the face of data and workload drift. For instance, as data is updated over time, learned query optimizers may lose precision because the knowledge they initially learned for generating query plans becomes obsolete [30, 32]. Likewise, learned CC algorithms may choose actions that were once optimal but no longer suit the current workload with drifted transaction arrival rates [56].

Achieving robust learned database components that remain effective regarding data and workload drift is crucial. In tandem with the advancement of ML techniques, more learned database components are being designed with robustness in mind. For example, learned query optimizers [40, 71] adopt techniques such as periodic model retraining and fine-tuning to handle workload drift. Learned indexes [12, 60] have also become updatable by supporting structural modifications with model retraining in response to new data. These rapid developments naturally raise a pertinent question: How well do existing learned database components perform under data and workload drift, and to what extent does drift affect their performance? Answering this question can provide valuable insights to guide the design of more robust learned database components.

Thus far, several benchmarks have been proposed to examine learned database components under specific types of drift. For example, existing evaluations of learned query optimizers typically model workload drift by training models on some queries and testing on other queries [30, 44], and model data drift by training models on one data snapshot and testing on other snapshots [22]. Similarly, existing works [52, 59] initialize learned indexes on a filtered subset of the data, and reinsert the excluded records at test time to model data drift. Although effective, these drift treatments are case-specific and therefore only partially reveal how drift affects the performance of learned database components. Some important insights, such as how their key design choices impact performance under drift, or the threshold at which learned components cease to be effective, remain implicit. Completely addressing this requires new benchmarks that support comprehensive empirical evaluations across a broad range of data and workload drift.

Systematic benchmarks typically use a metric to quantify the variable of interest and adjust it during evaluations to, in principle, cover all possible scenarios. For example, previous works [8] leverage the well-defined skew factor to measure data skewness and assess system performance under varying skew levels. Following this principle, enabling systematic benchmarks under drift has to address two key requirements: First, designing a metric to quantify drift. Given the various types of drift in databases, such as data drift, query pattern drift, *etc.*, defining a unified and effective drift metric is not trivial. Second, supporting meaningful evaluation under controlled drift. In practice, real-world data and workload drift follow certain patterns, such as data drift often occurring with

specific inherent correlation changes [44]. Controlling drift injection while preserving realistic drift patterns is therefore challenging.

In this paper, we present NeurBench, a novel benchmark suite designed to evaluate learned database components under controllable data and workload drift. We first introduce a concept called the *drift factor* to quantify both data and workload drift. In particular, we model data and workloads as underlying probabilistic distributions and measure drift as the distance between these distributions. We formally define the drift factor based on the distribution distance, *i.e., Jensen-Shannon (JS) divergence* [39], ensuring a bounded value range. Given that the performance of ML models is generally sensitive to the distance between training and testing distributions [2, 17, 66], the drift factor allows us to define and control evaluation scenarios with varying degrees of drift between training and testing data and workloads, thereby enabling the comprehensive robustness evaluation for learned database components.

Leveraging this concept, we then design NeurBench to supply data and workload drift according to a specified drift factor. Ideally, it would be desirable to adopt real-world data and workload drift directly. Nonetheless, purely relying on real-world drift constrains the scope of drift that can be evaluated. Real drift only reflects what has happened, and it cannot cover unseen or more severe drift that may occur in the future. Moreover, real data drift may be incomplete or inaccessible. For example, the public IMDB dataset switched to only providing partial data after 2017 [7], making real drift not fully observable. To address these problems, we opt to synthesize data and workloads that capture realistic drift patterns while allowing effective control over the drift factor. However, real-world data and workloads typically exhibit complex correlations. For example, movie ratings and the number of votes are highly correlated in the IMDB dataset [21]. Thus, we must capture the inherent correlations in the drift generation process to approximate real data and workload drift.

We therefore propose a drift-aware data and workload generation framework that enables the synthesis of realistic drift. Specifically, we first utilize a base generative model, called diffuser, to fit the underlying distribution of the real data and workload. We use the Denoising Diffusion Probabilistic Model (DDPM) [20] as the base generative model. We then train an independent *drifter* module to capture the real-world data and workload drift patterns. Given a drift factor, the drifter guides the base model, *i.e.*, the denoising process of DDPM, to synthesize data and workloads that approximate real drift with the desired severity. Further, by decoupling the drifter from the base model, our framework avoids retraining the base model when adapting to new drift patterns, enabling generalization across various types of drift.

In summary, we make the following contributions:

- We introduce NeurBench, a practical benchmark suite that facilitates the evaluation of learned database components under controllable data and workload drift. The suite will help further advance the design of next generation learned components.
- We formalize a unified concept called the drift factor to model data and workload drift, enabling the systematic performance evaluation of learned database components under diverse drift scenarios. NeurBench allows users to configure the settings to match their own environment and applications.
- We propose a drift-aware data and workload generation framework that synthesizes data and workload drift according to a specified drift factor while preserving their inherent correlations.
- We conduct extensive experiments to evaluate NeurBench, and the results demonstrate the effectiveness of NeurBench in generating realistic data and workload drift.
- We use NeurBench to evaluate start-of-the-art learned query optimizers, learned indexes, and learned CC. The results reveal how their design choices trade off performance under varying degrees and kinds of drift, offering insights into their robustness and generalizability.

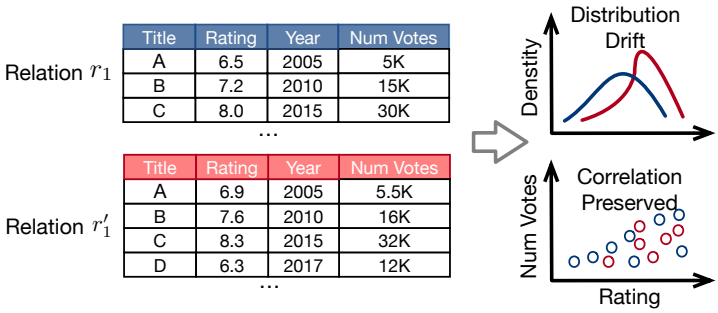


Fig. 1. Example of Real-World Data Drift

NeurBench is a benchmark suite designed to evaluate the efficiency and robustness of learned database components in NeurDB [46, 69].

The remainder of the paper is structured as follows. Section 2 provides relevant background. Section 3 formally defines the drift factor and describes the overall design of NeurBench. Section 4 presents the drift-aware data and workload generation framework in detail. Section 6 presents the implementation of NeurBench. Section 6 describes the experimental setup, and Section 7 shows the results. Section 8 discusses the related work, and Section 9 concludes.

2 Background

In this section, we provide the necessary background on learned database components, data and workload drift in databases, and diffusion models. Since learned database components typically rely on ML models, we provide a general definition of learned database components by mapping them to learnable ML functions:

Definition 1 (Learned Database Component)**.** Consider a database that stores data $\mathbf{R}$ and handles workload $\mathbf{Q}$. A learned database component can be formulated as an ML function $f(\cdot;\theta) : \mathbf{Z} \rightarrow \mathbf{Y}$, where θ denotes the trainable parameters. The input space $\mathbf{Z}$ consists of features derived from the data $\mathbf{R}$ and workload $\mathbf{Q}$, such as data distributions and query patterns. The output space $\mathbf{Y}$ represents the predictions or decisions, such as estimated query plans, index positions, or concurrency control actions.

2.1 Data and Workload Drift

Data stored in databases, as well as the workloads they handle, are subject to continuous drift. *Data drift* occurs due to operations including inserts, updates, deletes, and schema modifications. *Workload drift* refers to variations in the queries and transactions processed by the database over time, which can be further categorized into, 1) Query pattern drift: changes in query structure, such as join patterns (`FROM` clauses) and predicates (`WHERE` clauses); 2) Arrival rate drift: fluctuations in the volume and concurrency of queries and transactions.

The effectiveness of ML models is generally related to the distance between training and testing distributions [2, 17, 66], and therefore, we consider measuring the data and workload drift based on the distance of their underlying distributions. Drift may occur in the data, the workload, or both, depending on whether the distribution of $\mathbf{R}$, $\mathbf{Q}$, or their joint distribution drifts over time.

Definition 2 (Data and Workload Drift)**.** Consider a database that operates over a time window $\mathbf{T}$ consisting of N time slots, *i.e.*, $\mathbf{T} = (ts_1, ts_2, \ldots, ts_N)^\top$, where $N \geq 2$. Let $\mathbf{R}^{(i)}$ and $\mathbf{Q}^{(i)}$ be random variables representing the state of the data and the workload, respectively, at time ts_i. Data and workload drift occur when the distributions of data $\mathbf{R}$ and workload $\mathbf{Q}$ exhibit temporal changes

across the time window $\mathbf{T}$. Let $\mathbb{P}_{\mathbf{R}^{(i)},\mathbf{Q}^{(i)}}(\mathbf{r},\mathbf{q})$ represent the joint probability density function of data $\mathbf{R}$ and workload $\mathbf{Q}$ at time t_i. Drift happens if, for any two timestamps $i, j \in N$, where $i \neq j$,

$$\mathbb{P}_{\mathbf{R}^{(i)},\mathbf{Q}^{(i)}}(\mathbf{r},\mathbf{q}) \neq \mathbb{P}_{\mathbf{R}^{(j)},\mathbf{Q}^{(j)}}(\mathbf{r},\mathbf{q}).$$

Constructing realistic data and workload drift that captures meaningful distribution drift is crucial. As shown in Figure 1, we illustrate using movie ratings and their associated number of ratings (num_rates) as an example. Let the relation r_1 denote a data snapshot from 2013 and r_1' from 2017. First, while the schema remains unchanged, the underlying data distribution can drift over time as new data arrives [44, 54]. Second, inherent data correlations are largely preserved. Real-world data drift typically maintains the original correlation structure and distributional shape, with shifts mainly in density. Here, the movie rating is consistently positively correlated with the num_rates. For example, movie ratings and num_rates consistently exhibit a positive correlation as movies get added. Other attribute pairs, such as country and language, or release_year and genre, also tend to remain highly correlated across time, as observed in prior studies [30, 44]. Similarly, workload drift typically manifests as changes in the frequency or usage of query templates and predicates, while still preserving structural elements such as join patterns or access paths [54, 55]. These insights form the foundation for the drift modeling framework in NeurBench.

In our design, we learn the underlying distribution drift from real data and workloads, and propose a generation framework that enables the controllable synthesis of data and workload drift.

2.2 Diffusion Model

Diffusion models are powerful generative models that have been recently adopted to synthesize high-quality data, such as images [10], audio [35], and tabular data [25, 36]. Inspired by the physical process of diffusion, where information (*e.g.*, particles, data) is gradually dispersed over time, these models learn to reverse this process so that a target distribution can be generated from a given prior distribution. In particular, diffusion models operate in two key stages, *i.e.*, forward process and reverse process, with slight variations in the details depending on the specific model. For illustration purposes, we use the Denoising Diffusion Probabilistic Model (DDPM) as a representative to formally explain these stages:

Diffusion (Forward) Process. This process incrementally corrupts the data by adding Gaussian noise in discrete timesteps, ultimately transforming it into a simple noise distribution. Given a data sample $\mathbf{x}_0 \sim q(\mathbf{x})$ from the true data distribution $q(\mathbf{x})$, the forward process is formulated as a Markov chain:

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}),$$

where β_t represents the variance schedule controlling the noise added at each step t. The data distribution after T timesteps, $q(\mathbf{x}_T)$, approximates an isotropic Gaussian distribution $\mathcal{N}(\mathbf{0},\mathbf{I})$.

Denoising (Reverse) Process. This process reconstructs the target data $\mathbf{x}_0$ by iteratively denoising $\mathbf{x}_t$ in reverse timesteps, starting from $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0},\mathbf{I})$. The reverse process $p(\mathbf{x}_{t-1}|\mathbf{x}_t)$ can be approximated as a Gaussian distribution:

$$p(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}(\mathbf{x}_t, t; \theta), \sigma_t^2\mathbf{I}),$$

where $\boldsymbol{\mu}$ is learned parameters that estimate the mean of the reverse process, while σ_t^2 is derived from β_t.

