Toward Drift-Aware Database Benchmarking

Guanli Liu
The University of Melbourne
guanli.liu1@unimelb.edu.au

Renata Borovica-Gajic
The University of Melbourne
renata.borovica@unimelb.edu.au

ABSTRACT

Data and workload drift are critical to evaluating core database components such as caching, cardinality estimation, indexing, and query optimization, especially as AI-driven techniques increasingly permeate database systems. However, existing benchmarks remain largely static, offering little support for modeling drifts. This limitation arises from the absence of a shared vocabulary and practical tools for specifying and generating drift in both data and workloads.

Guided by this vision of making drift a first-class concept, we propose a taxonomy of data and workload drift and design *DriftSpec*, a declarative specification that makes these drifts executable. Building on this, we present *DriftBench*, which instantiates DriftSpec to generate controlled drifts and enable drift-aware benchmarking. Together, the taxonomy, DriftSpec, and DriftBench form a first step toward a standardized, executable language for studying how data and workload evolution influence database behavior. They shift benchmarking from static, one-off tests to controlled, continuous evaluation under drift.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Liuguanli/DriftBench.

1 INTRODUCTION

Have you ever read or reviewed a database paper and wondered: *does this system adapt to data or workload*[1] *drift*[2] *?* Or perhaps you have been on the receiving end of the same question, one that is easy to ask but surprisingly difficult to answer rigorously.

We raise this question because both data and workload drift have become increasingly prevalent in real-world systems [17, 57, 60, 61]. As a result, they have attracted growing attention in recent research such as query optimization, cardinality estimation, and learned indices [10, 14, 19, 20, 22, 27, 28, 33, 34, 38, 39, 43, 58], and have been highlighted in community reports such as the Cambridge Report on Database Research [1].

This interest is motivated by the practical impact of drift on database component behavior. For example, both traditional components such as PostgreSQL's cardinality estimator and learned estimators such as Naru [59] and MSCN [19] can become less reliable under data and workload drift. Such shifts can substantially affect estimation accuracy and, in turn, influence downstream query optimization and execution decisions (see Section 5.3).

However, existing research lacks a unified definition of data and workload drift, as well as principled methods for generating them. Instead, drift is often approximated by switching between static datasets [22, 29, 32, 35, 37, 56], or by manually varying query parameters (e.g., predicate ranges [27, 36]) and query types (e.g., *k*NN and range queries in spatial workloads [15, 43]). While these heuristics can simulate drift, they remain ad hoc and may fall short in capturing the dynamic nature of real-world drift.

This limitation is also reflected in existing benchmarks. For example, TPC-H [53] assumes static data and fixed workloads, while the Join Order Benchmark (JOB) [24] focuses on join queries without dynamic workloads. RedBench [21] improves real-world representativeness by deriving from Amazon Redshift [54], but it does not support controlled drift injection over arbitrary data schemas. Thus, drift evaluations remain ad hoc, with limited reproducibility and comparability across systems.

In particular, for data drift, no widely accepted definitions capture common forms of change such as distributional drifts and cardinality variations, or even schema modifications [25, 60, 61]. The notion of workload drift is even more ambiguous [17, 57]. Clarifying the definition of drift is only part of the solution. The community also needs practical tools for constructing drift scenarios.

To fill this gap, we advocate a unified foundation for *drift-aware benchmarking*, enabling the community to evaluate database systems under evolving data and workloads. At the core of this *vision* is a **taxonomy of drift** that captures recurring patterns of evolution observed across diverse database contexts. Building upon this taxonomy, we design **DriftSpec**, a declarative YAML-based *specification* that translates drift types into uniform and interoperable scenarios to bridge benchmark design and system evaluation.

On top of this specification, we envision **DriftBench** as a modular framework that connects declarative drift design with adaptive benchmarking. DriftBench supports the synthesis of **data drift** (e.g., cardinality variation, distribution shifts, outlier injection), **workload drift** (e.g., changes in predicate distributions, joins, and payloads), and **temporal drift** [18] (e.g., trends, cycles, and long-tail evolution). Together, these components provide a foundation for benchmarking that adapts to evolving data and workloads.

This paper makes the following contributions:

- **Taxonomy of Drift.** We conceptualize data, workload, and temporal drift within a unified taxonomy that captures recurring patterns of evolution in prior research on database systems, providing a foundation for systematic reasoning about drift.
- **Drift Specification.** We introduce *DriftSpec*, a declarative, YAML-based specification that operationalizes the taxonomy by encoding drift as explicit and reproducible scenarios. DriftSpec can be instantiated from both benchmark inputs (e.g., TPC-H) and summarized real traces (e.g., RedBench), enabling drift generation in both benchmark-driven and trace-driven settings.
- **Prototype Framework.** We present *DriftBench*, a modular prototype framework that implements DriftSpec for synthesis and evaluation of drifted data and workloads. We demonstrate its use through case studies, illustrating drift-aware benchmarking.

[1] For brevity, we use *workload* to refer to *query workload* throughout the paper.
[2] "Drift" and "shift" are often used interchangeably in data and workload characteristics. In this paper, we use *drift*, following the terminology in NeurDB [60].

2 A WORKING TAXONOMY OF DRIFT

We present a *working taxonomy* of data and workload drift that captures recurring patterns observed across prior systems and benchmarks. Rather than exhaustive coverage, we aim to distill a concise and actionable core for reproducible drift generation.

2.1 Data Drift

Data drift refers to changes in the cardinality or distribution of records within a database. We categorize it into four representative operations observed in prior benchmarks and research works:

(1) **Scaling cardinality** refers to changes in the overall number of records in the dataset, e.g., 0.5×, 2×, 10×. It models *net size* effects under a new snapshot without prescribing how tuples arrived or departed. For example, population records may gradually increase as coverage expands to new regions or as higher birth rates. This is widely used to stress storage footprint and plan costs under larger snapshots [32, 52, 53].

