# DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems

Bailu Ding
Microsoft Research
badin@microsoft.com

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Johannes Gehrke
Microsoft Research
johannes@microsoft.com

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

## ABSTRACT

We describe a new benchmark, DSB, for evaluating both workload-driven and traditional database systems on modern decision support workloads. DSB is adapted from the widely-used industrial-standard TPC-DS benchmark. It enhances the TPC-DS benchmark with complex data distribution and challenging yet semantically meaningful query templates. DSB also introduces configurable and dynamic workloads to assess the adaptability of database systems. Since workload-driven and traditional database systems have different performance dimensions, including the additional resources required for tuning and maintaining the systems, we provide guidelines on evaluation methodology and metrics to report. We show a case study on how to evaluate both workload-driven and traditional database systems with the DSB benchmark. The code for the DSB benchmark is open sourced and is available at https://aka.ms/dsb.

**PVLDB Reference Format:**
Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems PVLDB, 14(13): 3376 - 3388, 2021.
doi:10.14778/3484224.3484234

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://aka.ms/dsb.

## 1 INTRODUCTION

The TPC-DS benchmark [24] is a decision support benchmark that models the database and the workload for the data warehouse of a retail product supplier. It includes realistic database schema, real-world data distribution, and semantically meaningful queries. The benchmark has been widely used as an industrial-standard benchmark for evaluating commercial database systems.

While the TPC-DS benchmark has many merits noted above, after more than a decade since its publication, we observe that some new trends have emerged in decision support workloads that challenge the design of the TPC-DS benchmark:

Proceedings of the VLDB Endowment, Vol. 14, No. 13 ISSN 2150-8097.
doi:10.14778/3484224.3484234

- **More complex data distribution**: The data distribution in modern decision support databases consists of a variety of skews and correlations, including skews on individual columns, correlations between columns in the same table, and correlations of columns across multiple tables [4, 22, 24]. While the TPC-DS benchmark models skews in certain individual columns, it mostly assumes independence between columns and across tables.
- **More varieties in queries**: The queries in modern decision support workloads have a variety of join patterns, including non-equi joins and cyclic join graphs. In particular, queries sometimes join tables on columns that are neither primary keys nor foreign keys, e.g., many-to-many joins. However, the joins in the TPC-DS benchmark are mostly between a primary key and a foreign key or between a primary key and another primary key.
- **More fine-grained data slicing**: Exploratory queries often slice data at a fine granularity with multiple predicate filters on table columns. However, since the dominating tasks for decision support queries were reporting queries at the time when the TPC-DS benchmark was designed, many query templates in TPC-DS aggregate data at a much coarser granularity.

In addition to the changes in decision support workloads, there have also been recent initiatives to make database systems more *workload-driven*, especially with machine learning (ML) techniques. There have been several research efforts that attempt to re-architect database system components to take advantage of the workload and its execution feedback. These efforts exploit ML to address a wide range of challenging problems in index tuning, query processing, and query optimization [8, 12, 13, 19]. This leads to new requirements on the benchmark that aims to evaluate such workload-driven database systems:

- **More distinct query instances**: As a result of fine-grained data slicing, queries are issued with various parameter bindings. Thus, a benchmark should be capable of populating a sufficient number of *distinct* query instances. While the TPC-DS benchmark has parameterized query templates, the number of distinct query instances that can possibly be generated can be as small as 5 (Section 4.1).
- **Dynamic workloads**: Workloads are rarely static in practice. The parameter distributions, and hence the selectivity of parameterized predicate filters can vary over time. Moreover, the distribution of query templates can also change in real workloads. To evaluate the performance and adaptability of workload-driven database systems, a benchmark should be capable of populating dynamic workloads. Unfortunately, existing benchmarks such

as the TPC-DS benchmark only generate static workloads with limited variations.

- **More dimensions of performance metrics**: Compared with the traditional database architectures, workload-driven database systems are often designed using a different paradigm. There can be a warm-up phase where the database system observes the workload, collects the execution statistics, and tunes its performance based on the workload. In addition, workload-driven database systems can utilize a feedback loop to continuously monitor the workload and adapt to changes if needed. This raises additional considerations for the evaluation metrics in the benchmark. For example, the query optimizers leveraging ML techniques often need to collect execution data and train the ML models that these optimizers depend on before executing any query. Furthermore, during query optimization and query execution, the ML-enhanced optimizers may spend additional resources to perform model inference (e.g., for cardinality estimation), data collection, and model retraining. Since the TPC-DS benchmark is designed for traditional database architectures, it does not provide enough guidance on how to comprehensively compare the performance trade-offs for workload-driven and traditional database systems.