Guided Diffusion. The reverse process can incorporate auxiliary information c, yielding a conditional reverse process $p(\mathbf{x}_{t-1}|\mathbf{x}_t, c)$, where the conditioning variable c may represent factors such as correlation-preserving constraints, or contextual information [10, 36]. By applying the Bayes'

theorem, $p(\mathbf{x}_{t-1}|\mathbf{x}_t, c)$ can be decomposed as:

$$p(\mathbf{x}_{t-1}|\mathbf{x}_t, c) \propto p(\mathbf{x}_{t-1}|\mathbf{x}_t)\Pr(c|\mathbf{x}_{t-1}, \mathbf{x}_t).$$

In particular, $p(\mathbf{x}_{t-1}|\mathbf{x}_t)$ should perform unconditional generation, while $\Pr(c|\mathbf{x}_{t-1}, \mathbf{x}_t)$ guides the reverse process to introduce the specified condition c.

Inspired by such guided diffusion models, we treat the introduction of drift to data and workloads as a specific condition, and propose a novel drift-aware data and workload generation framework based on DDPM. Our generation process differs from prior work [49] in that we directly inject drift signals into the denoising steps. This allows NeurBench to generate drifted data controlled by a drift factor with correlation retained.

3 Overview of NeurBench

In this section, we define the drift factor and then present the system overview of NeurBench.

3.1 Drift Factor Modeling

We define a drift factor that can uniformly and effectively describe the data and workload drift. In our problem setting, the drift factor must satisfy two key properties: 1) It should effectively capture the differences between data or workloads while providing a bounded value range for users to specify the desired drift. 2) Its value should provide a consistent representation for all types of drift.

According to Definition 2, where drift is mapped to the underlying distribution of data and workloads, we define the drift factor based on the distance of distributions. To satisfy the properties above, we utilize the *Jensen-Shannon (JS) divergence* [39]. Specifically, given a distribution $\mathbb{P}$ and the drifted one $\mathbb{P}'$, their JS divergence $\mathcal{D}_{JS}(\mathbb{P} \parallel \mathbb{P}')$ is defined as:

$$\mathcal{D}_{JS}(\mathbb{P} \parallel \mathbb{P}') = \frac{1}{2}\mathcal{D}_{KL}(\mathbb{P} \parallel M) + \frac{1}{2}\mathcal{D}_{KL}(\mathbb{P}' \parallel \mathbb{M}),$$

where $\mathbb{M} = \frac{1}{2}(\mathbb{P} + \mathbb{P}')$, and D_{KL} is the Kullback–Leibler (KL) divergence, which is defined by:

$$\mathcal{D}_{KL}(\mathbb{P} \parallel \mathbb{P}') = \sum_x \mathbb{P}(x) \log \frac{\mathbb{P}(x)}{\mathbb{P}'(x)}.$$

$\mathcal{D}_{JS}$ satisfies $\mathcal{D}_{JS}(\mathbb{P} \parallel \mathbb{P}') = \mathcal{D}_{JS}(\mathbb{P}' \parallel \mathbb{P})$, and is bounded within $[0, \log 2]$. Normalizing it by $\log 2$, the drift factor d is defined as:

$$d(\mathbb{P}, \mathbb{P}') = \frac{\mathcal{D}_{JS}(\mathbb{P} \parallel \mathbb{P}')}{\log 2}, \quad d \in [0, 1].$$

We allow users to specify the desired drift level using a *drift factor* d, where $d \in [0, 1]$. By default, users can independently indicate drift factors for either data or workload to achieve effective control over each. A drift factor of d=0 indicates no drift, while d=1 denotes that the entire data or workload is fully drifted.

3.2 System Overview

As shown in Figure 2, we design NeurBench with two key modules, a *drift-aware data and workload generator*, and a *performance evaluator*. In particular, the *generator*, based on guided diffusion, consists of: 1) the *diffuser*, a DDPM-based generative model that learns the underlying distribution of the real data and workloads; 2) the *drifter*, a classification model that guides the diffuser to generate data and workloads with specified drift. The evaluator is responsible for assessing the performance of learned database components under drift. We organize the overall workflow of NeurBench into three stages: training, drift-aware generation, and evaluation. In the following, we focus on how to use NeurBench to conduct experiments under data drift for illustration purposes.

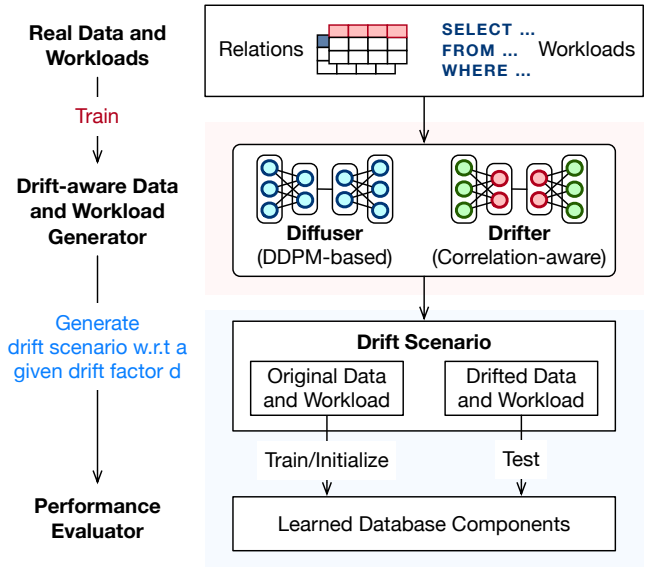


Fig. 2. System Overview of NeurBench

Diffuser and Drifter Training. We input real data and workloads to train the drift-aware data and workload generator, with the diffuser and drifter trained separately. The diffuser is first trained on a fixed data snapshot (*e.g.*, 2013 data) to understand the underlying distribution $\mathbb{P}_{\text{base}}$, following the DDPM framework. The drifter is then trained to model real drift patterns using a sequence of real data snapshots (*e.g.*, 2013–2016). In particular, the drifter maps real drift patterns to modulation vectors that condition the diffuser's denoising process, enabling the diffuser to generate data that reflects realistic drift while preserving key correlations. By decoupling the training of the diffuser and drifter, our framework allows the diffuser to remain fixed while adapting to new types of drift by simply learning new drifter mappings, thereby improving generality and reusability. We will detail the diffuser and drifter training in Section 4.3.

Drift-aware Data and Workload Generation. Given a specified drift factor d and original data as input, we use the generator to drift the given data to a target data that aligns with d. In particular, starting from a standard normal distribution, the generator iteratively invokes the diffuser to denoise the distribution, while leveraging the drifter to introduce controlled drift at each denoising step, gradually transforming the distribution into the target distribution. Section 4.4 details how to use the trained diffuser and drifter to generate required data and workload drift.

Performance Evaluation. We assess the robustness of learned database components under data and workload drift using the evaluator. For a given drift factor d, we first use the generator to produce a pair of data: the base and its drifted counterpart. A target component is then trained on the original data and workload and tested on the drifted data and workload. By repeating this procedure across a spectrum of drift factors, we can systematically evaluate performance degradation to varying degrees of drift.

4 Controllable Drift Synthesis Framework

In this section, we detail the design of the drift-aware data and workload generation framework.

4.1 Mapping Data and Workload Drift to Distribution Drift

Here we describe how we map both data and workload drift to the corresponding distribution drift in detail. For data, we consider a relational database $\mathbf{R}$ containing multiple tables, each comprising a set of attributes. The distribution of $\mathbf{R}$, $\mathbb{P}_{\mathbf{R}}$, can be approximated as the product of its marginal attribute distributions, *i.e.*, $\mathbb{P}_{\mathbf{R}}(A_1, A_2, \ldots, A_m) \approx \prod_{i=1}^{m} \mathbb{P}(A_i)$, where $\mathbb{P}(A_i)$ represents the distribution of the i-th attribute. Note that this approximation is used solely for distribution measurement. During data generation, we jointly synthesize multiple columns per tuple to preserve attribute correlations, as discussed in Section 4.4. For each attribute, we employ specific strategies based on its data type to approximate its distribution. Categorical attributes (*e.g.*, `CHAR`, `VARCHAR`) and discrete numerical values are handled by computing distributions over all observed values in the table. Continuous numerical attributes are approximated by discretizing values into a fixed number of bins. The corresponding distribution is then estimated by computing the probability density of each bin. This binning strategy is commonly employed in database histograms to estimate attribute distributions and query selectivity.

For a workload consisting of N queries $\mathbf{Q} = \{\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_N\}$, we define the workload distribution $\mathbb{P}_{\mathbf{Q}}$ as the distribution over *query templates*. Specific features, such as query patterns, query arrival rates, *etc.*, characterize each query template. To model query patterns and arrival rates in a unified manner, we define a workload table with 3 attributes: Join Pattern, Predicate, and Query Interval. The Join Pattern and Predicate columns capture the query pattern, representing table join patterns (*e.g.*, $r_1 \bowtie r_2$), and predicates (*e.g.*, Bal = '10K'). The Query Interval column quantifies the arrival rate by recording the time gap between consecutive queries. We control the distributions of the workload table to synthesize workloads with query pattern drift, arrival rate drift, or both, while preserving the inherent correlations. Note that the workload table used here is for illustration purposes; it can be extended with additional attributes for more complex queries.

4.2 Theoretical Analysis

To generate data and workload drift that satisfy a specified d, we provide a formal problem definition and then theoretically explain how we enable the drift-aware data and workload generation.

4.2.1 Problem Definition. We define a database running in a time window $\mathbf{T}$ with $N (N \geq 2)$ time slots, *i.e.*, $\mathbf{T} = (ts_1, ts_2, ..., ts_N)^\top$. Recall that we denote $\mathbf{R}$ and $\mathbf{Q}$ as random variables of the data and workload in the database, respectively, and we map them to their underlying distributions $\mathbb{P}_{\mathbf{R}}^{(i)}$ and $\mathbb{P}_{\mathbf{Q}}^{(i)}$ at ts_i. Formally, we define the drift-aware data and workload generation below.

Definition 3 (Drift-aware Data and Workload Generation)**.** Given a series of drift factors $\mathbf{D} = (d_1, d_2, ..., d_N)^\top$, where each drift factor d_i quantifies the degree of change in the distribution between adjacent time slots (ts_j, ts_i). The distribution of either data or workload at time ts_j is obtained by transforming the distribution at time ts_i with a function g, according to the drift factor d_i:

$$\mathbb{P}_{\mathbf{R}}^{(j)} = g_{\mathbf{R}}(\mathbb{P}_{\mathbf{R}}^{(i)}, d_i, C), \quad \mathbb{P}_{\mathbf{Q}}^{(j)} = g_{\mathbf{Q}}(\mathbb{P}_{\mathbf{Q}}^{(i)}, d_i, C),$$

where C is the corresponding inherent correlations, and $g_{\mathbf{Z}} : \Omega_{\mathbf{Z}} \times [0, 1] \times C \to \Omega_{\mathbf{Z}}$ is the drift function applied on input space $\mathbf{Z}$, transforming its distribution on the same domain.

Given a distribution $\mathbb{P}^{(i)}$ and the drift factor d_i, the generated new distribution $\mathbb{P}^{(j)}$ should 1) exhibit drift quantified by d from $\mathbb{P}^{(i)}$; and 2) preserve inherent correlations C. The quality of $\mathbb{P}^{(j)}$ degrades as either requirement is violated. Mathematically, this is equivalent to finding a hypersurface in $\Omega_{\mathbf{Z}}$, the domain of input space $\mathbf{Z}$ (*i.e.*, either data and workload), which satisfies

the equation:

$$d(\mathbb{P}_i, \mathbb{P}_j) \approx d^* \quad \text{subject to} \quad C(\mathbb{P}_j) \approx C(\mathbb{P}_i). \tag{1}$$

However, there are infinite solutions to Equation 1, which makes it difficult to interpret and control. To solve this, we limit the expressiveness of $\mathbb{P}_j$ through a DDPM-based generator.