(2) **Updating cardinality** typically applies a time-ordered stream of inserts and deletes, i.e., the dataset is mutated over time rather than resized in one shot. For example, individuals are continuously added and removed due to births, deaths, and migration. This stresses index maintenance and statistics freshness, and is commonly used to evaluate continuous-update behavior [4, 16, 22, 43].

(3) **Shifting column distributions** capture changes in column value distributions, such as increased skewness [3, 49]. This drift appears in spatial related workloads without altering dataset cardinality [26, 27, 43], where sudden event surges shift data concentration. For example, the concentration of students during an enrollment period can abruptly shift the distribution of education-related attributes.

(4) **Injecting outliers** uses rare or extreme values to test system robustness under distributional anomalies [6]. Such outliers can distort column statistics, leading the optimizer to misestimate selectivity and choose suboptimal plans [42]. For example, a small number of records with extremely large household sizes or unusually high incomes can act as outliers and distort column statistics. While this issue has gained attention in learned index research (e.g., data poisoning [20]), it remains largely overlooked in other database components.

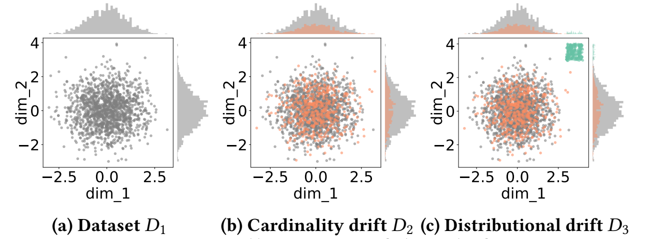
These operations form a practical foundation for modeling data drift. While we focus on these representative types, our goal is not to exhaustively enumerate all possible drift scenarios, but rather to capture the most common patterns. These abstractions motivate the need for a formal definition, which we provide below.

Definition 1 (Data Drift). *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ denote two versions of a dataset over the same schema $\mathcal{S}$. We define data drift as a* ***significant change*** *in the statistical properties or volume of data between $\mathcal{D}_1$ and $\mathcal{D}_2$. It can be characterized along two primary subtypes:*

- ***Cardinality Drift:*** *A substantial change in the total number of records, i.e., $|\mathcal{D}_1|$ and $|\mathcal{D}_2|$ differ by more than a defined threshold α (e.g., $||\mathcal{D}_2| - |\mathcal{D}_1|| > \alpha \times |\mathcal{D}_1|$).*
- ***Distributional Drift:*** *A change in the column value distribution. Such changes can be categorized as (i)* global*, where a divergence metric $\delta(\mathcal{D}_1, \mathcal{D}_2)$ exceeds a threshold ϵ (e.g., changes in skewness); or (ii)* local*, where small-scale modifications (e.g., point injection) affect specific regions of the distribution but can substantially influence system behavior.*

The thresholds α and ϵ in our definition are commonly defined by database systems or practitioners based on operational policies. For example, in Oracle, the deletion of 20% of a table's rows is used as a threshold to trigger index rebuilding [45].

Example. Figure 1 illustrates an example of progressive data drift. In D_1, the data follows a stable normal distribution on two dimensions. In D_2, a new mode emerges (orange), where the data distribution partially overlaps with the original. In D_3, a second mode appears (green), shifted further in value space due to the injection of outliers. This reflects how real-world data can evolve through overlapping shifts.



(a) Dataset D_1 **(b) Cardinality drift D_2** **(c) Distributional drift D_3**

Figure 1: Illustration of data drift.

Discussion. While we define each drift type individually, in practice, multiple types may co-occur between two time points. Our DriftBench supports such compound drift by allowing intermediate drifted datasets to be reused as inputs, enabling the simulation of multi-drift scenarios over time. In addition, our approach inherently supports multi-table schemas, where drift propagates across tables through foreign-key dependencies and inter-table correlations, and is executed via constraint-aware operations to preserve integrity.

2.2 Workload Drift

Workload drift refers to changes in the structure or statistical properties of queries executed against a database over time. We categorize workload drift into four common operations, each of which can significantly impact query processing behavior:

(1) **Changing predicate distributions** reflects drifts in the statistical distribution of predicates over time. This directly affects query optimizers [33, 58] and learned indices [14, 26, 27] that rely on historical access patterns. For example, in a census workload, queries increasingly concentrate on a small number of regions (e.g., major cities), rather than being evenly spread across all areas.

(2) **Varying selectivity** arises when queries from the same logical template exhibit varying predicate ranges [26, 38, 43]. Such drift commonly occurs in response to changing analytical demands (e.g., broader time ranges) and leads to different join strategies or plan choices. For example, a census workload repeatedly uses the same query template while expanding the predicate (e.g., age) ranges over time.

(3) **Modifying query structure** changes in query templates, such as modified predicates or join conditions [33, 34, 38, 56, 58]. These shifts can trigger re-optimization or impact index usage. For example, as analysis requirements evolve, census queries can be extended with additional joins or predicates (e.g., joining household or employment tables).

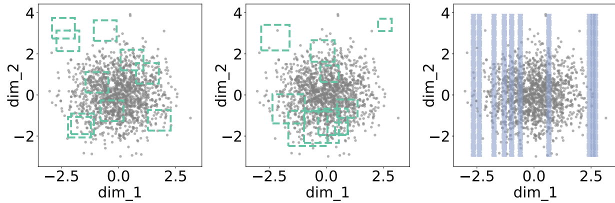
(4) **Changing payloads** refers to changes in the set of projected columns. While it modifies the query structure, we treat it as a separate drift type due to its distinct impact on I/O cost and column scan behavior [21, 36, 56]. For example, a reporting workload evolves from lightweight aggregate queries to detailed record inspection, increasing the number of attributes.