We propose a new benchmark, DSB, for evaluating both workload-driven and traditional database systems on modern decision support workloads. DSB is adapted from the widely-used industrial-standard TPC-DS benchmark[1]. Our adaptation preserves the realistic database schema from TPC-DS, and it enriches the TPC-DS benchmark in a number of ways:

- **More complex data distribution:** We introduce more skews and correlations to individual columns and multiple columns within a table and across tables (Section 3).
- **New query templates with more join patterns:** We introduce new and semantically meaningful query templates to enrich the join patterns in the benchmark, i.e., non primary-key-foreign-key joins and non-equi joins (Section 4.3).
- **More distinct query instances with fine-grained data slicing:** We introduce additional predicate filters in the query templates for fine-grained data slicing. This also increases the total number of distinct query instances (Section 4.1).
- **Configurable and dynamic query workloads:** We introduce configurable parameterization in generating queries, which enables creating workloads of different parameter distributions and workloads that change over time (Section 4.2). The DSB benchmark can be especially useful for evaluating a large body of prior work on workload-driven database systems, including [18–20, 25], which needs sufficient query instances for tuning the system and / or dynamic workloads to assess their capability of adapting to changes.
- **Guidelines on evaluating workload-driven and traditional database systems**: Because a workload-driven database system observes the workload, passively or actively collects the execution feedback, and tunes itself based on the workload, it can require additional inputs and resources, such as sample workloads, computation, and storage. Since the workload-driven and traditional database systems have different performance dimensions, it has been challenging to compare their performance due to the diverse evaluation setups and metrics reporting [13, 15, 19, 20, 25]. DSB provides detailed methodology on how to evaluate these database systems (Section 5). We also give guidance on the information to disclose and metrics to report.

We analyze and compare the characteristics of the DSB benchmark and the TPC-DS benchmark (Section 6). We show that, with 1000 query instances generated per query template, the DSB benchmark only has 8% of duplicate query instances; in contrast, the TPC-DS benchmark has 63% of duplicate query instances. Compared with the TPC-DS benchmark, the DSB benchmark leads to more complexity in query optimization on both Microsoft SQL Server 2019 and Postgres 13. Finally, we show a case study of how to evaluate both workload-driven and traditional database systems using the DSB benchmark (Section 7). We conclude in Section 8. Although we have started with the TPC-DS benchmark due to its popularity and extensive adoption, our adaptation and the evaluation methodology can be extended to enhance other benchmarks. The code for the DSB benchmark is open sourced and is available at [1].

## 2 RELATED WORK

In this section, we describe related work on decision support benchmarks and data distribution generation.

**Benchmarks:** The TPC-DS benchmark [24] is proposed to evaluate the performance of decision support database systems. It models the data warehouse of a retail product supplier. Its schema consists of multiple snowflake schemas with shared dimension tables. The TPC-DS benchmark incorporates real-world data distribution into the generation of the database. The database itself can scale from 1GB to 100TB. The TPC-DS benchmark defines 99 parameterized query templates to model the decision support tasks in retail. The benchmark has been widely used as an industrial-standard benchmark for commercial database systems.

The Join Order Benchmark (JOB) [16] is proposed to evaluate the cardinality estimation and join ordering in query optimization. The benchmark uses the Internet Movie Data Base (IMDB) dataset [2]. It consists of 113 single-block, select-project-join query instances. The queries are synthetic. The presence of skews and correlations in the data as well as the large number of joins and selections in the queries make JOB challenging for query optimizers with respect to cardinality estimation and join ordering.

The TPC-H benchmark [23] is a decision support benchmark that precedes TPC-DS. JCC-H [7] introduces join-crossing-correlations and skews into TPC-H. The CH Benchmark [9] combines TPC-H and TPC-C to evaluate database systems on hybrid transaction and analytical (HTAP) workloads.

Compared with prior benchmarks, the DSB benchmark is designed for evaluating both workload-driven and traditional database systems on modern decision support workloads. It can populate a large number of distinct query instances, dynamic workloads, and databases with complex skews and correlations. The DSB benchmark can also be extended to use alternative query parameter generation techniques such as [14].

[1]Disclaimer: The DSB benchmark is derived from TPC-DS and as such is not comparable to published TPC-DS results, as the DSB benchmark does not comply with the TPC-DS benchmark.

**Data distribution generation:** Random data distribution is commonly used to populate synthetic datasets, where the value of a column is uniformly drawn from a given domain. Additional data distributions, such as Zipfian and exponential data distribution, are also widely used for introducing skews into the data [3, 4]. Prior work has also generated Zipfian distribution over multiple dimensions with bucketization [5]. TPC-DS introduces realistic data distribution by hard coding a weighted distribution of values in a domain. JCC-H introduces join correlation by controlling the join fanouts [7]. The DSB benchmark leverages techniques from prior work and introduces skews and correlations for single columns and multi-dimensional data within a table and across tables.

## 3 DATA GENERATION

The TPC-DS benchmark incorporates real-world data distribution in generating the database. However, most of the data in the columns are still populated from uniform distribution. In addition, the data in the columns have little correlation. In practice, the data distribution can be more skewed and correlated. Thus, the DSB benchmark enhances the data generation of the TPC-DS benchmark with additional skews and correlations.

### 3.1 Skewed data distribution

Exponential distribution is a skewed distribution that is frequently observed in practice [22]. We introduce three types of skewed distributions to data in a single column for both categorical and numerical values based on exponential distribution. Since generating values from an exponential distribution takes $O(1)$ time and space complexity, it adds negligible overhead to populating the database.

**Exponential distribution for categorical data:** For categorical data with relatively small domains, e.g., i_category in item table, we draw the frequency of each category with an exponential distribution. Specifically, the cumulative distribution function (CDF) of an exponential distribution is

$$f(x) = 1 - e^{-\lambda x} \tag{1}$$

Given the probability $p$, we have

$$x = -\ln(1-p)/\lambda \tag{2}$$

We apply a random permutation to the categorical values to avoid creating a smooth distribution over a fixed order of the values.