4.2.2 Theoretical Proof. Given a desired drift factor d, our objective as outlined in Section 4.2.1 is to generate the drifted data $\mathbf{x}_{\text{drift}}$ by sampling from $p(\mathbf{x}_{t-1}|\mathbf{x}_t, d)$, *i.e.*, a conditional DDPM reverse process that incorporates the desired drift d. As outlined in Section 2.2, $p(\mathbf{x}_{t-1}|\mathbf{x}_t, d)$ can be decomposed as:

$$p(\mathbf{x}_{t-1}|\mathbf{x}_t, d) \propto p(\mathbf{x}_{t-1}|\mathbf{x}_t)\Pr(d|\mathbf{x}_{t-1}, \mathbf{x}_t), \tag{2}$$

where $p(\mathbf{x}_{t-1}|\mathbf{x}_t)$ can be modeled by the denoising process of DDPM, *i.e.*, $p(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}(\mathbf{x}_t, t; \theta), \sigma_t^2\mathbf{I})$, and $\Pr(d|\mathbf{x}_{t-1}, \mathbf{x}_t)$ guides the process to introduce the specified drift d. Since the drifted data $\mathbf{x}_{drift}$ is the target output, Equation (2) can be rewritten as:

$$p(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_{\text{drift}}) \propto p(\mathbf{x}_{t-1}|\mathbf{x}_t)\Pr(\mathbf{x}_{\text{drift}}|\mathbf{x}_{t-1}, \mathbf{x}_t). \tag{3}$$

Note that the mean of the Gaussian distribution, $\boldsymbol{\mu}(\mathbf{x}_t, t; \theta)$, determines the trajectory of the denoising process [10, 36]. We therefore leverage $\Pr(\mathbf{x}_{\text{drift}}|\mathbf{x}_{t-1}, \mathbf{x}_t)$ to introduce a controlled drift to the mean $\boldsymbol{\mu}$.

Theorem 1. The controlled reverse distribution $p(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_{\text{drift}})$ is able to be approximated using $\Pr(\mathbf{x}_{\text{drift}}|\mathbf{x}_t)$ and its gradient at $\mathbf{x}_t$, or $\nabla_{\mathbf{x}_t}\Pr(\mathbf{x}_{\text{drift}}|\mathbf{x}_t)$.

Proof. For Equation (3), the first part is approximated by a Gaussian model, while the second part can be simplified using conditional independence, which transforms the equation to:

$$p(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_{\text{drift}}) \approx z\mathcal{N}(\mathbf{x}_t; \boldsymbol{\mu}, \Sigma)\Pr(\mathbf{x}_{\text{drift}}|\mathbf{x}_t),$$

where z is the normalizing factor. Next, as proven by Dickstein et al. [50], we can perform Taylor expansion around $\boldsymbol{\mu}$ to further approximate $p(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_{\text{drift}})$ so that a perturbed Gaussian distribution can be used to model it:

$$\begin{aligned}
\log p(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_{\text{drift}}) &\approx \log \mathcal{N}(\mathbf{x}_t; \boldsymbol{\mu}, \Sigma) + \log \Pr(\mathbf{x}_{\text{drift}}|\mathbf{x}_t) \\
&\approx -\frac{1}{2}(\mathbf{x}_t - \boldsymbol{\mu})^\top \Sigma^{-1}(\mathbf{x}_t - \boldsymbol{\mu}) + (\mathbf{x}_t - \boldsymbol{\mu}) \cdot g + C \\
&= -\frac{1}{2}(\mathbf{x}_t - \boldsymbol{\mu} - \Sigma \cdot g)^\top \Sigma^{-1}(\mathbf{x}_t - \boldsymbol{\mu} - \Sigma \cdot g) + C \\
&= \log \mathcal{N}(\mathbf{x}_t; \boldsymbol{\mu} + \Sigma \cdot g, \Sigma),
\end{aligned}$$

where $g = \nabla_{\mathbf{x}_t}\Pr(\mathbf{x}_{\text{drift}}|\mathbf{x}_t)$, and the constant term C can be ignored. In other words, $p(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_{\text{drift}})$ can be approximated by $\mathcal{N}(\mathbf{x}_t; \boldsymbol{\mu} + \Sigma \cdot \nabla_{\mathbf{x}_t}\Pr(\mathbf{x}_{\text{drift}}|\mathbf{x}_t), \Sigma)$. □

Conceptually, Theorem 1 ensures the generated data moves toward the target drifted distribution. Unlike existing data generators [49], our method directly injects drift signals into the denoising steps, and with the ability of the DDPM to preserve correlations in the latent space, we enable NeurBench to generate drifted data controlled by a drift factor with correlation retained.

4.3 Diffuser and Drifter Training

Based on Theorem 1, we design a generation framework comprising two key components: the diffuser and the drifter. Concretely, the Diffuser learns the reverse denoising distribution to map $\mathcal{N}(0, I)$ to data, while the Drifter conditions each reverse step to match a specified drift factor. We illustrate the training process of the diffuser and drifter in Figure 3.

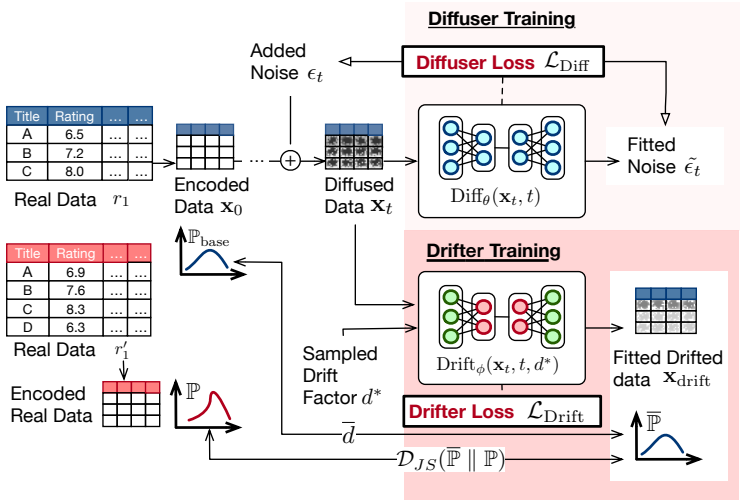


Fig. 3. Diffuser and Drifter Training

Diffuser Training. The diffuser `Diff` serves as the core generative model, learning to approximate the unconditional distribution $p(\mathbf{x}_{t-1}|\mathbf{x}_t)$. To achieve this, we train `Diff` to predict the mean $\boldsymbol{\mu}(\mathbf{x}_t, t; \theta)$ of the reverse denoising process, which is directly related to the noise added during the forward process. In particular, the reverse process can be rewritten in terms of noise ϵ_t, where: $\boldsymbol{\mu}(\mathbf{x}_t, t; \theta) = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \sqrt{1-\alpha_t} \cdot \epsilon_t\right)$, and α_t is a parameter derived from the variance schedule σ_t. This formulation makes noise reconstruction a central task in training, and thus, the noise prediction output of the diffuser at timestep t, or $\texttt{Diff}(\mathbf{x}_t, t)$, is trained by minimizing the noise reconstruction error:

$$\mathcal{L}_{\texttt{Diff}} = \mathbb{E}_t \left[\|\epsilon_t - \texttt{Diff}(\mathbf{x}_t, t)\|^2 \right].$$

As shown in Figure 3, the diffuser is trained to reconstruct samples from the base data with the distribution $\mathbb{P}_{\text{base}}$ by learning to predict the added noise during the forward diffusion process. In each training iteration, a clean encoded sample $\mathbf{x}_0$ is drawn from real data $r_1 \sim \mathbb{P}_{\text{base}}$. A timestep t and Gaussian noise ϵ_t are sampled, and noise is added to $\mathbf{x}_0$ to produce the diffused sample $\mathbf{x}_t$. The diffuser `Diff` takes $\mathbf{x}_t$ and t as inputs, and predicts the noise $\hat{\epsilon}_t$ used in denoising. We train `Diff` by minimizing the mean squared error between the predicted noise $\hat{\epsilon}t$ and the true noise ϵ_t, denoted as $\mathcal{L}_{\texttt{Diff}}$. This enables the diffuser to capture the underlying distribution and latent correlations in $\mathbb{P}_{\text{base}}$.

Drifter training. The drifter `Drifter` is used to fit $\Pr(\mathbf{x}_{\text{drift}}|\mathbf{x}_t)$ to generate $\mathbf{x}_{\text{drift}}$ with the same size as the current noisy data $\mathbf{x}_t$, and satisfies any drift factor d, *i.e.*, $\mathbf{x}_{\text{drift}} = \texttt{Drifter}(\mathbf{x}_t, t, d)$. After that, the expected drift factor d^* is input into `Drifter` to compute $\nabla_{\mathbf{x}_t} \Pr(\mathbf{x}_{\text{drift}}|\mathbf{x}_t)$ and guide `Diff` toward the expected distribution. The drifter is trained to guide the denoising process toward drifted target distributions while preserving key statistical properties of the base data. To achieve this, we adopt a composite loss function so that it balances among the accuracy of drift-introduction, preservation of original correlations, and coherence with the real data.

The detailed algorithm to train the drifter is illustrated in Algorithm 1. It first initializes drifter parameters ϕ, pre-computes α_t and $\bar{\alpha}_t$ based on $\{\beta_t\}_{t=1}^T$, where both the definition of $\{\beta_t\}$ and the computation of α_t and $\bar{\alpha}_t$ (Lines 1-2). After that, it starts the training by looping through the training steps. At each training step, it first samples a clean base sample x_0 from $\mathbb{P}_{\text{base}}$ and selects a target real data $\mathbb{P}$ from $\{\mathbb{P}_{\text{target}}\}$. After that, a random drift factor d^*, a timestep t, and a noise $\epsilon \sim \mathcal{N}(0, I)$ are sampled (Lines 4-5). The drifter then constructs a noisy sample $\mathbf{x}_t$ via the forward

Algorithm 1: Drifter Training Procedure

Input: Base distribution $\mathbb{P}_{\text{base}}$, drifted target distribution set $\{\mathbb{P}_{\text{target}}\}$, schedule variables $\{\beta_t\}_{t=1}^{T}$, loss balancer λ_1, λ_2

Output: Trained drifter $\texttt{Drifter}_\phi$

1 Initialize drifter parameters ϕ;

/* Pre-compute α_t and $\bar{\alpha}_t$ for all timesteps */

2 $\{\alpha_t\}_{t=1}^{T} \leftarrow \{1-\beta_t\}_{t=1}^{T}$, $\{\bar{\alpha}_t\}_{t=1}^{T} \leftarrow \{\prod_{i=1}^{t} \alpha_i\}_{t=1}^{T}$;

3 **while** not converged **do**

4 $\quad \mathbf{x}_0 \sim \mathbb{P}_{\text{base}}, \mathbb{P} \sim \{\mathbb{P}_{\text{target}}\}$;

5 $\quad d^* \sim U(0,1), t \sim \{1,\ldots,T\}, \boldsymbol{\epsilon}_t \sim \mathcal{N}(0,\mathbf{I})$;

6 $\quad \mathbf{x}_t \leftarrow \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_t$;

7 $\quad \mathbf{x}_{\text{drift}} \leftarrow \texttt{Drifter}_\phi(\mathbf{x}_t, t, d^*)$;

8 $\quad \overline{\mathbb{P}} \leftarrow p(\mathbf{x}_{\text{drift}})$; // get distribution of $\mathbf{x}_{\texttt{drift}}$

9 $\quad \bar{d} \leftarrow \frac{1}{\log 2}\mathcal{D}_{JS}(\mathbb{P}_{\text{base}} \parallel \overline{\mathbb{P}})$;

10 $\quad \mathcal{L}_{\text{Drift}} = \left\| \bar{d} - d^* \right\|_2^2 + \lambda_1 \left\| C(\mathbf{x}_{\text{drift}}) - C(\mathbf{x}_t) \right\|_2^2 + \lambda_2 \mathcal{D}_{JS}(\overline{\mathbb{P}} \parallel \mathbb{P})$;

11 $\quad$ Update ϕ using gradient descent;

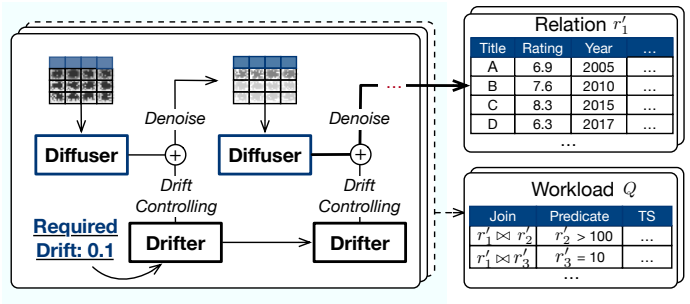


Fig. 4. Data and Workload Generation Process

diffusion process and applies Drift to denoise, retrieving the drifted sample $\mathbf{x}_{\text{drift}}$, and computes its distribution $\overline{\mathbb{P}}$ (Lines 6-8). The drift factor between distributions of the base and synthesized data $\bar{d}$ is computed afterward (Line 9). The loss value $\mathcal{L}_{\text{Drift}}$ is constructed by combining three factors (Line 10). First, a drift loss between $\bar{d}$ and the real drift factor d^* is measured. Second, a correlation loss is also computed and adjusted by the hyperparameter λ_1 to preserve statistical dependencies in the base data, such as the strong primary-foreign key correlation between two tables. Third, a divergence loss between the synthesized and real drifted sample is included and adjusted by λ_2 to shift the distribution toward real drift. Finally, $\mathcal{L}_{\text{Drift}}$ is used to update `Drifter` via gradient descent.