While payload, predicate, and join are conceptually orthogonal, we group predicate and join together because both directly affect cardinality estimation and thus plan generation. In contrast, payload primarily defines the output schema and has a relatively limited impact on execution plans. We now present a formal definition of workload drift based on these observations.

Definition 2 (Workload Drift). *A workload W is defined as a distribution $P(W_\tau(\theta))$ over queries instantiated from a parameterized template τ, where $\theta \in \Theta$ denotes the parameter-generating operator. Let $Sel(W_\tau(\theta))$ denote the total selectivity of W. Workload drift is characterized by the following subtypes:*

- ***Parametric Drift:*** *Two workloads $W_{1,\tau}$ and $W_{2,\tau}$ are instantiated from the same template τ, but differ in their parameter distribution. That is, for some $\theta_1, \theta_2 \in \Theta$, $\delta(P(W_{1,\tau}(\theta_1)), P(W_{2,\tau}(\theta_2))) > \epsilon$ or $|Sel(W_{1,\tau}(\theta_1)) - Sel(W_{2,\tau}(\theta_2))| > \alpha \times Sel(W_{1,\tau}(\theta))$.*
- ***Structural Drift:*** *Two workloads W_{τ_1} and W_{τ_2} are instantiated from templates τ_1 and τ_2, respectively, with structural differences in predicates, joins, or payloads.*

We omit the detailed explanation of δ, α, and ϵ, as they have been discussed in the context of data drift.



(a) Workload W_1 **(b) Parametric drift W_2** **(c) Structural drift W_3**

Figure 2: Illustration of workload drift

Example. Figure 2 shows workload drift. Each green dashed box represents a query's predicate region over two dimensions. In W_1, queries are uniformly distributed with identical predicate bounds, modeling a stable workload. In W_2, the predicate ranges shift and vary slightly (i.e., *parametric drift*). In W_3, queries (blue) remove their predicate on `dim_2`, resulting in a broader scan (i.e., *structural drift*). Such evolution mirrors real-world analytics, where user interests and data exploration patterns shift gradually.

Discussion. While data and workload drift may be related in practice, DriftBench handles such coupling through sequential conditioning. Specifically, it first applies data drift to materialize a drifted dataset and then generates workload drift conditioned on this snapshot, enabling related multi-drift scenarios.

Scope and Terminology. Our taxonomy is grounded in recurring patterns from prior research and benchmarking practice. While system-level changes such as configuration updates and hardware upgrades can affect performance, they are not the focus of this work. Although the term *drift* is also used in the context of *concept drift* [13], the two notions differ fundamentally. Concept drift concerns changes data-label relationships in learning streams, whereas our notion of drift targets data and workloads in database systems.

2.3 Temporal Drift Patterns

We model temporal drift as the evolution of data or workloads over time, typically following non-stationary patterns such as bursts, trends, or repeats. Rather than proposing a new taxonomy, we adopt the classification in Sibyl [18], which defines four representative temporal patterns: *uniform, periodic, trend*, and *long-tail*.

Discussion. These patterns can be applied independently or in combination with data and workload drift, enabling more expressive and customizable drift scenarios. For example, query timestamps can be generated per instance to simulate realistic query streams with temporal variation. *Repetitive query instances* are common in real-world cases [21] and can be assigned with periodic timestamps. Similarly, data updates (e.g., insertions and deletions) can follow scheduled intervals to reproduce read-heavy or write-heavy workloads [22]. Moreover, combining temporal drift with data or workload drift can reveal cross-effects such as performance degradation under bursty query arrivals or delayed adaptation to long-term trends. These capabilities enable constructing temporally evolving benchmark scenarios that better reflect real operational dynamics.

3 DRIFTSPEC: A DRIFT SPECIFICATION

To make drift comparable across studies, we argue for a shared, executable, and verifiable specification beyond taxonomy alone. DriftSpec provides this contract: a minimal language that lets researchers state *what* changes, *how much* it changes, and *when* it unfolds, such that independent teams can align on comparable scenarios. Concretely, DriftSpec encodes each drift instance with four compact blocks, defined as follows:

(1) **`type`** declares the drift family, category, and subtype (e.g., data/distribution/column shift), such that the instance is clearly placed in the taxonomy and downstream tools know which operator to invoke.
(2) **`data_source`** specifies input/output endpoints via `kind`, `uri`, optional `table`, and `output_path`, unifying csv files and database sources.
(3) **`variables`** collects the control knobs for execution (e.g., workload size, target columns, drift parameters such as modes, ranges, and seeds). The block is intentionally open ended: task-specific options may be added, and implementations handle parsing, validation, and sensible defaults.
(4) **`temporal`** (optional) defines the temporal pattern of drift (e.g., `uniform`, `periodic`, `trend`, `long_tail`) that controls how drift evolves or arrives over time.

In short, `type` says *what*, `variables` says *how* and *how much*, `temporal` says *when*, and `data_source` makes the description executable and shareable. We illustrate DriftSpec with a minimal example rather than exhaustively detailing every field, and refer readers to the shared artifacts for more runnable YAMLs.