**Bucketized exponential distribution for categorical data:** If the number of categories is too large, e.g., dates, drawing the values from an exponential distribution with a random permutation can be noisy. Since such categories often have a natural order, e.g., dates, we bucketize the categories into ranges. We draw a value from a skewed distribution in three steps. First, we select a bucket using an exponential distribution. Then we map the selected bucket into a corresponding range of categorical values. Again, we apply a random permutation to the buckets to avoid creating a smooth distribution for consecutive ranges. Finally, within a bucket, we choose a value in the corresponding range from either uniform distribution or another exponential distribution.

**Bucketized exponential distribution for numerical data:** For numerical data, e.g., wholesale prices, we generate a skewed distribution following a similar process to that of generating bucketized exponential distribution for categorical data.

```
GenPositiveCorrelation(n, a, m, r, d, p):
    Input: Domain size $n$ of column $A$, ID $a$ of a value in the
           domain of column $A$, domain size $m$ of column $B$,
           radius $r \in [0, 1]$, distribution $d$, permutation $p$ of
           the values in the domain of column $B$
    Output: A value $v$ in the domain of column $B$
1   $ratio \leftarrow a/n$
2   $bMin \leftarrow max(1, m * ratio)$
3   $bMax \leftarrow max(1, m * (ratio + radius))$
4   $b \leftarrow GenRandom(dist, bMin, bMax) \mod m$
5   $v \leftarrow MapPermutation(b, p)$
6   return $v$
```

**Algorithm 1:** Generate a value for column $B$ where column $B$ is positively correlated with column $A$. Function *GenRandom* returns a value in a given range with a specified distribution. Function *MapPermutation* takes an ID and a permutation of the values in a domain and returns the value corresponding to the given ID

```
GenJointDistributionMultiTables($D_1.A$, $D_2.B$, F, $map_1$,
  $map_2$, m, r, d, $p_a$, $p_b$):
    Input: Correlated columns $D_1.A$, $D_2.B$, a map $map_1$
           from $D_1.A$ to primary keys of $D_1$, a map $map_2$
           from $D_2.B$ to primary keys of $D_2$, distribution $d$,
           permutation $p_a$ and $p_b$ of the values in the
           domain $D_1.A$ and $D_2.B$
    Output: a pair of primary keys $(k_1, k_2)$
1   $(a, b) = GenJointDistribution(D_1.A, D_2.B, d)$
2   $v_a = MapPermutation(a, p_a)$
3   $\mathcal{A} \leftarrow map_1(v_a)$
4   $v_b = MapPermutation(b, p_b)$
5   $\mathcal{B} \leftarrow map_2(v_b)$
6   $k_1 = SelectRandom(\mathcal{A})$
7   $k_2 = SelectRandom(\mathcal{B})$
8   return $(k_1, k_2)$
```

**Algorithm 2:** Introduce correlations between two columns in two dimension tables that are joined together with a fact table. *GenJointDistribution* draws a pair of values from a given distribution $d$. Function *MapPermutation* takes an ID and a permutation of the values in a domain and returns the value of the corresponding ID. Function *SelectRandom* returns a random value from a given set of values.

### 3.2 Skewed data correlation

To model the rich and complex data correlations, we introduce data correlations for columns both in a single table and columns of the result of joining multiple tables.

We add the following correlations for columns in a single table:

- **Exponential 2D distribution with bucketization:** This models a skewed distribution on combinations of values from two columns in the same table. For a pair of columns $A$ and $B$, we take the cross product of the domains of the two columns and flatten them into one dimension. Then we populate a skewed distribution over the combination of these two column values the same way as described in Section 3.1. Since the cross product of the values in two domains often results in a large number of value combinations, we use bucketization as described earlier.

- **Positive correlation with a driving column:** This models positive correlations between values of two or more columns in the same table. For column $A$ and column $B$, if the value $b$ in the domain of column $B$ is positively correlated with the value $a$ in the domain of column $A$, we call column $A$ the *driving column*. At a high level, we map each value $a$ in the domain of column $A$ to a set of values $\mathcal{B}(a)$ in the domain of column $B$. For each value $a$ in $A$, we select a value $b \in \mathcal{B}(a)$ using either an exponential or uniform distribution. Algorithm 1 shows the details of the process. This can be extended to numerical values with bucketization.

Combining the two techniques above, we can create complex data correlations between multiple columns in the same table. For example, if an exponential 2D distribution is introduced to column $A$ and $B$ and a positive correlation is introduced to column $C$ with driving column $A$, then the introduced data correlation impacts all three column $A, B$ and $C$. The data correlation can implicitly propagate to the join result of multiple tables through join columns.

**Data correlation for joins** In addition to correlations among columns within the same table, realistic data can also have correlations between columns from multiple tables. For example, since the average weather in California is warmer than that in Alaska, the people in California are more likely to buy shorts and dresses than people in Alaska. Since customer addresses, item categories, and the sales data are stored in three tables, this data correlation between customer addresses and item categories can only be observed after the three tables are joined together.

We introduce data correlation between two columns from two dimension tables which are joined together with a fact table. At a high level, we first populate the data in the two dimension tables, and based on the values of the columns and the data correlation, we then decide the value of the corresponding foreign keys on the fact table.