4.4 Drift-aware Data and Workload Generation

We synthesize data and workload drift based on the diffuser and the drifter, as shown in Figure 4. After sampling an initial noise $\mathbf{x}^T \sim \mathcal{N}(0,\mathbf{I})$, the diffuser transforms $\mathbf{x}^T$ into drifted data $\tilde{\mathbf{x}_0}$ over T discrete timesteps. At each step t, the drifter computes a gradient $g_t = \nabla_{\mathbf{x}_t}\Pr(\mathbf{x}_{\text{drift}}|\mathbf{x}_t) = \nabla_{\mathbf{x}_t}\texttt{Drifter}(\mathbf{x}_t, t, d)$, which adjusts the predicted mean $\tilde{\boldsymbol{\mu}}_t = \boldsymbol{\mu} + \Sigma \cdot g_t$, and use the adjusted mean $\tilde{\boldsymbol{\mu}}_t$ to sample the next state $\mathbf{x}_{t-1}$. We repeat this process for timesteps $t = T, T-1, \ldots, 1$, gradually

refining $\mathbf{x}_t$ toward $\mathbf{x}_{\text{drift}}$. After decoding $\mathbf{x}_{\text{drift}}$, we obtain the data or workload that reflects the desired drift while maintaining the inherent correlations.

Correlation Preservation. NeurBench generates data as full tuples with multiple columns, thereby preserving inter-column correlations throughout the process. First, the diffuser learns the joint distribution of columns and captures their correlations in the latent space. Second, the drifter is trained using a composite loss (Algorithm 1, Line 10) that includes both a drift loss and a correlation preservation loss. As a result, during generation, the drifter injects drift signals that drift data distributions, while the diffuser ensures that column correlations are retained.

Snapshot-based Generation. We would like to note that using a single generator trained across a broad temporal span (*e.g.*, 2014–2017) may miss fine-grained drift patterns, especially for a small drift factor d. For example, if a user sets 2013 as the baseline and specifies $d = 0.05$ to simulate drift toward 2014, a generator trained on 2014–2017 data may capture overall trends but overlook subtle shifts between 2013 and 2014, due to smoothing effects from large-scale drift exposure. To alleviate this, we introduce an *agent-based* mechanism that selects the most appropriate generator for a target drift d^*. Specifically, we train multiple snapshot-specific generators $\mathcal{G}_i \in \{\mathcal{G}_{14\text{-}15}, \mathcal{G}_{15\text{-}17}, \dots\}$, where each $\mathcal{G}_i$ consists of a diffuser trained on data up to time i, and a drifter guided by subsequent data. For example, $\mathcal{G}$14-15 uses data up to 2014 for diffusion and captures drift patterns up to 2015. At runtime, the agent selects the best-matching generator by computing a distance metric $\text{dist}(d, \alpha_i)$ between the requested drift d and the representative drift α_i of each generator S_i. The agent then selects the generator $\mathcal{G}_{i^*}$ with the minimal distance: $\mathcal{G}_{i^*} = \arg\min_{\mathcal{G}_i} \text{dist}(d,\ \alpha_i)$.

5 Implementation

We have made the source code of NeurBench available [45], which includes a drift-aware generator and evaluation suites for learned database components. We implement the generator based on DDPM [25, 36]. In particular, the diffuser is based on a Multilayer Perceptron (MLP) with hidden layer dimensions of (512, 1024, 1024, 512) and the drifter is a similar MLP with hidden layer dimensions of (512, 512). We use tables to represent both data and workloads, as specified in Section 4.1, and encode the tables using the analog bit encoding [36]. For each table, we dynamically adjust the learning rate from $1e^{-4}$ to $2e^{-3}$ to ensure the convergence of the training process.

NeurBench provides user commands such as `gd` for drift-aware data generation and `tqo` for learned query optimizer evaluations. Users can easily reconfigure these settings to match their own environments and workloads for customized benchmarking. Since no existing database integrates all tested learned database components, we adopt and extend existing evaluation suites [30, 56, 59] to support benchmarking under data and workload drift. NeurBench is designed to be extensible to support additional evaluators.

NeurBench is functionally adequate for our use case by enabling the generation of both numerical and categorical attributes. However, as a diffusion-based method, NeurBench introduces training and inference overhead. Its generation efficiency can be constrained by available resources, and the long generation time required for TB-scale data is a potential limitation.

6 Experimental Setup

In this section, we specify the experimental setup.

6.1 Datasets and Workloads

We use two real-world datasets and generate synthetic datasets based on them. 1) **IMDB [21]** is a real-world dataset containing 21 tables with complex correlations. By default, we use the IMDB version collected by Leis et al. [31], containing real data up to 2013. In addition, we collected

snapshots from the IMDB archives [1], which contain IMDB data up to 2017. **STACK [40]** consists of questions and answers from StackExchange websites [51]. We split the table by year to construct multiple yearly snapshots.

We execute JOB queries [31] on IMDB, which consists of 113 queries derived from 33 real-world query templates, and execute another 113 queries sampled by [30] on STACK.

6.2 Evaluated Data Generators

We compare the drift-aware data and workload generation framework of NeurBench with two data generation techniques that can be extended to support drift data generation: 1) **RelDDPM [36]** is a state-of-the-art data synthesis framework that generates correlation-preserving tabular data using DDPM. RelDDPM originally supported only non-drift data generation. We adjust the strength of the guidance in the denoising process of RelDDPM to generate drifted data, and use it as our baseline. 2) **CTGAN [63]** is a widely used framework for modeling tabular data distributions via conditional GANs. As CTGAN lacks explicit drift control, we vary its hyperparameters to generate drifted datasets.

6.3 Evaluated Learned Components

We select representative learned database components, as listed in Table 1, to conduct our evaluations. We ensure the evaluated components cover as many key design choices as possible. Due to space constraints, we opt not to conduct exhaustive evaluations of all existing learned components.

6.3.1 End-to-end Learned Query Optimizer. Given a query $\mathbf{q}$, the optimizer first performs *plan search* to obtain a set of candidate query plans $\mathbf{P}(\mathbf{q}) = \{\mathbf{p}_0, \mathbf{p}_1, ..., \mathbf{p}_N\}$, and then select the best one from $\mathbf{P}(\mathbf{q})$ for real execution. During the plan search and selection, it interacts with an *ML model* to assess the plan quality. Existing approaches [40, 41, 64, 65, 71] typically repeat the above steps to construct an experience set, which is then used to guide learning.

Model. There are two main types of models for assessing plan quality: 1) *Regression model* predicts the execution cost or latency $\hat{L}(\mathbf{p})$ of a given plan $\mathbf{p}$, optionally providing a confidence score $C(\mathbf{p})$ that reflects the model's certainty, *i.e.*, $f_{qo}(\mathbf{p}) = (\hat{L}(\mathbf{p}), C(\mathbf{p}))$ [40, 41, 65]. 2) *Ranking model* predicts a binary ranking value r for two given plans $\mathbf{p}_1$ and $\mathbf{p}_2$, where $r = 1$ if $\mathbf{p}_1$ is expected to outperform $\mathbf{p}_2$, and $r = 0$ otherwise, *i.e.*, $f_{qo}(\mathbf{p}_1, \mathbf{p}_2) = r$, $r \in \{0, 1\}$ [71].

Plan Search. There are two paradigms in plan search, which differ in whether they rely on traditional query optimizers: 1) *White-box search* provides auxiliary information to traditional query optimizers for generating candidate plans. Based on the type of information provided, it can be further categorized into: (a) *Hint-based.* It supplies hints (parameters that control physical operators, such as join or scan types) to the traditional optimizer, guiding it to generate corresponding query plans [40, 65]. (b) *Cardinality-based.* It adjusts (by magnifying or reducing) estimated cardinalities multiple times to generate a variety of candidate plans [71]. 2) *Black-box search* embeds the model directly into the plan search process without relying on the traditional optimizers [41]. It starts with an empty or partial plan and iteratively constructs a complete query plan until a predefined number of candidate plans is reached. At each step, multiple subplans are generated. After using an ML model to evaluate their quality, ineffective subplans are pruned while the remaining ones are expanded in subsequent iterations.

6.3.2 Updatable Learned Index. Given a search key $\mathbf{k}$, the learned index f_{idx} predicts its position $\hat{PS}(\mathbf{k})$ in the data table, *i.e.*, $f_{idx}(\mathbf{k}) = \hat{PS}(\mathbf{k})$. Learned indexes organize a set of ML models in a layered structure, where each node contains a model for position prediction.

Table 1. A Taxonomy of Evaluated Learned Components

	Method	**Model**	**Plan Search**
End-to-end Learned Query Optimizer	Bao [40]	Regression	Hint-based
	Balsa [64]	Regression	Black-box
	Lero [71]	Ranking	Cardinality-based
	Method	**Structure**	**Index Insertion**
Updatable Learned Index	ALEX [12]	Heterogeneous	In-place
	LIPP [60]	Homogeneous	In-place
	XIndex [53]	Heterogeneous	Delta-buffer
Learned CC	PolyJuice [56]		

Structure. Existing learned indexes can be classified into two representative structures, based on whether they have different node types: 1) *Heterogeneous structure.* It consists of two node types: inner nodes and leaf nodes. Inner nodes do not store data but predict the next step in the index. Leaf nodes store the actual data, and a local search is performed to locate the exact key position. 2) *Homogeneous structure.* Each node is responsible for storing part of the data and also predicting whether the key falls within its range or should be passed to a corresponding child node.
Index Insertion. A learned index is regarded as updatable if it supports key insertions. There are two main key insertion mechanisms: 1) *In-place.* Index nodes reserve gaps (empty slots), and newly inserted keys are directly inserted into an appropriate node. 2) *Delta-buffer.* Dedicated buffers are maintained, and new keys are first inserted into the buffer and later merged into the main structure periodically.

6.3.3 Learned Concurrency Control (CC). Learned CC algorithm dynamically selects the most suitable CC actions, such as locking, optimistic validation, *etc.*, to gain higher transaction throughput. It takes a transaction operation op (*e.g.*, read/write) and the current system condition $C(op)$ (*e.g.*, contention levels) as inputs, and leverages an ML model f_{cc} to predict the most appropriate CC action for the operation op. Formally, let $\mathbf{A}$ be the set of available CC actions, f_{cc} can be defined as $f_{cc}(op, C(op)) = a$, where $a \in \mathbf{A}$.

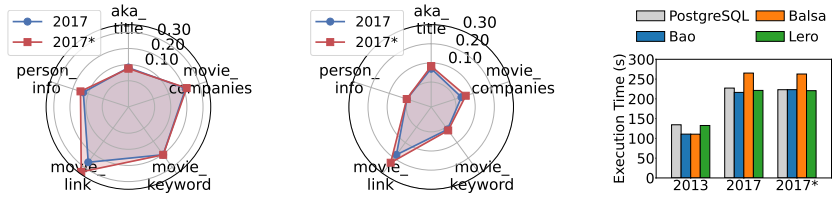
6.4 Evaluation Metrics

We assess the quality of synthesized drift by comparing its **drift factor** and **correlation difference (corr. diff.)** to those observed in real data. A drift factor closer to the real drift indicates more realistic distributional changes. Corr. diff., measured via Pearson correlation coefficient [58], quantifies how much attribute-level correlation deviates from the original data. Lower correlation difference, i.e., similar to that under real drift, suggests better correlation preservation. We define a good drift as one that achieves the target drift factor while preserving correlations as much as possible.