Example 3.1 (Data drift specification). In Listing 1, DriftSpec declares a data-side distribution drift by right-skewing `age` while preserving basic moments and bounds, with explicit input/output paths for reproducibility. The `type` block anchors the instance

```yaml
pattern_id: dp-d-census-01
seed: 42   # optional
type:                # (family/category/subtype)
  family: data
  category: distribution
  subtype: column_shift
data_source:
  kind: postgres
  uri: 'postgresql://user:pass@host:5432/dbname'
  table: public.census
  output_path: outputs/dp-d-census-01.csv
variables:           # vars from definitions
  params:
    numeric:
      column: age
      mode: skew_right
      bounds: [18, 90]
temporal:            # optional
  pattern: uniform
  rate_per_sec: 5
  duration_sec: 600
```

Listing 1: DriftSpec for a data drift on age column.

(Data→Distribution→Column); `variables` supplies transformation parameters and an example magnitude threshold.

Takeaway. DriftSpec is a generic, executable description of drift scenarios. It builds on our formal definitions to provide a concise protocol file that different stakeholders (e.g., authors and reviewers) can read, write, and share. This makes drift scenarios consistently specified, comparable, and reusable. In addition, DriftSpec is extensible to benchmarks such as TPC-H [53]. For data drift, we follow the original schema to produce drift operations over the benchmark tables. On the workload side, we reuse the benchmark SQL templates and specify the parameter distributions in DriftSpec to generate workloads. For real traces [2], summarized traces produced by tools such as RedBench [21] are mapped to DriftSpec, enabling trace-driven drift without replaying raw queries.

4 DRIFTBENCH: A PROTOTYPE FRAMEWORK

DriftBench turns the taxonomy and DriftSpec into runnable experiments: it ingests real data sources, extracts schema information, and materializes drifted datasets and workloads from the specifications. As shown in Figure 3, intermediate results such as schema, distributions, and templates are reused to generate drift scenarios. DriftBench has seven key components:

(1) **DriftSpec parser** serves as the entry point of DriftBench, responsible for translating declarative YAML specifications into executable drift scenarios. It validates the structure and semantics of a given specification, expands parameter ranges, and resolves references between data, workload, and temporal components.
(2) **Schema extractor** parses the input data source and extracts a schema containing column-level metadata, such as logical types (e.g., numeric and categorical), value ranges, and histogram bounds. It supports both file-based and relational sources, with current implementations for CSV and PostgreSQL, two of the most commonly used formats in benchmarking and prototyping.
(3) **Distribution simulator** applies controlled transformations to simulate data drift, including value skew, outlier injection, cardinality variation, and selective deletion. Drift can be column-specific and parameterized by intensity.

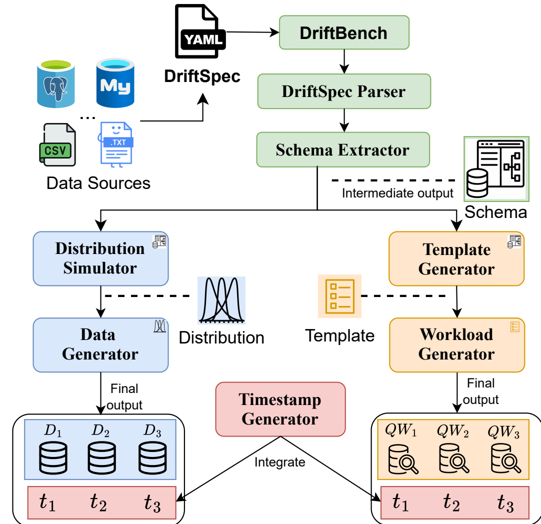


Figure 3: Architecture of DriftBench.

(4) **Data generator** synthesizes new rows based on column-level statistics inferred from the schema, using type-aware methods such as KDE [48] for numerics and frequency-based sampling for categoricals. The module supports cardinality scaling, row deletions, and insertions. For multi-table cases, we generate relation-consistent data by jointly sampling join keys and ensuring that referencing tables include values compatible with the base tables.
(5) **Template generator** generates parameterized query templates over the extracted schema. The generator supports both single-table and multi-table workloads by leveraging column-level statistics and join candidates. The key difference is that multi-table templates include *join* clauses. We assume that join relationships are either defined in the schema or provided by the user.
(6) **Workload generator** instantiates query templates by sampling concrete predicate values from specified distributions (e.g., uniform, normal, Zipfian). This module transforms abstract logical templates to executable queries.
(7) **Timestamp generator** can generate timestamps following defined patterns. It simulates non-stationary query arrival processes by controlling inter-arrival intervals, enabling realistic modeling of time-evolving workloads.

The current DriftBench prototype is implemented in Python with adapters for data sources. It fully implements DriftSpec, executing deterministic runs to ensure reproducibility. This prototype serves as a *proof of concept* for the broader vision of establishing DriftSpec as a shared protocol and DriftBench as a modular framework.

5 CASE STUDIES

To demonstrate the practical use of DriftBench for drift-aware benchmarking, we present case studies that systematically generate, visualize, and analyze data, workload, and temporal drifts using the real-world `census` dataset [46, 56].

5.1 Data Drift

We select two representative attributes from the `census` dataset: one numeric (`age`) and one categorical (`workclass`). For categorical features, we show only the top 5 most frequent categories.

5.1.1 *Updating Cardinality.*

Cardinality updates capture row-level changes caused by data insertions and deletions. While insertions can reuse the scaling functionality, deletions must operate on existing data. To simulate realistic deletions, we randomly select 10% of records from the `census` dataset. As shown in Figure 4, the deleted subset is sampled proportionally to the original distribution, ensuring the operation reflects natural workload patterns without introducing artificial bias.