Algorithm 2 shows the process of how to introduce a distribution $d$ between two columns $D_1.A$ and $D_2.B$ to the join result of table $D_1$, $D_2$, and $F$, where $D_1$ and $D_2$ joins with $F$ with primary-key-foreign-key (PKFK) joins. Given the table $D_1$, we record the mapping $map_1 : a \rightarrow \mathcal{K}_1(a)$, where $\mathcal{K}_1(a)$ is the set of primary keys in $D_1$ with $D_1.A = a$ (line 2-3). Similarly, we record the mapping $map_2 : b \rightarrow \mathcal{K}_2(b)$, where $\mathcal{K}_2(b)$ is the set of primary keys in $D_2$ with $D_2.B = b$ (line 4-5). When generating the corresponding foreign keys for a tuple in the table $F$, we first populate a pair of values $(a, b)$ from the given distribution $d$ (line 1), and then we select the values of foreign keys in $F$ from $\mathcal{K}_1(a)$ and $\mathcal{K}_2(b)$ with uniform distribution (line 6-7).

This algorithm can be further extended to populate correlations for columns of join results between a fact table and chains of dimension tables with PKFK joins, i.e., snowflake queries, by propagating the mapping between the correlated columns and the corresponding primary keys.

Note that introducing data correlation into the join result does not affect the data distribution of individual dimension tables. Thus, with additional data correlation among columns within each individual table, the introduced correlation in the join result can span beyond the targeted columns in the join result. For example, if a correlation is introduced to column $D_1.A_1$ and $D_2.B$ in the join result, and another correlation is introduced between column $D_1.A_1$ and $D_1.A_2$, there will be implicit correlation between $D_1.A_2$ and $D_2.B$ in the join result as well.

## 3.3 Physical design configuration

In practice, we often change the physical design configuration of a database to improve query performance, including creating indexes and views. We tune a sample 100GB DSB database instance with a sample workload using the Database Tuning Advisor from Microsoft SQL Server 2019 [6]. The resulting 56 secondary B+ tree indexes can be used as a sample physical design configuration for the benchmark. The specification of the indexes is included in the released code [1].

## 4 QUERY GENERATION

In this section, we describe how the DSB benchmark augments the TPC-DS query templates and populates dynamic workloads. We also discuss how to reduce generating query instances that are literally different but semantically equivalent.

## 4.1 Fine-grained data slicing

While the query templates in TPC-DS are parameterized, they may only aggregate the data at a coarse grained with a small number of predicate filters. As a result, the parameter space of the query templates can be very limited. For example, query template 17 only has one parameter $YEAR$, which ranges from 1998 to 2002, resulting in a total number of 5 distinct query instances.

We augment the query templates from TPC-DS with additional predicate filters for fine-grained data slicing. This increases the parameter space of query templates and enables generating a much larger number of distinct query instances. For example, query template 99 has the sequence ID of the month as its parameter with 48 distinct values in total. We augment this query with additional parameterized predicate filters on call_center, ship_mode, warehouse, and catalog_sales table to increase the space of parameterization, enabling generating more than 10,000 distinct query instances from this augmented query template.

For queries where the domain of the parameters is a subset of the values in the domain, we enlarge the set of the parameter values. For example, query template 3 aggregates the sales data over a $MONTH$, where the $MONTH$ is either November or December. We enlarge the range of the $MONTH$ parameter to be from January to December.

Note that with our augmentation, the semantics of the query templates only change slightly, i.e., inquiring over a smaller or a different slice of the data.

We exclude some query templates from the TPC-DS benchmark in DSB because they have a small number of joins or their parameter space cannot be easily augmented without adding more joins and / or significantly altering the semantics of the queries. We also exclude query templates that are very similar to other query templates, e.g. query 94 and 95.

## 4.2 Varied and dynamic query workloads

Evaluating the adaptability of workload-driven database systems requires a benchmark to generate a variety of workloads as well as dynamic workloads. The TPC-DS query generation follows a given distribution when populating query parameters. This distribution can be either weighted or uniform, and the options of variations are limited. We enhance the query workload generation by populating categorical parameter values in the query templates from multiple Gaussian distributions. The categorical parameter values are first drawn from a random permutation, and then the weights of the parameter values are drawn from a Gaussian distribution, where the mean and the variance of the distribution are configurable.

To further support generating dynamic query workloads, DSB can take a sequence of workload distribution configurations to generate a query workload that changes over time. Each workload distribution configuration includes a set of query templates, the number of query instances per query template, the parameters of the Gaussian distribution (i.e,. the mean and the variance), and the random number generator seed which determines the permutation of the parameter values in each domain.

The DSB benchmark can generate various types of dynamic workloads with different workload distribution configurations. For example, DSB can create a dynamic workload where certain parameter values of the queries become more popular over time by increasing the variance of the Gaussian distribution. DSB can also create a workload where the 'hot', i.e., frequent, parameter values shift over time by changing the mean of the distribution and / or the permutation of the parameter values in each domain. DSB can further create workloads with different degrees of similarities by controlling the differences in the mean and the variance of the Gaussian distributions. The similarity of the parameter value distributions in two workloads can be quantified by measuring the distance (e.g., Kullback–Leibler divergence) of the probability distribution of the parameter values in each domain from the two workloads. Finally, we can control the rate of the workload shift by setting the number of query instances for each workload configuration in the sequence.