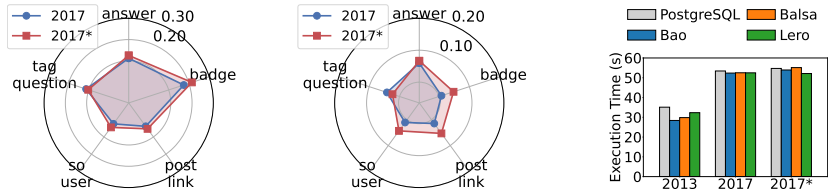
We use **execution time** to assess learned query optimizers, which is the actual time to execute a query. We do not include query planning time in the execution time. We use **throughput**, the number of operations/transactions processed per second, to evaluate learned indexes and learned CC.

6.5 Default Configuration

We conduct all experiments on a server with 2 Intel Xeon Silver 4214R @ 2.40GHz CPUs, a total of 24 cores and 48 threads, and 128 GB of memory. The server is equipped with 8 NVIDIA RTX 3090 GPUs. All experiments are run within Docker containers (Ubuntu 22.04 with CUDA 11.8.0).

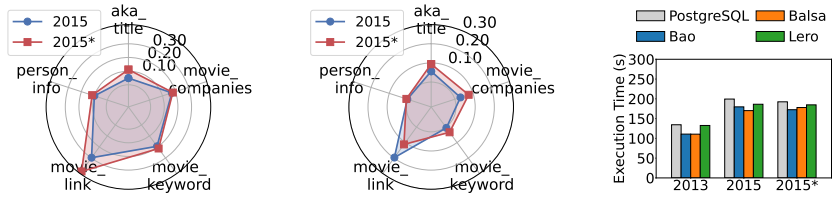


(a) Drift Factor (IMDB) (b) Corr. Diff. (IMDB) (c) Exec. Time (IMDB)

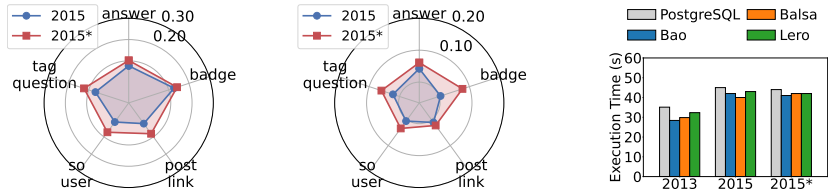


(d) Drift Factor (STACK) (e) Corr. Diff. (STACK) (f) Exec. Time (STACK)

Fig. 5. Extrapolation Performance of Drift-aware Generator



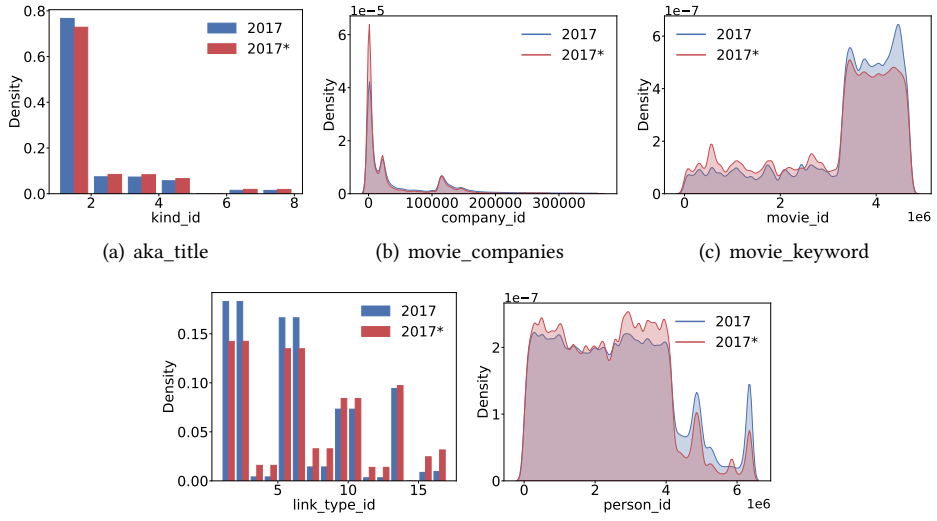
(a) Drift Factor (IMDB) (b) Corr. Diff. (IMDB) (c) Exec. Time (IMDB)



(d) Drift Factor (STACK) (e) Corr. Diff. (STACK) (f) Exec. Time (STACK)

Fig. 6. Interpolation Performance of Drift-aware Generator

Unless otherwise specified, we train the evaluated components on one dataset and workload, and test on other datasets and workloads to rigorously evaluate drift [44]. We randomly sample and repeatedly execute the queries provided in Section 6.1 to train the evaluated learned query optimizers, following current practices [30, 64]. By default, we execute 1500/33900/1500 queries to train Bao/Balsa/Lero, respectively. Note that Lero may execute additional queries during its exploratory process. We only report execution time in our experiments, while excluding planning time. We do not implement parallel planning for Bao, and use the planning code provided in the original Bao repository [40]. For data drift experiments, we compare the delta between the drifted data and the original data, and apply updates or inserts accordingly. The statistics are updated in PostgreSQL. We standardize the PostgreSQL version to v12.20 and apply consistent configurations to ensure fair comparisons. Following [30], we set the shard buffer size of PostgreSQL to 32GB. Consequently, the absolute performance levels of the evaluated learned query optimizers may differ from those reported in their original papers.



(a) aka_title (b) movie_companies (c) movie_keyword

(d) movie_link (e) person_info

Fig. 7. Column-level Distribution of Generated Data (IMDB)

7 Experimental Results

In this section, we first evaluate the effectiveness of NeurBench in generating realistic drift, and then conduct benchmarks on selected learned database components using NeurBench.

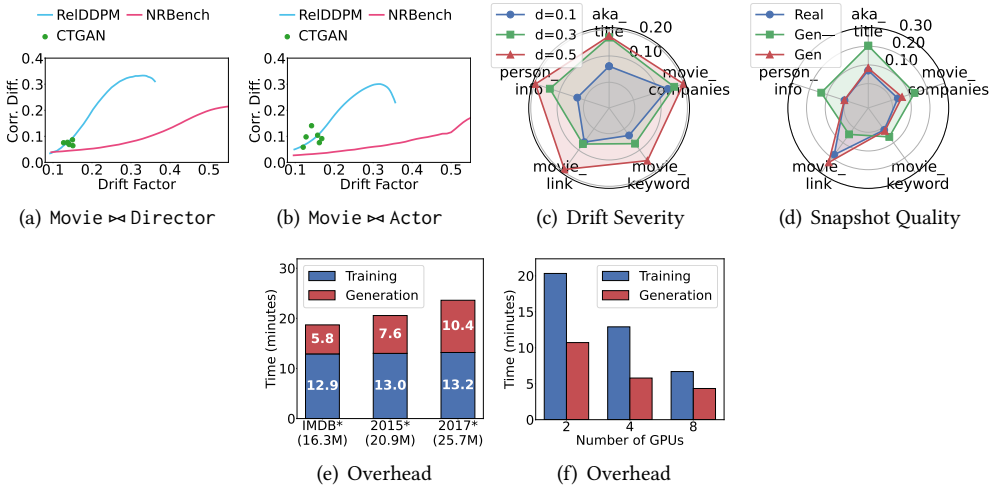
7.1 Experiments on Drift Generator

7.1.1 Experiments on the quality of synthesized data drift. We evaluate the effectiveness of NeurBench in drift data generation by comparing synthesized drift with real drift across two settings: extrapolation and interpolation. In the extrapolation setting, we train NeurBench using data from 2013–2016 and generate synthetic data for 2017 (denoted as 2017*). In the interpolation setting, we train using 2013–2017 excluding 2015, and generate 2015* to fill the gap. For illustrative purposes, we first analyze the results on the IMDB dataset, followed by those on the STACK dataset.

Drift Analysis. Figure 5(a) and Figure 6(a) show that the synthesized snapshots exhibit drift factors highly similar to real snapshots, confirming that the generated data induces realistic distributional drift. We further report the per-column distribution of 2017* in Figure 7, which shows that the generated data exhibits comparable distribution patterns and similar density to the real data.

Correlation Preservation. Figure 5(b) and Figure 6(b) report the correlation difference between generated data and real data. We observe that the generated snapshots maintain low correlation differences, comparable to those of real data, indicating that NeurBench effectively preserves attribute dependencies during generation.

Downstream Performance. To evaluate the realism of the generated drift from a workload perspective, we train learned query optimizers (Bao, Balsa, Lero) on IMDB and evaluate them on both real (e.g., 2017) and generated (e.g., 2017*) datasets. We report the accumulated query execution time in Figure 5(c) and Figure 6(c). We can observe that all evaluated systems exhibit performance degradation on both real and synthetic drifted data. This is expected, as learned query optimizers may suffer from suboptimal plan choices due to outdated knowledge about data distributions, which we will further evaluate in Section 7.2. Further, the degradation patterns are highly similar, indicating that the generated data successfully simulates the impact of real-world drift on query performance.



(a) Movie ⋈ Director (b) Movie ⋈ Actor (c) Drift Severity (d) Snapshot Quality

(e) Overhead (f) Overhead

Fig. 8. Overhead and In-depth Analysis

We also perform experiments in generating drifted STACK data. As shown in Figure 5(d) and Figure 5(e), NeurBench is able to synthesize drifted data with a relatively low correlation difference. In addition, Figure 5(f) demonstrates similar performance degradation for learned query optimizers under both synthesized and real drift.

Observation: NeurBench can synthesize data with realistic distribution drift and strong correlation fidelity, which in turn enables faithful downstream performance evaluation across multiple datasets.

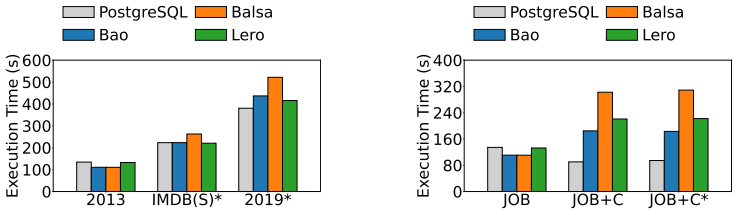
7.1.2 Experiments on generating drifted datasets under varying drift factors. We compare the data generation quality of NeurBench against RelDDPM and CTGAN. Following RelDDPM [36], we evaluate on joined IMDB tables and measure the correlation difference between the generated and real tables to quantify drift quality. Specifically, we construct two joined tables: 1) `Movie` ⋈ `Director`, where movies are linked to directors via director ID; and 2) `Movie` ⋈ `Actor`, where movies are linked to actors via actor ID. We plot the results of these two joint tables in Figure 8(a) and Figure 8(b), respectively. We can observe that NeurBench consistently achieves lower average correlation differences across all drift factors. RelDDPM fails to generate data for drift factors greater than 0.4, and the correlation difference fluctuates unpredictably, showing no clear relationship with the drift factor. CTGAN produces data with less consistent correlation control. Its conditional generation mechanism is not tailored for modeling data drift, making it difficult to produce a controlled spectrum of drifted distributions.

Observation: NeurBench effectively synthesizes data across a wide range of drift factors with relatively low correlation differences.

7.1.3 Generation Quality Analysis. We test the factors that can affect the data generation quality.

We first evaluate the effects of drift severity. We generate a series of synthetic datasets from IMDB with drift factors $d \in 0.1, 0.3, 0.5$, representing mild, medium, and severe drift. We keep the data size the same as the original IMDB. As shown in Figure 8(c), the correlation difference increases with drift factor, indicating that preserving correlations becomes more challenging under more severe drift. However, as shown in Figure 8(a) and Figure 8(b), the average correlation difference rises nearly linearly with increased drift. The results confirm that NeurBench can effectively synthesize drifted data while preserving correlations in a controlled manner.

We then evaluate the effects of snapshot quality. With the same setup as in Figure 5(b), we additionally include a variant (Gen–) that omits the 2016 snapshot, representing a case without



(a) Data Drift Generator (b) Workload Drift

Fig. 9. Extended Data Drift and Workload Drift Analysis

a high-quality snapshot. As shown in Figure 8(d), Gen– exhibits higher correlation differences compared to Gen, confirming that high-quality snapshots improve the correlation preservation ability.

Observation: The drift data generation quality can be affected by the drift severity and snapshot quality.