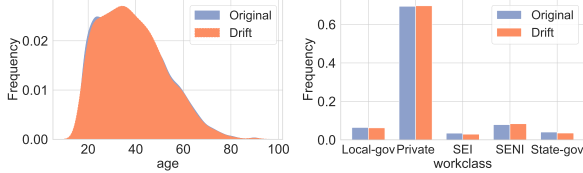


Figure 4: Data distributions under cardinality updates.

5.1.2 *Shifting Column Distributions.*

We simulate drift by replacing original columns with more skewed distributions. For numeric attributes, we generate values that preserve the mean and standard deviation but increase skewness toward one side. For categorical attributes, we upweight the most frequent categories to increase skew, amplifying the dominance of popular values. Figure 5 illustrates these changes for representative columns.

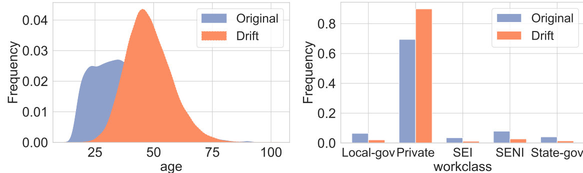


Figure 5: Data distributions under column skew.

5.2 Workload Drift

We use the `age` column from the `census` dataset to illustrate different forms of workload drift in conjunction with timestamps.

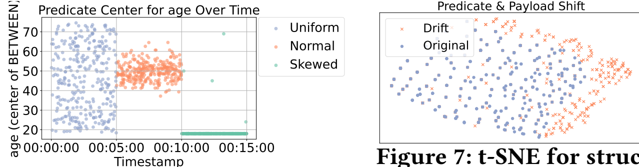


Figure 6: Predicate center drift

Figure 7: t-SNE for structural drift.

5.2.1 *Changing Predicate Distributions.*

We demonstrate predicate shift with three workloads by varying the value distribution of the `age` column across three timestamp groups, each uniformly distributed within five minutes. As shown in Figure 6, the query predicates follow uniform, normal, and skewed distributions, respectively, reflecting increasing levels of locality and skew.

5.2.2 *Structural Drift.*

We group these two operations (*modifying query structure* and *changing payloads*) together, as logical mutations often co-occur with changes in the number of predicates, joins, and other clauses. These changes are structural and difficult to visualize directly. To capture their overall effect, we extract high-level features from each query template and project them using t-SNE (t-distributed stochastic neighbor embedding). Figure 7 visualizes the difference between two sets of query templates: one generated with at most 5 predicates and 6 projected columns, and another with larger limits of 7 and 8, respectively. The resulting clusters reflect how query structures evolve under logical and payload drift.

5.3 Evaluating Estimator Behavior under Drift

We evaluate how cardinality estimators respond to different types of drift by analyzing their behavior under systematic data and workload changes. We use three representative estimators: PostgreSQL (rule-based), Naru [59] (data-driven), and MSCN [19] (data- and query-driven). Using the `census` dataset, we model data drift through cardinality updates, generating ten successive snapshots (D1–D10) that uniformly scale the dataset from 1.0X to 3.0X over time. To model workload drift, we use six-phase temporal workloads that vary predicates and selectivity over time (W1: `age` and `workclass`; W2: `education` and `marital_status`; W3: `capital_gain` and `occupation`; W4: `hours_per_week` and `workclass`; W5: `capital_loss` and `marital_status`; W6: `age` and `native_country`).

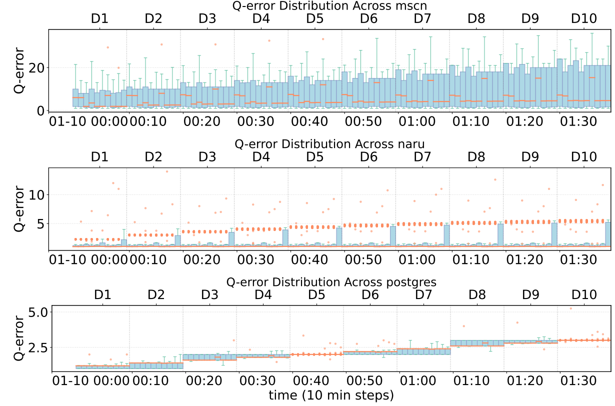


Figure 8: Q-error under data drift with a fixed workload.

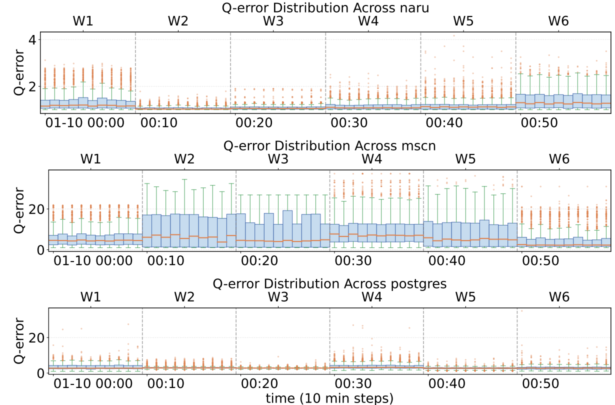


Figure 9: Q-error under workload drift with a fixed dataset.

As shown in Figure 8, as dataset cardinality increases across successive snapshots, the same query workload leads to larger true result sizes, while both PostgreSQL and learned models rely on stale statistics or models that are not updated in time. As a result, Q-error increases and outlier behavior becomes more pronounced, indicating growing estimation mismatch as data evolves.

In contrast, Figure 9 highlights estimator behavior under workload drift. Here, MSCN shows particularly strong sensitivity to query changes, as its predictions vary substantially across workload phases. This behavior is expected, since MSCN is trained on

query features and thus directly depends on workload distributions. By comparison, PostgreSQL and Naru, which are primarily data-driven, exhibit more stable behavior across workload phases.