## 4.3 Additional join patterns

Since the tables in the TPC-DS database schema are connected by PKFK constraints, the joins in the query templates in TPC-DS are mostly limited to PKFK joins between fact and dimension tables or primary-key-primary-key joins between fact tables. We add three new query templates that inquire customer purchase patterns with additional join patterns, including many-to-many joins, non-equi joins, and cyclic joins:

**Query 100:** Find items that are frequently sold together. This query includes many-to-many self-joins and non-equi self-joins.

**Query 101:** Find cases where an item is purchased and returned from the web, and then the same item is purchased again from the store. This query includes non-PKFK many-to-many joins between fact tables and non-equi joins between dimension tables.

**Query 102:** Find cases where an item is first purchased from the store and then purchased again from the web, where the initial purchase could have been made from the web based on its inventory. This query includes non-PKFK many-to-many joins between fact tables and non-equi joins between fact and dimension tables.

## 4.4 Reduce duplication in query generation

The TPC-DS benchmark can generate query instances that are literally different but semantically equivalent. For example, query template 13 generates a list of states to filter the customer address table. Since the states are used in an 'IN' predicate filter, the permutation of the states in the list does not change the semantics of the query instance. Such equivalent query instances lead to over-counting when measuring the number of distinct query instances by query text. We enhance the syntax of the query templates and query instance generation such that the parameterization is generated with a canonical order to reduce duplication when possible.

## 4.5 DSB derived SPJ queries

Aside from the main benchmark, we also include a set of single-block SPJ queries derived from the query templates in DSB. These query templates are included only to provide a ramp for evaluating techniques that target at optimizing SPJ queries, e.g., techniques on join ordering. The evaluation of the derived single-block SPJ queries is optional for the evaluation of the full DSB benchmark, and the performance on these queries is not required to be reported.

## 5 METHODOLOGY AND METRICS

In this section, we describe how to evaluate a database system with DSB. In particular, we separate the evaluation into the preparation and test stage, and we design our evaluation methodology and metrics to expose the performance trade-offs of workload-driven and traditional database systems. We also give a conceptual example of how to evaluate a traditional and two workload-driven database systems with ML-enhanced query optimizers. Table 1 summarizes the information to disclose and the metrics to report.

## 5.1 Runtime environment

The evaluation should be performed on the same hardware using the same database instance and workload for the database systems of interest. The hardware and software specification should be disclosed. For reproducibility, we recommend to perform the evaluation on hardware that is publicly accessible, e.g., a VM instance in the cloud.

The client can issue queries to the database concurrently. Concurrent query execution will increase resource consumption at runtime and impact the query performance. The degree of concurrency used for the evaluation should be disclosed.

## 5.2 Data and query generation

We provide a software tool to populate the database at different scale factors from 1GB to 100TB. The evaluation should be performed and reported on a database instance of at least 100GB. The databases of 1GB and 10GB should only be used for testing purpose.

We provide a toolkit to populate a workload with a given distribution (Section 4.2). A performance test should be evaluated for at least 10 query instances per query template. For workload-driven database systems, if needed, the toolkit can be used to populate a separate workload for preparing the database system, i.e., a training

**Table 1: Information to disclose and metrics to report in the evaluation**

| Category | Item | Description |
|---|---|---|
| Runtime environment | Hardware | Specification of the hardware. Recommend using hardware that is publicly accessible, e.g., a VM instance in the cloud. |
| | Software | The database runtime. |
| | Concurrency level | The number of concurrent client connections that issue the queries. |
| Data and query generation | Database scale | Scale of the benchmark database. Recommend at least 100GB. |
| | Query templates | Query templates used for training and test workloads. Recommend to perform the evaluation on the full set of query templates. |
| | Training query workload | Query instances used for training if applicable, including the number of query instances and the distribution of parameterization. |
| | Test query workload | Query instances used for test, including the number of query instances and the distribution of parameterization. Recommend testing with at least 10 query instances per query template. |
| | Duplicate ratio of the workload | The ratio of duplicate query instances in the combined set of training and test workload. Recommend having a duplicate ratio of less than 0.1, i.e., 10%. |
| Preparation stage | Time | The time spent on preparation. |
| | Resource | The resources spent on preparation. |
| Test stage | Average query elapsed time | The average elapsed time per query instance for the test query workload, including any overhead incurred to optimize and execute the query and adapt the system, such as query optimization time, model inference time, data collection, and model update. Can be normalized. |
| | Average query CPU time | The average CPU time per query instance for the test query workload, including any overhead incurred to optimize and execute the query and adapt the system, such as query optimization time, model inference time, data collection, and model update. Can be normalized. |
| | Percentile query elapsed time (recommended) | The distribution of query elapsed time for the test query workload. Can be normalized. |
| | Percentile query CPU time (recommended) | The distribution of query CPU time for the test query workload. Can be normalized. |
| | Snapshots of query performance (optional) | If the performance of the database system takes time to converge during the test stage, e.g., reinforcement learning based query optimizers, the performance can be reported periodically, e.g., per batch of test query instances. Can be normalized. |
| | Other overhead | Any additional overhead that is not captured by query elapsed time and CPU time, such as GPUs and FPGAs. |

workload. The training workload can be generated from a similar or different distribution compared with that of the test workload.