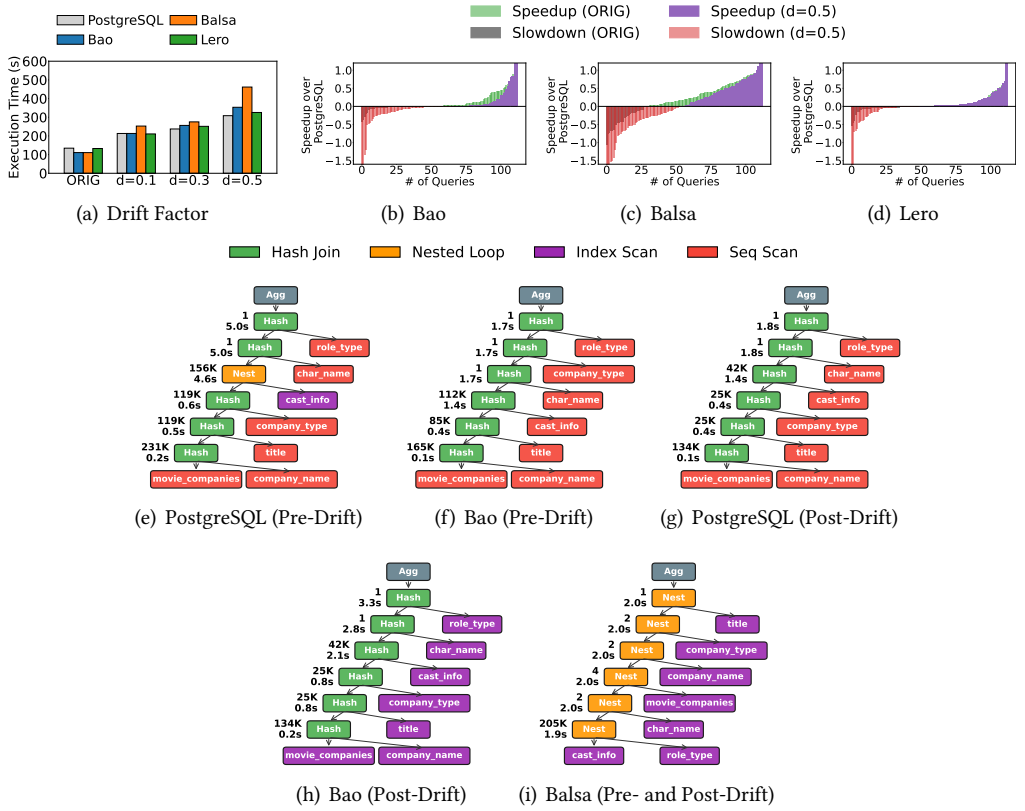
7.1.4 Overhead Analysis. We evaluate the time overhead incurred by drift generator training and drift data generation. We first study the overhead under varying data sizes. IMDB* refers to the IMDB dataset generated with different drift factors. As shown in Figure 8(e), generation time increases from 5.8 minutes on IMDB* (16.3M tuples) to 10.4 minutes on IMDB 2017* (25.7M tuples). This is due to the fact that the generator produces one batch of tuples at a time, and as the dataset grows, more batches are required, leading to a near-linear increase in generation time. In contrast, the training time remains stable (~13 minutes), as it depends solely on the number of training steps, which we fix to 500 across all experiments. We then evaluate the overhead under varying GPU resources, and plot the results in Figure 8(f). As observed, both training and generation times decrease as the number of GPUs increases from 2 to 8. This is expected, as batch-based generation can be effectively parallelized across multiple GPUs, demonstrating good scalability of our approach.

Observation: The factors that determine the generation overhead are data sizes and GPU resources.

7.1.5 Data Drift Generator Extensibility. We study whether a drifter trained on one dataset can be transferred to another dataset. Specifically, we apply a drifter trained on the STACK dataset to an IMDB diffuser to generate drifted IMDB data, denoted as IMDB(S)*. As shown in Figure 9(a), this transfer is feasible because the drifter predicts noise at each denoising step of the diffuser. Since the diffusion process starts from a Gaussian distribution, injecting such noise still induces meaningful distributional changes on the target dataset. We further examine whether IMDB data can be extrapolated to approximate a snapshot with incomplete information. We collect partial IMDB data for 2019 from [21] and use it as a snapshot to train the data generator, synthesizing 2019*. As shown in Figure 9(a), 2019* leads to more severe performance degradation than 2017*, indicating that NeurBench can synthesize stronger drift over a larger temporal span.

Observation: A drifter trained on one dataset can be transferred to another dataset to enable drift generation.

7.1.6 Experiments on the quality of synthesized workload drift. We validate the effectiveness of NeurBench in generating realistic workload drift. Following Krid et al.[14], we adopt JOB+C, a hybrid workload composed of 46 queries from JOB and 66 from CEB [43], constructed based on workload pattern mappings derived from real Redshift traces [55]. We use NeurBench to synthesize JOB+C*, aiming to mimic the drift from JOB to JOB+C. The workload generator of NeurBench is trained with a query set collected from [13, 43, 65], excluding those used in JOB+C. We fix the



(a) Drift Factor (b) Bao (c) Balsa (d) Lero

(e) PostgreSQL (Pre-Drift) (f) Bao (Pre-Drift) (g) PostgreSQL (Post-Drift)

(h) Bao (Post-Drift) (i) Balsa (Pre- and Post-Drift)

Fig. 10. Evaluation of Learned Query Optimizers under Synthetic Data Drift with Various Drift Factors

dataset to the default IMDB, train each system on JOB, and test on both JOB+C and JOB+C*. As shown in Figure 9(b), the performance on both workloads is highly comparable, indicating that NeurBench effectively captures realistic workload drift. Moreover, the effectiveness of learned query optimizers reduces under drifted workloads, which we will further investigate in Section 7.4.1.

7.2 Experiments on Learned Query Optimizers under Synthetic Data Drift

In this set of experiments, we evaluate the selected learned query optimizers under the drift synthesized by NeurBench.

7.2.1 Experiments under synthesized data drift with various drift factors. We construct synthesized datasets with gradually increasing drift factors to evaluate the impact of data drift. By default, we use the drifted datasets generated in Section 7.1.3. We train each system on the original IMDB (denoted as ORIG) and evaluate on datasets with increasing d. As shown in Figure 10(a), the execution time increases with larger drift values. This aligns with our expectation that models trained on less similar data distributions experience more severe staleness.

To better understand the performance degradation caused by data drift (Figure 10(a)), we compute model accuracy metrics and report the results in Table 2. We use the median Q-error to measure the ratio between the estimated and true execution time [19, 22]. Top-3 AUC measures how often the best plan appears in the top 3 results ranked by the model. We observe that learned query optimizers generally suffer from reduced model accuracy under data drift, which aligns with their overall performance degradation. Among them, Balsa shows the most significant decline, as it

Table 2. Model Accuracy of Learned Query Optimizers

Competitors	Metrics	ORIG	d=0.1	d=0.3	d=0.5
Bao	Median Q-error	1.184	1.489	2.057	2.671
Balsa		1.723	1.974	2.156	3.512
Lero	Top-3 AUC	0.646	0.478	0.407	0.389

follows a black-box design that fully depends on the model's predictions to generate query plans. In contrast, white-box approaches such as Bao embed learned hints into the optimizer, and therefore, they still partially leverage the DBMS's native cardinality estimates.

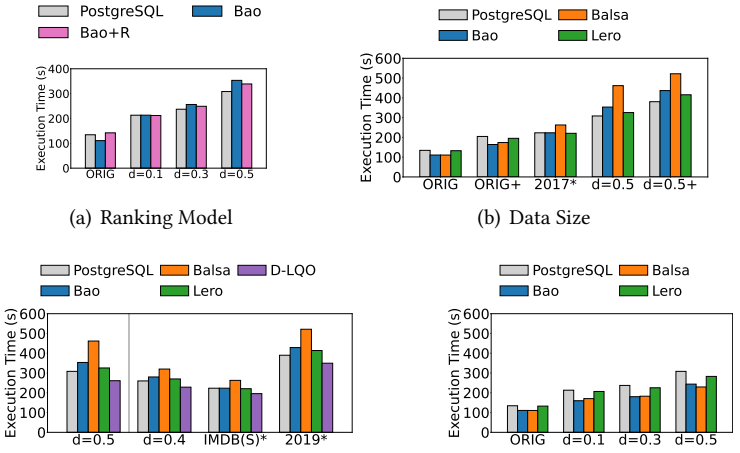
To further validate this, we analyze the speedup of query execution times under data drift, and plot the speedup in Figure 10(b), Figure 10(c) and Figure 10(d). We observe that Balsa exhibits much higher variance in speedup than Bao, confirming its greater sensitivity to drift. Interestingly, for certain queries, Balsa even outperforms PostgreSQL by a larger margin, suggesting that fine-grained black-box models can still identify better plans when they generalize well. We take a closer look at the plans generated by each system for a representative query, q10c. Before the drift, as shown in Figure 10(e) and Figure 10(f), Bao improves performance by applying the hint 'SET enable_indexscan TO off'. However, after the drift (Figure 10(g) and Figure 10(h)), Bao switches to disabling nested loop joins with 'SET enable_nestloop TO off'. Despite this change, the join order selected by Bao remains largely similar to that of PostgreSQL because Bao relies on PostgreSQL's initial plan structure. In contrast, Balsa directly predicts join orders via its model, which allows it to produce plans that differ significantly from both PostgreSQL and Bao. Notably, as shown in Figure 10(i), Balsa generates the same plan before and after the drift, suggesting that the model failed to adapt to distributional changes. This static behavior under drift likely leads to mispredicted plans and performance degradation.

Observation: White-box approaches can achieve better robustness. Black-box approaches, with larger search spaces, offer greater potential robustness but are often limited by current model capacity.

7.2.2 Regression model vs. Ranking model. We further study the effect of the regression model and the ranking model on the performance of white-box approaches. We implement Bao+R, a variant of Bao that employs Lero's ranking model while retaining Bao's hint-based plan search strategy. Therefore, comparing the performance degradation of Bao+R and Bao can directly evaluate the robustness of the ranking model. As shown in Figure 11(a), Bao+R suffers less performance degradation than Bao, suggesting that the ranking model improves robustness under drift. However, because the ranking model does not explicitly estimate predicate costs, its performance under non-drift scenarios is less competitive compared to Bao's regression-based model.

Observation: Ranking-based models tend to be slightly more robust under data drift but may struggle to achieve optimal performance in stable settings.

7.2.3 Experiments under synthesized data drift with varying data sizes. We also evaluate the individual effects of data distribution drift and dataset size on learned query optimizers. We construct ORIG+ by increasing the dataset size of ORIG to match that of 2017, while minimizing distribution drift. Comparing ORIG+ to ORIG isolates the effect of data size, while comparing 2017* to ORIG+ indicates the impact of distribution drift alone. As shown in Figure 11(b), both data scale and distribution drift affect the performance of learned query optimizers. In particular, under pure distribution drift (*i.e.*, d=0.5), we observe clear performance degradation, indicating that learned optimizers are sensitive to distributional changes. We further introduce d=0.5+, a variant of drifted IMDB data d=0.5 with the data size matching 2017*. Compared to the standard distribution drift



(a) Ranking Model (b) Data Size

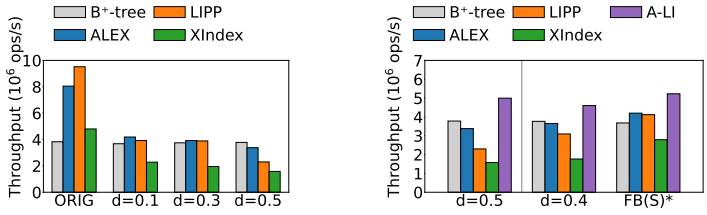
(c) Comparison with D-LQO (d) Periodical Retraining

Fig. 11. Analysis of Learned Query Optimizers under Various Synthetic Data Drift

d=0.5, d=0.5+ leads to more pronounced performance degradation, suggesting that distributional drift is the dominant factor and its negative effects can be amplified by larger data sizes.
Observation: While data size contributes to performance variance, distribution drift more directly affects the models' robustness.