In addition, we evaluate learned indices via a benchmark [22] under data drift (cardinality scaling, distribution skew) and workload drift (range growth), revealing that PGM [11] generally outperforms FITing Tree [12] and B^+-tree [50], while exhibiting occasional sensitivity to drift. Due to space limitations, these results are provided in the artifacts under the indexing task. These findings underscore the importance of evaluating database components under controlled data and workload drift. DriftBench enables such analysis by systematically generating drift scenarios.

6 OPEN CHALLENGES AND ROADMAP

Our case studies demonstrate that controlled drift provides a structured way to observe estimator behavior under evolving data and workload conditions. However, these results also highlight how early we are in this journey, as current drift generation still relies on manual design and has limited integration with mature benchmarks. We therefore outline **open challenges** and present a **roadmap** toward an automated foundation for drift-aware benchmarking.

We identify four open challenges for drift-aware benchmarking:

(1) **From ad hoc to auto.** Many parameters in drift generation still rely on expert intuition. Achieving automation demands monitoring- and history-driven models that infer drift types and trigger conditions from system logs or workload traces.
(2) **Mining drift from real-world traces.** We have demonstrated that DriftSpec can extend benchmark suites such as TPC-H [53] and RedBench [21]. However, mining drift from real-world traces remains challenging. Existing approaches are often coarse-grained, leaving opportunities to identify richer drift types and patterns that can be translated into DriftSpec.
(3) **From drift to robust systems.** Drift generation itself is not the goal. The real opportunity lies in using observed performance regressions, such as estimator mispredictions or plan volatility, to guide system repair and continuous improvement.
(4) **Cross-domain drift generalization.** Beyond tabular data, new abstractions are needed to express and quantify drift across heterogeneous data modalities, such as vector [40] and spatial data [41] stored in relational systems (e.g., PostgreSQL) and accessed via specialized operators.

Building upon these challenges, our roadmap focuses on practical paths toward standardization and community adoption:

(1) **From ad hoc to guided automation.** We envision a gradual path toward automation that begins with mining query templates and workload statistics from real-world traces [21] to define domain-specific drift types and parameter ranges. Within this structured drift space, *DriftBench* supports sensitivity-guided exploration to identify hot spots where drift has the strongest system impact. In practice, evaluation can start from a broad set of lightweight drift scenarios and progressively focus on hot spots that exhibit degrading behavior.
(2) **Closing the loop.** Beyond generating drifted datasets, our long-term goal is to use drift feedback to strengthen systems. Controlled drift experiments can serve as regression guards, guiding optimizer, estimator, and index adaptation in continuous integration pipelines.
(3) **Beyond tabular domains.** Extending DriftSpec and DriftBench to columnar, vector, graph, and document stores remains an engineering frontier. Once the methodology stabilizes, this cross-domain generalization will foster a **shared, evolvable drift ecosystem** spanning diverse data models.

7 RELATED WORK

Classic benchmarks such as TPC-H [53] and TPC-DS [52] evaluate analytical workloads over static schemas and datasets, while TPC-C [51] and OLTPBench [8] focus on transactional throughput. DSB [9] introduces dynamic workloads on top of TPC-DS but does not explicitly control data or workload drift. LST-Bench [7] further explores benchmarking for log-structured tables through package-based workloads that capture distinct access patterns.

Workload-aware approaches have been explored from both generation and evaluation perspectives. QAGen [30] and its extensions [31] generate databases that satisfy query-level constraints, while application-oriented workload generators [44] assume fixed data characteristics. Surprise benchmarking [5] evaluates systems under unexpected workloads, but these approaches do not model temporal evolution or support systematic data or workload drift.

RedBench [21, 55] takes a step toward realism by incorporating real-world query patterns sampled from Amazon Redshift [2], capturing query and distribution drift. However, it remains limited to replaying existing queries and does not support controlled drift generation or workload synthesis across schemas.

Recent systems such as SQLStorm [47] and SQLBarber [23] leverage LLMs to generate executable SQL workloads, emphasizing query diversity and cost-awareness through generation and filtering pipelines. Nevertheless, they treat workloads as static artifacts and do not provide explicit control over data evolution.

8 CONCLUSIONS

We envision transforming *drift* from an abstract notion into a concrete, executable, and reproducible dimension of data management. This vision is articulated through three complementary elements: a compact taxonomy that formalizes drift, a declarative specification to describe drift, and a framework that brings these descriptions to life across data, workload, and temporal dimensions. Together, these elements make drift a tangible concept, enabling researchers to specify *what* changes, *how much*, and *when*.

Looking ahead, automating drift detection and generation can shift database evaluation from passive observation to active validation, while drift-aware evaluation can inform the design of adaptive optimizers, estimators, and indexes by exposing system behavior under evolving data and workloads. Building on this perspective, such drift-aware investigations are currently underway and represent a natural next step beyond the scope of the present paper. In addition, extending drift beyond tabular data to spatial, vector, and graph models further opens the door to a cross-domain evaluation ecosystem. This perspective invites the community to view database evaluation as a continuous process that measures not only performance, but also a system's capacity to adapt and evolve.

ACKNOWLEDGMENTS

Renata Borovica-Gajic is supported by ARC DECRA DE230100366 and the 2025 Google Foundational Science Fund.