Since the parameter space of a query template is finite, there can be duplicate query instances generated in a workload. We call the ratio of duplicate query instances in a workload as the **duplicate ratio**. We recommend to generate a query workload with less than 0.1 duplicate ratio for all query instances used in the evaluation, i.e., the combined set of training and test workload. This ensures that the evaluation is performed on a test workload where the query instances are mostly distinct and unseen.

## 5.3 Preparation stage

The database system can spend additional resources on preparing itself before running any test queries. In the preparation stage, the system has access to the full database instance and the training workload. For example, a traditional database system can create histograms; and a workload-driven database system can collect execution statistics and learn models from the data and / or the training workload. The database system may decide to make changes to the physical design, e.g., creating indexes or materialized views. Such changes should be reported.

The time and resource spent on the preparation stage should be disclosed for comparison. The resource consumption can include CPU, memory, GPU, and others.

## 5.4 Test stage

The test stage is where the actual evaluation happens. In this stage, we run the queries in the test workload in isolation, either individually or concurrently. We measure the query performance by their elapsed time and CPU time over all the query instances in the test workload. Since some query instances are more expensive than others, which can overshadow the performance of less expensive query instances in the aggregated statistics, we recommend reporting percentile statistics of query performance, such as the distribution of query elapsed time and CPU time. The elapsed time and CPU time can be normalized.

The query elapsed time and CPU time should include any overhead incurred by processing the query. For example, for traditional database systems, the elapsed time and CPU time should include query optimization time; for ML-enhanced database systems, the elapsed time and CPU time should include the overhead of model inference if applicable. If a workload-driven database system needs to collect additional execution statistics and adapt to the workload, e.g., database systems with reinforcement learning or interleaved query optimization, the elapsed time and CPU time should include any other overhead from additional execution or adaptation. If the overhead cannot be included in the elapsed time or CPU time, e.g., GPUs or FPGAs, the source of such overhead should be disclosed, and the overhead should be measured and reported.

For database systems that can leverage execution feedback to improve their performance and converge over time, the evaluation can optionally report the aggregated query performance periodically, e.g., per batch of test query instances.

## 5.5 Example

We walk through an example of how to evaluate a traditional database system (DbT), a ML-enhanced database system with supervised learning based cardinality estimator (DbML), and a ML-enhanced database system with reinforcement learning (DbRL) to predict the query plan. We will describe a concrete case study of the evaluation in Section 7.

**Runtime environment:** The evaluation is performed on Azure D48ds v4 virtual machine instance, with 48 vCPUs, 192GiB RAM, and 1TB local SSD.

**Data and query generation:** We populate a 100GB DSB database with the physical design configuration as described in Section 3.3. We generate 100 query instances per query template as the training workload and 100 query instances per query template as the test workload for all the query templates using the default parameter distribution. The duplicate ratio is 0.05 in the combined set of training and test workload.

**Preparation stage:** The DbT creates additional single column statistics on the database based on the query templates. The time spent on creating the statistics and their storage consumption are reported. The DbML uses GPUs to train models based on the database and the training workload. The resources consumed, the time spent on collecting data labels and model training, and the size the resulting model are reported. The DbRL does not need to do any tuning upfront, so its resource consumption is zero.

**Test stage:** For each query, the DbT optimizes and executes the query. The elapsed time of a query using the DbT includes the query optimization and execution time. During query optimization, the DbML needs to predict the cardinality of each subplan expression while searching the plan space. Thus, the elapsed time of a query using the DbML includes query optimization with model inference and query execution. Finally, for DbRL, in addition to generating a query plan and executing the plan, it needs to collect query execution statistics and update its model. The elapsed time of a query using the DbRL includes the time on query optimization with model inference, query execution, data collection, and model update. Similarly, the CPU time of a query for each database system also includes all the computation incurred for processing a query. Since the DbRL takes time to converge to a good policy and reward function, we report both the aggregated query performance for the test workload, i.e., 3700 query instances, and the aggregated query performance every 20% of the test workload, i.e., a batch of 740 query instances.

## 6 EVALUATION OF DSB

In this section, we first quantify several important overall characteristics of the DSB benchmark and highlight the differences compared with the TPC-DS benchmark. Next, we compare the two benchmarks by their ability to generate distinct query instances. We then analyze the complexity of query optimization for the two benchmarks on both Microsoft SQL Server 2019 and Postgres 13 database systems. We further quantify the cardinality misestimates in the DSB and the TPC-DS benchmark w.r.t. independence assumption. We finally provide a detailed analysis of the room for improvement in plan quality for DSB on Microsoft SQL Server 2019.

We populate both DSB and TPC-DS database with the scale factor 100GB. We implement the physical design configuration on both databases as described in Section 3.3. The evaluation is performed on the same hardware as described in Section 5.5.

## 6.1 Overall statistics

We first compare the overall characteristics of the DSB and the TPC-DS benchmark (Table 2). We include the statistics of the derived SPJ queries from DSB for readers who are interested (see Section 4.5). Here, we highlight two differences:

- **Number of queries**: As shown in Table 2, the DSB benchmark consists of 37 query templates, including single-block queries with aggregates and group-bys (Agg) and multi-block queries (MultiBlock). We also derive 15 single-block SPJ queries from the DSB query templates (D-SPJ). In contrast, the TPC-DS benchmark has 99 query templates, where 59% of the query templates are multi-block queries and 2% are single-block SPJ queries. The DSB benchmark consists of a variety of query types, which is suitable for evaluating general database system performance as well as specific techniques for SPJ queries, e.g., join ordering.
- **Number of joins**: As shown in Table 2, the average number of joins in the DSB benchmark is higher than that of the TPC-DS benchmark, i.e., 10.8 vs. 8.1. We will further show that the query templates in the DSB benchmark are significantly more challenging for query optimization compared with these in the TPC-DS benchmark (Section 6.3).