7.2.4 Experiments for learned query optimizers with dynamic structures. The results presented thus far indicate that both the model type and the plan search strategy have an effect on learned query optimizer robustness. Therefore, we investigate the possibility of a learned query optimizer that is allowed to dynamically select a model based on the system conditions. We implement a prototype of a dynamic learned query optimizer, D-LQO, which uses a coarse-grained router to select the optimizer (trained on non-drift data) expected to yield the lowest execution time for a given query under a specific drift. The router is trained using query encodings as inputs and their respective execution times under different drift levels (d=0.1, 0.3, 0.5) in Figure 10(a), as supervision. As shown in Figure 11(c), D-LQO achieves an 15.01% performance improvement over PostgreSQL without retraining under d=0.5. To further study the robustness, we evaluate D-LQO on three unseen drift scenarios, d=0.4, IMDB(S)*, and 2019*, as specified in Section 7.1.5. As observed in Figure 11(c), D-LQO still maintains up to 12.13% improvement in these cases. This robustness stems from the fact that learned query optimizers respond differently to drift. By learning these patterns, D-LQO selects the best-performing optimizer under a given drift.
Observation: A learned query optimizer with dynamic structures can deliver robustness without retraining.

7.2.5 Performance with periodical retraining. We evaluate the effectiveness of periodic retraining in learned query optimizers under data drift. In this setting, we simulate an online deployment by continuously injecting drifted data and retraining the model after a fixed number of query executions. As shown in Figure 11(d), periodic retraining helps the learned optimizers to regain their performance advantage over PostgreSQL. This stems from the model's gradual adaptation to the evolving data distribution. As the system executes queries on drifted data, it accumulates additional knowledge reflective of the new distribution, which is then leveraged during periodic retraining to restore performance.



(a) Drift Factor (b) Comparison with A-LI

Fig. 12. Throughput of Updatable Learned Indexes under Synthetic Data Drift

Table 3. Detailed Metrics of Updatable Learned Indexes

Competitors	Detailed Metrics	ORIG	d=0.1	d=0.3	d=0.5
B$^+$-tree		3.467	3.507	3.630	3.652
ALEX	Avg Search Bound	3.428	4.090	4.126	4.222
XIndex		4.556	5.179	11.620	12.842
LIPP	Avg Search Depth	2.279	2.837	2.976	3.229

Observation: When drift is smoothly injected and resources are sufficient, retraining is a viable strategy to enhance robustness, as its cost can be amortized across executions.

7.3 Experiments on Updatable Learned Indexes under Synthetic Data Drift

We study the performance of updatable learned indexes with varying synthesized data drift. Since all the evaluated learned indexes are in-memory, we include the in-memory B$^+$-tree [59] as a baseline. We use the Facebook [16] dataset to conduct our experiments. Based on it, we generate an original dataset (ORIG) and three drifted versions with drift factors d= 0.1, 0.3, and 0.5. Each drifted dataset is kept the same size as ORIG. We initialize each learned index using ORIG, and then incrementally apply insertions and deletions to introduce data drift. After applying drift, we perform point lookups to evaluate performance under different drift levels.

7.3.1 Experiments under synthesized data drift with various drift factors. We plot the overall throughput in Figure 12. As data drift increases, all learned indexes exhibit degraded performance, while the B$^+$-tree maintains relatively stable throughput. This is due to the fact that learned indexes rely on trained models to guide key placements, but the model accuracy deteriorates as the data distribution changes, leading to higher search costs. ALEX and LIPP insert data in-place, which leads to higher key lookup overhead as the data distribution evolves. XIndex, though designed to handle updates via a separate buffer, suffers from increased buffer size under drift, leading to additional lookup overhead. In contrast, B$^+$-tree does not rely on data distribution modeling; its structural properties remain effective regardless of drift, thus preserving stable search bounds and throughput. Among the learned indexes, LIPP experiences the most significant performance drop due to its chaining-based conflict resolution mechanism, which triggers frequent node splits. These splits deepen the tree structure, further increasing search cost. To validate these observations, Table 3 reports the search bounds on leaf nodes. A higher search bound and tree height indicate reduced model accuracy. We observe that the search bounds of ALEX, LIPP, and XIndex increase consistently with the degree of drift, aligning with their respective throughput degradation trends.
Observation: For learned indexes, heterogeneous structures generally offer greater stability under drift compared to homogeneous ones, but may fail to achieve optimal performance in non-drift or well-sampled scenarios.

7.3.2 Experiments for learned index with dynamic structures. To investigate how to enhance the robustness of learned indexes, we build A-LI, a learned index where each node independently adopts different design choices, including node size and split strategy. We employ a model to predict the optimal configuration parameters for a given drift level. We execute point lookups on data with drift factors (d=0.1,0.3,0.5) using various parameter settings and collect the resulting throughout as supervision signals to train the model. In our experiments, we collect 1000 samples. Apart from d=0.5, we evaluate the performance of A-LI under d=0.4 and FB(S)*, a variant of the Facebook dataset generated using the STACK drifter. As shown in Figure 12(b), A-LI achieves up to 32.18% higher throughput than the next-best learned index under d=0.5, and up to 24.45% improvement under unseen drifts. A-LI adapts its structure by applying different split strategies and node sizes across nodes according to local drift characteristics. In particular, nodes experiencing stronger drift are assigned larger node sizes, which helps reduce search error and maintain high performance. Under d=0.5, A-LI achieves an average search error of 3.037, lower than all evaluated baselines. These results demonstrate that a learned index with an adaptive structure can remain efficient under drift. Recent work, such as SELIX [18], further explores this direction.
Observation: Designing adaptive learned indexes that employ different structures for different nodes or subtrees can be a promising direction to improve robustness.

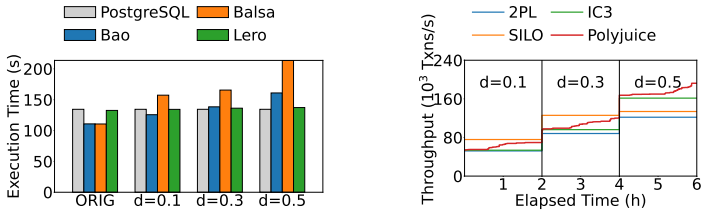
7.4 Experiments under Workload Drift

7.4.1 Experiments on learned query optimizer. We apply different drift factors ($d \in 0.1, 0.3, 0.5$) to the join patterns. We use the generated drifted workloads to train learned query optimizers, and evaluate them on the original JOB workload, ensuring the performance can be compared between different drifts. As observed in Figure 13(a), Balsa exhibits the most pronounced performance degradation, due to its reliance on a black-box plan enumeration process. When training join patterns no longer align with the test workload, Balsa's model produces inaccurate plan predictions, resulting in suboptimal performance. This is consistent with our finding in Section 7.2.1.

7.4.2 Evaluation on learned CC. We test Polyjuice [56], with two traditional algorithms, 2PL and OCC [62], and a hybrid CC algorithm, IC3 [57] under varying workload drift. We use an initial arrival rate simulated with a single client thread, and increase the drift factors to simulate transaction arrival rates. Polyjuice is first trained on the initial arrival rate and then retrained after each arrival rate drift, following the original retraining procedure. We execute the default TPC-C [9] transactions and plot the throughput in Figure 13(b). As drift increases, Polyjuice achieves higher throughput due to the increased arrival rate. However, it requires more time to converge to an efficient policy under higher drift, as greater drift increases learning complexity due to limited prior knowledge In addition, Polyjuice's retraining is time-consuming, hindering its responsiveness to continuous drift in arrival rates. These results demonstrate that learned CC still has room for improvement in fast adaptation, as discussed in recent work [47].
Observation: Fast adaptation is crucial for learned CC to be practical in real-world scenarios, since transactions can be completed in milliseconds or less.

8 Related Work

Data Generators. Existing data generators for tabular data typically focus on modeling the distribution of a static table [25, 49, 67]. To improve flexibility, some recent approaches [36, 63] propose conditional generation models that support data synthesis under given schema constraints or workload patterns. However, these methods do not explicitly incorporate distribution drift into the generation process. As a result, their conditional mechanisms are not designed to support effective generation of evolving data distributions. In contrast, the drift generator in NeurBench



(a) Learned Query Optimizer (b) Learned CC

Fig. 13. Evaluations under Workload Drift

explicitly models distributional changes, enabling the synthesis of data with controlled drift. This further enables fine-grained evaluation of learned database components under evolving drift conditions.

Benchmarks for Learned Database Components. Multiple benchmarks have been devised for various learned database components [6, 19, 29, 38, 42, 52, 68]. Among them, several recent studies have attempted to benchmark learned database components under data and workload drift [22, 30, 44, 52, 59]. These efforts typically treat drift in a case-by-case manner, and therefore only partially reveal how different types and degrees of drift affect system performance. NeurBench, in contrast, provides controllable and effective drift-aware data and workload generation, enabling a more comprehensive performance evaluation under diverse drift scenarios. As more learned database components emerge, such as cardinality estimators based on more generalizable models [23, 32, 70], and as end-to-end learned DBMSs begin to integrate these components [46], NeurBench serves as a valid and general benchmark suite to support the continued improvement of their robustness.

Benchmarks with Distribution Drift. Several benchmarks have been proposed to evaluate performance under distribution drift. DSB [11], a standard benchmark for decision-making applications, enables evaluations under distribution drift using TPC-DS. However, it is restricted to exponential distributions and does not quantitatively measure drift as NeurBench. In addition, concept drift modeling [4, 5, 15, 24, 33, 37, 48] has gained traction for evaluating ML-based prediction and analysis tasks. Although they are not directly applicable to database benchmarking, these methods also treat drift as distribution drift. Emerging dynamic benchmarks [3, 15] show promising potential. As a dynamic benchmark, NeurBench introduces measurable and controllable data and workload drift based on real drifts to enable systematic performance evaluations.

9 Conclusion

This paper introduced NeurBench, a new benchmark suite designed to evaluate end-to-end learned DBMSs containing all learned components under controllable data and workload drift. We defined the drift factor to effectively quantify drift and introduced a novel drift-aware data and workload generation framework. Extensive experimental results show the effectiveness of NeurBench in generating realistic drift, and demonstrate that different learned database components withstand and adapt to data and workload drift to varying degrees, depending on their specific design choices. Through in-depth analysis, we offer recommendations for enhancing the robustness of learned database components.

Acknowledgments

We would like to thank the anonymous reviewers and meta-reviewer for their constructive comments. This work is partially supported by NUS Faculty Development Fund. Meihui Zhang's work is supported by National Natural Science Foundation of China (U2441237). Gang Chen's work is supported by the National Regional Innovation and Development Joint Fund (No. U24A20254).