REFERENCES

- [1] Anastasia Ailamaki, Samuel Madden, Daniel Abadi, Gustavo Alonso, Sihem Amer-Yahia, Magdalena Balazinska, Philip A. Bernstein, Peter Boncz, Michael Cafarella, Surajit Chaudhuri, Susan Davidson, David DeWitt, Yanlei Diao, Xin Luna Dong, Michael Franklin, Juliana Freire, Johannes Gehrke, Alon Halevy, Joseph M. Hellerstein, Mark D. Hill, Stratos Idreos, Yannis Ioannidis, Christoph Koch, Donald Kossmann, Tim Kraska, Arun Kumar, Guoliang Li, Volker Markl, Renée Miller, C. Mohan, Thomas Neumann, Beng Chin Ooi, Fatma Ozcan, Aditya Parameswaran, Ippokratis Pandis, Jignesh M. Patel, Andrew Pavlo, Danica Porobic, Viktor Sanca, Michael Stonebraker, Julia Stoyanovich, Dan Suciu, Wang-Chiew Tan, Shiv Venkataraman, Matei Zaharia, and Stanley B. Zdonik. 2025. The Cambridge Report on Database Research. *CoRR* abs/2504.11259 (2025).
- [2] Amazon AWS. 2016. *https://aws.amazon.com/quickstart/architecture/amazon-redshift*. Accessed: 2025-05-31.
- [3] Lars Arge, Mark De Berg, Herman Haverkort, and Ke Yi. 2008. The Priority R-tree: A Practically Efficient and Worst-case Optimal R-tree. *ACM Transactions on Algorithms* 4, 1 (2008), 9:1–9:30.
- [4] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*. 322–331.
- [5] Lawrence Benson, Carsten Binnig, Jan-Micha Bodensohn, Federico Lorenzi, Jigao Luo, Danica Porobic, Tilmann Rabl, Anupam Sanghi, Russell Sears, Pınar Tözün, and Tobias Ziegler. 2024. Surprise Benchmarking: The Why, What, and How. In *DBTest*. 1–8.
- [6] Battista Biggio, Blaine Nelson, and Pavel Laskov. 2012. Poisoning Attacks gainst Support Vector Machines. In *ICML*. 1467–1474.
- [7] Jesús Camacho-Rodríguez, Ashvin Agrawal, Anja Gruenheid, Ashit Gosalia, Cristian Petculescu, Josep Aguilar-Saborit, Avrilia Floratou, Carlo Curino, and Raghu Ramakrishnan. 2024. LST-Bench: Benchmarking Log-Structured Tables in the Cloud. *Proc. ACM Manag. Data* 2, 1 (2024). https://doi.org/10.1145/3639314
- [8] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: an extensible testbed for benchmarking relational databases. *PVLDB* 7, 4 (2013), 277–288.
- [9] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: a decision support benchmark for workload-driven and traditional database systems. *PVLDB* 14, 13 (2021), 3376–3388.
- [10] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. *PVLDB* 12, 9 (2019), 1044–1057.
- [11] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB* 13, 8 (2020), 1162–1175.
- [12] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-Aware Index Structure. In *SIGMOD*. 1189–1206.
- [13] João Gama, Indrundefined Žliobaitundefined, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A Survey on Concept Drift Adaptation. *ACM Comput. Surv.* 46, 4 (2014), 37.
- [14] Jian Gao, Xin Cao, Xin Yao, Gong Zhang, and Wei Wang. 2023. LMSFC: A Novel Multidimensional Index based on Learned Monotonic Space Filling Curves. *PVLDB* 16, 10 (2023), 2605–2617.
- [15] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. 2023. The RLR-Tree: A Reinforcement Learning Based R-Tree for Spatial Data. *PACMMOD* 1, 1 (2023), 63:1–63:26.
- [16] Antonin Guttman. 1984. R-trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*. 47–57.
- [17] Marc Holze and Norbert Ritter. 2007. Towards workload shift detection and prediction for autonomic databases. In *Ph.D. Workshop in CIKM*. 109–116.
- [18] Hanxian Huang, Tarique Siddiqui, Rana Alotaibi, Carlo Curino, Jyoti Leeka, Alekh Jindal, Jishen Zhao, Jesús Camacho-Rodríguez, and Yuanyuan Tian. 2024. Sibyl: Forecasting Time-Evolving Query Workloads. *PACMMOD* 2, 1 (2024), 27.
- [19] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *CoRR* abs/1809.00677 (2018).
- [20] Evgenios M. Kornaropoulos, Silei Ren, and Roberto Tamassia. 2022. The Price of Tailoring the Index to Your Data: Poisoning Attacks on Learned Index Structures. In *SIGMOD*. 1331–1344.
- [21] Skander Krid, Mihail Stoian, and Andreas Kipf. 2025. Redbench: A Benchmark Reflecting Real Workloads. *CoRR* abs/2506.12488 (2025).
- [22] Hai Lan, Zhifeng Bao, J. Shane Culpepper, and Renata Borovica-Gajic. 2023. Updatable Learned Indexes Meet Disk-Resident DBMS - From Evaluations to Design Choices. *PACMMOD* 1, 2 (2023).
- [23] Jiale Lao and Immanuel Trummer. 2023. SQLBarber: A System Leveraging Large Language Models to Generate Customized and Realistic SQL Workloads. *CoRR* abs/2309.06354 (2023). https://doi.org/10.48550/arXiv.2507.06192
- [24] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *PVLDB* 9, 3 (2015), 204–215.
- [25] Beibin Li, Yao Lu, and Srikanth Kandula. 2022. Warper: Efficiently Adapting Learned Cardinality Estimators to Data and Workload Drifts. In *SIGMOD*. 1920–1933.
- [26] Jiangneng Li, Zheng Wang, Gao Cong, Cheng Long, Han Mao Kiah, and Bin Cui. 2023. Towards Designing and Learning Piecewise Space-Filling Curves. *PVLDB* 16, 9 (2023), 2158–2171.