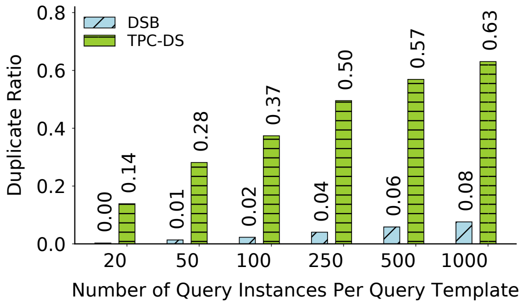
## 6.2 Duplicate ratio

We measure the capability of generating distinct query instances by measuring the duplicate ratio of the query workloads. Since deciding the equivalence of SQL queries is NP-hard, we consider a query instance distinct if there is no other query instance that has the same query text in the workload after applying our technique to reduce generating duplicate query instances (Section 4.4). Figure 1 shows the duplicate ratio of DSB and TPC-DS benchmark varying the number of query instances populated per query template, both with the default parameter distribution. With 20 query instances per query template, the DSB benchmark has 0.3% duplicate query instances, while the TPC-DS benchmark already has 14% duplicate query instances. The duplicate ratio increases with the number of query instances generated per query template. With 1000 query

Table 2: Summary statistics of DSB and TPC-DS benchmark

| Statistics | DSB | | | | | | TPC-DS | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | All | SPJ | Agg | MultiBlock | New | D-SPJ | All | SPJ | Agg | MultiBlock |
| Query template # | 37 | 0 | 15 | 22 | 3 | 15 | 99 | 2 | 39 | 58 |
| Average join # | 10.8 | 0 | 6.5 | 13.7 | 9.0 | 6.4 | 8.1 | 4.5 | 5.3 | 10.5 |
| Data distribution | Skewed and correlated | | | | | | Mostly uniform and independent | | | |
| Query workload generation | Default or configurable Gaussian distribution | | | | | | Limited variations | | | |

Figure 1: Duplicate ratio varying the number of query instances with the default parameter distribution



instances per query template or 37,000 query instances in total, which can be demanded as a training query workload for some workload-driven database systems, the DSB benchmark still maintains a low percentage of duplicate query instances, i.e., 8%. In contrast, the duplicate ratio of the TPC-DS benchmark shoots up to 63%, making it difficult to construct either distinct test query instances or disjoint training and test query workload. Our query template enhancement, as described in Section 4, effectively reduces the duplicate ratio by $8\times - 47\times$, which provides sufficient training and test query workload for workload-driven database systems.

## 6.3 Complexity of query optimization

We analyze the complexity of query optimization for the DSB and TPC-DS benchmark by optimizing the queries in a database system.

With the same database system, a query can result in more time of query optimization if the query is more complex, i.e., a larger search space or no obvious good plan based on heuristics. Thus, a benchmark can be more challenging for the database system if the query optimization time is higher.

We populate a query workload for each benchmark from the default parameter distribution with 20 query instances per query template. Table 3 shows the average query optimization time for the DSB and the TPC-DS benchmark normalized by the same constant. On Microsoft SQL Server 2019, the optimizer spends 2.7× time in query optimization with the DSB queries compared with that of the TPC-DS queries. In particular, the three newly added query templates in the DSB benchmark require the highest query optimization time. The overall result is similar on Postgres 13, where the normalized average query optimization time on the DSB benchmark is 2.9× of that on the TPC-DS benchmark. We observe that the MultiBlock queries take less time to optimize than other query types on Postgres 13 due to the greedy-based join ordering heuristics that often kick in for MultiBlock queries.

The query optimizer will produce different query plans for query instances of the same query template if different parameterization leads to different selectivities and thus different optimal plans. The larger the number of distinct optimal plans, the more diverse the selectivity is for different parameterization, at least from the optimizer's estimates.

Table 4 shows the average number of distinct query plans per query template for the DSB and the TPC-DS benchmark. On Microsoft SQL Server 2019, among the 20 query instances, query templates in the TPC-DS benchmark only have 1.5 distinct query plans in average, while query templates in the DSB benchmark have an average number of 7.9 distinct query plans, which is 5.3× of that for the TPC-DS benchmark. We observe similar results in Postgres 13: 4.9 distinct plans per query template on the DSB benchmark vs. 2.5 distinct plans per query template on the TPC-DS benchmark.

We further break down the impact of our enhancement on the complexity of query optimization:

- **DSB queries with TPC-DS data (*TPC-DS Data*)**: This is the benchmark with the database populated from the TPC-DS benchmark and the query workload from the DSB benchmark. Compared with DSB, this variation lacks the additional skews and correlations introduced to the database.
- **TPC-DS queries with DSB data (*DSB Data*)**: This is the benchmark with the database populated from DSB and the query workload from the TPC-DS benchmark. Compared with DSB, this variation lacks the enhancement of the query templates. Note that the three newly added query templates are not included in this variation.