References

[1] IMDB Archive. 2025. *IMDB Archive.* https://ftp.fu-berlin.de/pub/misc/movies/database/frozendata/diffs/
[2] Shai Ben-David, John Blitzer, Koby Crammer, Alex Kulesza, Fernando Pereira, and Jennifer Wortman Vaughan. 2010. A theory of learning from different domains. *Mach. Learn.* 79, 1-2 (2010), 151–175.
[3] Lawrence Benson, Carsten Binnig, Jan-Micha Bodensohn, Federico Lorenzi, Jigao Luo, Danica Porobic, Tilmann Rabl, Anupam Sanghi, Russell Sears, Pinar Tözün, and Tobias Ziegler. 2024. Surprise Benchmarking: The Why, What, and How. In *DBTest@SIGMOD*. ACM, 1–8.
[4] Maximilian Böther, Ties Robroek, Viktor Gsteiger, Robin Holzinger, Xianzhe Ma, Pinar Tözün, and Ana Klimovic. 2025. Modyn: Data-Centric Machine Learning Pipeline Orchestration. *Proc. ACM Manag. Data* 3, 1 (2025), 55:1–55:30.
[5] Maximilian Böther, Foteini Strati, Viktor Gsteiger, and Ana Klimovic. 2023. Towards A Platform and Benchmark Suite for Model Training on Dynamic Datasets. In *EuroMLSys@EuroSys*. ACM, 8–17.
[6] Yannis Chronis, Yawen Wang, Yu Gan, Sami Abu-El-Haija, Chelsea Lin, Carsten Binnig, and Fatma Özcan. 2024. CardBench: A Benchmark for Learned Cardinality Estimation in Relational Databases. *CoRR* abs/2408.16170 (2024).
[7] Cinemagoer. 2025. *Old IMDB Data Files.* https://cinemagoer.readthedocs.io/en/latest/usage/ptdf.html
[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. ACM, 143–154.
[9] The Transaction Processing Council. 2024. *TPC-C.* http://www.tpc.org/tpcc/
[10] Prafulla Dhariwal and Alexander Quinn Nichol. 2021. Diffusion Models Beat GANs on Image Synthesis. In *NeurIPS*. 8780–8794.
[11] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek R. Narasayya. 2021. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. *Proc. VLDB Endow.* 14, 13 (2021), 3376–3388.
[12] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD Conference*. ACM, 969–984.
[13] Andreas Kipf et al. 2025. *JOB-light Worklods.* https://github.com/andreaskipf/learnedcardinalities/tree/master/workloads
[14] Skander Krid et al. 2025. *Redbench Implementation.* https://github.com/utndatasystems/redbench
[15] Josh Gardner, Zoran Popovic, and Ludwig Schmidt. 2023. Benchmarking Distribution Shift in Tabular Data with TableShift. In *NeurIPS*.
[16] Minas Gjoka, Maciej Kurant, Carter T. Butts, and Athina Markopoulou. 2010. Walking in Facebook: A Case Study of Unbiased Sampling of OSNs. In *INFOCOM*. IEEE, 2498–2506.
[17] Ishaan Gulrajani and David Lopez-Paz. 2021. In Search of Lost Domain Generalization. In *ICLR*.
[18] Baofu Han, Guoyu Hu, Bing Li, Xiaokui Xiao, Zhanhao Zhao, and Beng Chin Ooi. 2026. On Self-Designing Learned Indexes. *Proc. ACM Manag. Data* 4, 2 (2026), 219:1–219:25.
[19] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *Proc. VLDB Endow.* 15, 4 (2021), 752–765.
[20] Jonathan Ho, Ajay Jain, and Pieter Abbeel. 2020. Denoising Diffusion Probabilistic Models. In *NeurIPS*.
[21] IMDB. 2025. *IMDB Dataset.* https://www.imdb.com/
[22] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. 2022. Learned Cardinality Estimation: An In-depth Study. In *SIGMOD Conference*. ACM, 1214–1227.
[23] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*. www.cidrdb.org.
[24] Pang Wei Koh, Shiori Sagawa, Henrik Marklund, Sang Michael Xie, Marvin Zhang, Akshay Balsubramani, Weihua Hu, Michihiro Yasunaga, Richard Lanas Phillips, Irena Gao, Tony Lee, Etienne David, Ian Stavness, Wei Guo, Berton Earnshaw, Imran S. Haque, Sara M. Beery, Jure Leskovec, Anshul Kundaje, Emma Pierson, Sergey Levine, Chelsea Finn, and Percy Liang. 2021. WILDS: A Benchmark of in-the-Wild Distribution Shifts. In *ICML*, Vol. 139. 5637–5664.
[25] Akim Kotelnikov, Dmitry Baranchuk, Ivan Rubachev, and Artem Babenko. 2023. TabDDPM: Modelling Tabular Data with Diffusion Models. In *ICML*, Vol. 202. 17564–17579.
[26] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD Conference*. ACM, 489–504.
[27] Meghdad Kurmanji, Eleni Triantafillou, and Peter Triantafillou. 2024. Machine Unlearning in Learned Databases: An Experimental Analysis. *Proc. ACM Manag. Data* 2, 1 (2024), 49:1–49:26.
[28] Meghdad Kurmanji and Peter Triantafillou. 2023. Detect, Distill and Update: Learned DB Systems Facing Out of Distribution Data. *Proc. ACM Manag. Data* 1, 1 (2023), 33:1–33:27.
[29] Hai Lan, Zhifeng Bao, J. Shane Culpepper, and Renata Borovica-Gajic. 2023. Updatable Learned Indexes Meet Disk-Resident DBMS - From Evaluations to Design Choices. *Proc. ACM Manag. Data* 1, 2 (2023), 139:1–139:22.

[30] Claude Lehmann, Pavel Sulimov, and Kurt Stockinger. 2024. Is Your Learned Query Optimizer Behaving As You Expect? A Machine Learning Perspective. *Proc. VLDB Endow.* 17, 7 (2024), 1565–1577.

[31] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.

[32] Pengfei Li, Wenqing Wei, Rong Zhu, Bolin Ding, Jingren Zhou, and Hua Lu. 2023. ALECE: An Attention-based Learned Cardinality Estimator for SPJ Queries on Dynamic Workloads. *Proc. VLDB Endow.* 17, 2 (2023), 197–210.

[33] Wendi Li, Xiao Yang, Weiqing Liu, Yingce Xia, and Jiang Bian. 2022. DDG-DA: Data Distribution Generation for Predictable Concept Drift Adaptation. In *AAAI*. AAAI Press, 4092–4100.

[34] Yu-Shan Lin, Ching Tsai, Tz-Yu Lin, Yun-Sheng Chang, and Shan-Hung Wu. 2021. Don't Look Back, Look into the Future: Prescient Data Partitioning and Migration for Deterministic Database Systems. In *SIGMOD Conference*. ACM, 1156–1168.

[35] Haohe Liu, Zehua Chen, Yi Yuan, Xinhao Mei, Xubo Liu, Danilo P. Mandic, Wenwu Wang, and Mark D. Plumbley. 2023. AudioLDM: Text-to-Audio Generation with Latent Diffusion Models. In *ICML*, Vol. 202. 21450–21474.

[36] Tongyu Liu, Ju Fan, Nan Tang, Guoliang Li, and Xiaoyong Du. 2024. Controllable Tabular Data Synthesis Using Diffusion Models. *Proc. ACM Manag. Data* 2, 1 (2024), 28:1–28:29.

[37] Yuanhui Luo, Minhui Xie, Yiheng Tong, Shichao Jiang, and Yunpeng Chai. 2025. Understanding Robustness Issues of Updatable Learned Indexes: [Experiments & Analysis]. *Proc. ACM Manag. Data* 3, 4 (2025), 270:1–270:25.

[38] Chaohong Ma, Xiaohui Yu, Yifan Li, Xiaofeng Meng, and Aishan Maoliniyazi. 2022. FILM: a Fully Learned Index for Larger-than-Memory Databases. *Proc. VLDB Endow.* 16, 3 (2022), 561–573.

[39] Christopher D. Manning and Hinrich Schütze. 2001. *Foundations of statistical natural language processing*. MIT Press.

[40] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *SIGMOD Conference*. ACM, 1275–1288.

[41] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (2019), 1705–1718.

[42] Songsong Mo, Yile Chen, Hao Wang, Gao Cong, and Zhifeng Bao. 2023. Lemo: A Cache-Enhanced Learned Optimizer for Concurrent Queries. *Proc. ACM Manag. Data* 1, 4 (2023), 247:1–247:26.

[43] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proc. VLDB Endow.* 14, 11 (2021), 2019–2032.

[44] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust Query Driven Cardinality Estimation under Changing Workloads. *Proc. VLDB Endow.* 16, 6 (2023), 1520–1533.

[45] NeurBench. 2026. *NeurBench Implementation*. https://github.com/neurdb/neurbench

[46] Beng Chin Ooi, Shaofeng Cai, Gang Chen, Yanyan Shen, Kian-Lee Tan, Yuncheng Wu, Xiaokui Xiao, Naili Xing, Cong Yue, Lingze Zeng, Meihui Zhang, and Zhanhao Zhao. 2024. NeurDB: An AI-powered Autonomous Data System. *SCIENCE CHINA Information Sciences* 67, 10 (2024). doi:10.1007/s11432-024-4125-9

[47] Hexiang Pan, Shaofeng Cai, Yuncheng Wu, Yeow Meng Chee, Tien Tuan Anh Dinh, Gang Chen, and Beng Chin Ooi. 2026. Modeling Concurrency Control as a Learnable Function. *Proc. ACM Manag. Data* 4, 2 (2026), 211:1–211:28.

[48] Piotr Porwik and Benjamin Mensah Dadzie. 2022. Detection of data drift in a two-dimensional stream using the Kolmogorov-Smirnov test. In *KES (Procedia Computer Science, Vol. 207)*. Elsevier, 168–175.

[49] Maria F. Davila R., Sven Groen, Fabian Panse, and Wolfram Wingerath. 2024. Navigating Tabular Data Synthesis Research Understanding User Needs and Tool Capabilities. *SIGMOD Rec.* 53, 4 (2024), 18–35.

[50] Jascha Sohl-Dickstein, Eric A. Weiss, Niru Maheswaranathan, and Surya Ganguli. 2015. Deep Unsupervised Learning using Nonequilibrium Thermodynamics. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015 (JMLR Workshop and Conference Proceedings, Vol. 37)*, Francis R. Bach and David M. Blei (Eds.). JMLR.org, 2256–2265.

[51] StackOverflow. 2025. *StackOverflow*. https://stackoverflow.com

[52] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned Index: A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 16, 8 (2023), 1992–2004.

[53] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: a scalable learned index for multicore data storage. In *PPoPP*. ACM, 308–320.

[54] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proc. VLDB Endow.* 17, 11 (2024), 3694–3706.

[55] Alexander van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. *Proc. VLDB Endow.* 16, 6 (2023), 1413–1425.

[56] Jia-Chen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. 2021. Polyjuice: High-Performance Transactions via Learned Concurrency Control. In *OSDI*. USENIX Association, 198–216.

[57] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling Multicore Databases via Constrained Parallel Execution. In *SIGMOD Conference*. ACM, 1643–1658.
[58] Wikipedia. 2025. *Pearson Correlation Coefficient*. https://en.wikipedia.org/wiki/Pearson_correlation_coefficient
[59] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are Updatable Learned Indexes Ready? *Proc. VLDB Endow.* 15, 11 (2022), 3004–3017.
[60] Jiacheng Wu, Yong Zhang, Shimin Chen, Yu Chen, Jin Wang, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *Proc. VLDB Endow.* 14, 8 (2021), 1276–1288.
[61] Peizhi Wu and Zachary G. Ives. 2024. Modeling Shifting Workloads for Learned Database Systems. *Proc. ACM Manag. Data* 2, 1 (2024), 38:1–38:27.
[62] Yu Xia, Xiangyao Yu, Matthew Butrovich, Andrew Pavlo, and Srinivas Devadas. 2022. Litmus: Towards a Practical Database Management System with Verifiable ACID Properties and Transaction Correctness. In *SIGMOD Conference*. ACM, 1478–1492.
[63] Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. 2019. Modeling Tabular data using Conditional GAN. In *NeurIPS*. 7333–7343.
[64] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *SIGMOD Conference*. ACM, 931–944.
[65] Xiang Yu, Chengliang Chai, Guoliang Li, and Jiabin Liu. 2022. Cost-based or Learning-based? A Hybrid Query Optimizer for Query Plan Selection. *Proc. VLDB Endow.* 15, 13 (2022), 3924–3936.
[66] Sepanta Zeighami and Cyrus Shahabi. 2024. Theoretical Analysis of Learned Database Operations under Distribution Shift through Distribution Learnability. In *ICML*.
[67] Hengrui Zhang, Jiani Zhang, Zhengyuan Shen, Balasubramaniam Srinivasan, Xiao Qin, Christos Faloutsos, Huzefa Rangwala, and George Karypis. 2024. Mixed-Type Tabular Data Synthesis with Score-based Diffusion in Latent Space. In *ICLR*.
[68] Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, Guoan Wu, and Andy Rudoff. 2022. PLIN: A Persistent Learned Index for Non-Volatile Memory with High Performance and Instant Recovery. *Proc. VLDB Endow.* 16, 2 (2022), 243–255.
[69] Zhanhao Zhao, Shaofeng Cai, Haotian Gao, Hexiang Pan, Siqi Xiang, Naili Xing, Gang Chen, Beng Chin Ooi, Yanyan Shen, Yuncheng Wu, and Meihui Zhang. 2025. NeurDB: On the Design and Implementation of an AI-powered Autonomous Database. In *CIDR*. www.cidrdb.org.
[70] Jiaqi Zhu, Shaofeng Cai, Yanyan Shen, Gang Chen, Fang Deng, and Beng Chin Ooi. 2025. In-Context Adaptation to Concept Drift for Learned Database Operations. *ICML* (2025).
[71] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. *Proc. VLDB Endow.* 16, 6 (2023), 1466–1479.

Received October 2025; revised January 2026; accepted February 2026