- [27] Guanli Liu, Lars Kulik, Christian S. Jensen, Tianyi Li, Renata Borovica-Gajic, and Jianzhong Qi. 2024. Efficient Cost Modeling of Space-filling Curves. *PVLDB* 17, 13 (2024), 4773–4785.
- [28] Guanli Liu, Jianzhong Qi, Christian S. Jensen, James Bailey, and Lars Kulik. 2023. Efficiently Learning Spatial Indices. In *ICDE*. 1572–1584.
- [29] Qiyu Liu, Maocheng Li, Yuxiang Zeng, Yanyan Shen, and Lei Chen. 2024. How Good Are Multi-dimensional Learned Indices? An Experimental Survey. *CoRR* abs/2405.05536 (2024).
- [30] Eric Lo, Nick Cheng, and Wing-Kai Hon. 2008. QAGen: Generating Query-Aware Test Databases. In *VLDB*. 341–352.
- [31] Eric Lo, Nick Cheng, and Wing-Kai Hon. 2010. Generating Databases for Query Workloads. *PVLDB* 3, 1–2 (2010), 848–859.
- [32] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *PVLDB* 14, 1 (2020), 1–13.
- [33] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *SIGMOD*. 1275–1288.
- [34] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: a learned query optimizer. *PVLDB* 12, 11 (2019), 1705–1718.
- [35] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2019. Learning Multi-dimensional Indexes. In *MLForSys@NeurIPS*.
- [36] Shoji Nishimura and Haruo Yokota. 2017. QUILTS: Multidimensional Data Partitioning Framework Based on Query-Aware and Skew-Tolerant Space-Filling Curves. In *SIGMOD*. 1525–1537.
- [37] Varun Pandey, Alexander van Renen, Andreas Kipf, Ibrahim Sabek, Jialin Ding, and Alfons Kemper. 2020. The Case for Learned Spatial Indexes. *CoRR* abs/2008.10349 (2020).
- [38] R. Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic. 2021. DBA bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees. In *ICDE*. 600–611.
- [39] R. Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic. 2023. No DBA? No Regret! Multi-Armed Bandits for Index Tuning of Analytical and HTAP Workloads With Provable Guarantees. *IEEE TKDE* 35, 12 (2023), 12855–12872.
- [40] pgvector Contributors. 2026. *pgvector*. Accessed: 2026-01-10.
- [41] PostGIS. 2024. *https://postgis.net/docs/manual-3.5/using_postgis_dbmanagement.html#spgist_indexes*. Accessed: 2024-12-27.
- [42] PostgreSQL. 2025. *Routine Vacuuming - PostgreSQL*. https://www.postgresql.org/docs/current/routine-vacuuming.html Accessed: 2025-07-15.
- [43] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. 2020. Effectively Learning Spatial Indices. *PVLDB* 13, 11 (2020), 2341–2354.
- [44] Luyi Qu, Yuming Li, Rong Zhang, Ting Chen, Ke Shu, Weining Qian, and Aoying Zhou. 2022. Application-Oriented Workload Generation for Transactional Database Performance Evaluation. In *ICDE*. 420–432.
- [45] Robert Gravelle. 2020. *Navicat Blog: Oracle Rebuild*. https://www.navicat.com/en/company/aboutus/blog/1303-how-to-tell-when-it-s-time-to-rebuild-indexes-in-oracle Accessed: 2025-07-01.
- [46] Ron Kohavi. 2025. *Census Income*. https://archive.ics.uci.edu/dataset/20/census+income Accessed: 2025-05-15.
- [47] Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. 2025. SQLStorm: Taking Database Benchmarking into the LLM Era. *Proc. VLDB Endow.* 18, 11 (2025), 4144–4157.
- [48] Bernard W. Silverman. 1986. *Density Estimation for Statistics and Data Analysis*. Springer.
- [49] Snowflake Inc. 2025. *SKEW function*. https://docs.snowflake.com/en/sql-reference/functions/skew
- [50] STX B+ Tree. 2007. *https://panthema.net/2007/stx-btree*. Accessed: 2023-05-31.
- [51] Transaction Processing Performance Council. 2025. *TPC-C Benchmark*. https://www.tpc.org/tpcc/ Accessed: 2025-06-31.
- [52] Transaction Processing Performance Council. 2025. *TPC-DS Benchmark*. https://www.tpc.org/tpcds/ Accessed: 2025-06-31.
- [53] Transaction Processing Performance Council. 2025. *TPC-H Benchmark*. https://www.tpc.org/tpch/ Accessed: 2025-06-31.
- [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. *PVLDB* 17, 11 (2024), 3694–3706.
- [55] Johannes Wehrstein, Roman Heinrich, Mihail Stoian, Skander Krid, Martin Stemmer, Andreas Kipf, Carsten Binnig, and Muhammad El-Hindi. 2025. Redbench:

Workload Synthesis From Cloud Traces. *CoRR* abs/2511.13059 (2025).
[56] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are updatable learned indexes ready? *PVLDB* 15, 11 (2022), 3004–3017.
[57] Peizhi Wu and Zachary G. Ives. 2024. Modeling Shifting Workloads for Learned Database Systems. *PACMMOD* 2, 1 (2024), 27.
[58] 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*. 931–944.
[59] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep unsupervised cardinality estimation. *PVLDB* 13, 3 (2019), 279–292.
[60] 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*.
[61] Zhanhao Zhao, Haotian Gao, Naili Xing, Lingze Zeng, Meihui Zhang, Gang Chen, Manuel Rigger, and Beng Chin Ooi. 2025. NeurBench: Benchmarking Learned Database Components with Data and Workload Drift Modeling. *CoRR* abs/2503.13822 (2025).