**Impact of data enhancement**: Table 3 shows that DSB spends 21% more time on average in query optimization compared with *TPC-DS Data* on Microsoft SQL Server 2019. Similarly, Table 4 shows that DSB produces 6.1× number of distinct plans per query template compared with *TPC-DS Data* on Microsoft SQL Server 2019. Thus, adding more data skews and correlations significantly increases the complexity of query optimization. The trend is similar on Postgres 13.

**Impact of query template enhancement**: Table 3 shows that DSB spends 46% more time on average in query optimization compared with *DSB Data* on Microsoft SQL Server 2019. Similarly, Table 4 shows that DSB produces 3.0× number of distinct plans per query template compared with *DSB Data* for Microsoft SQL Server 2019. Thus, enhancing the query templates also dramatically increases the complexity of the benchmark. The trend is similar on Postgres 13.

**Table 3: Average normalized compile time per query instance with 20 query instances per query template**

| Database and workloads | Microsoft SQL Server 2019 | | | | | | Postgres 13 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | All | SPJ | Agg | MultiBlock | New | D-SPJ | All | SPJ | Agg | MultiBlock | New | D-SPJ |
| DSB | **0.51** | N/A | 0.49 | 0.53 | 1.00 | 0.42 | **0.23** | N/A | 0.34 | 0.15 | 1.00 | 0.28 |
| TPC-DS | **0.19** | 0.06 | 0.12 | 0.26 | N/A | N/A | **0.08** | 0.10 | 0.09 | 0.07 | N/A | N/A |
| DSB queries w/ TPC-DS data | 0.42 | N/A | 0.28 | 0.51 | 0.65 | 0.23 | 0.21 | N/A | 0.31 | 0.14 | 0.90 | 0.25 |
| TPC-DS queries w/ DSB data | 0.35 | 0.25 | 0.40 | 0.38 | N/A | N/A | 0.09 | 0.10 | 0.10 | 0.08 | N/A | N/A |

**Table 4: Average number of distinct query plans per query template with 20 query instances per query template**

| Database and workloads | Microsoft SQL Server 2019 | | | | | | Postgres 13 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | All | SPJ | Agg | MultiBlock | New | D-SPJ | All | SPJ | Agg | MultiBlock | New | D-SPJ |
| DSB | **7.9** | N/A | 5.9 | 9.3 | 5.0 | 5.5 | **4.9** | N/A | 4.7 | 5.1 | 5.7 | 5.4 |
| TPC-DS | **1.5** | 1.0 | 1.2 | 1.7 | N/A | N/A | **2.5** | 1.5 | 1.3 | 3.5 | N/A | N/A |
| DSB queries w/ TPC-DS data | 1.3 | N/A | 1.2 | 1.3 | 1.0 | 1.1 | 3.6 | N/A | 3.5 | 3.7 | 6.5 | 4.1 |
| TPC-DS queries w/ DSB data | 2.6 | 1.9 | 2.3 | 3.0 | N/A | N/A | 2.9 | 2.0 | 1.4 | 3.9 | N/A | N/A |

**Table 5: Percentile of q-errors in cardinality estimation w.r.t. independence assumption**

| % | GB1 | | GB2 | | GB3 | | GB4 | |
|---|---|---|---|---|---|---|---|---|
| | TPC-DS | DSB | TPC-DS | DSB | TPC-DS | DSB | TPC-DS | DSB |
| 25 | 1 | 1.3 | 1.0 | 1.3 | 1.0 | 1.7 | 1.0 | 2.3 |
| 50 | 1.1 | 1.6 | 1.0 | 3.5 | 1.0 | 2.5 | 1.0 | 4.5 |
| 75 | 1.1 | 2.1 | 1.0 | 4.8 | 1.1 | 3.7 | 1.1 | 10.9 |
| 90 | 1.2 | 2.7 | 1.1 | 4.8 | 1.1 | 8.2 | 1.1 | 31.9 |
| 95 | 1.6 | 3.2 | 1.1 | 4.9 | 1.1 | 13.1 | 1.1 | 73.9 |
| Max | 198.2 | 211.7 | 1.3 | 4.9 | 1.4 | 818.6 | 3.9 | 2213.7 |

## 6.4 Cardinality misestimates

The accuracy of cardinality estimation in a database system is crucial for the plan quality. Intuitively, if the data distribution leads to more cardinality misestimates, it is more difficult for a database system to produce a good plan. We evaluate and compare the degree of cardinality misestimates in DSB and TPC-DS by calculating the q-errors [21] w.r.t. independence assumption. We calculate the joint probability of all the value combinations from multiple columns of a single table or of the join result of multiple tables. Then we estimate the cardinality with the product of marginal probabilities of the values in the corresponding tables assuming independence. We compute the q-errors between the actual cardinality and the estimates under independence assumption.

We evaluate the q-errors of cardinality estimates for subexpressions from the query templates of DSB. Table 5 shows the percentiles of the q-errors for a variety of subexpressions from the DSB query templates:

- GB1: *item* grouped by i_category and i_manager_id.
- GB2: The join result of *customer, customer_address*, and *customer_demographics* grouped by cd_marital_status and ca_state.
- GB3: The join result of *store_sales, item, customer*, and *customer_address* grouped by i_class_id and ca_state.
- GB4: The join result of *catalog_sales, item, customer, customer_address*, and *customer_demographics* grouped by i_category, cd_education_status, and ca_state.

Table 5 shows that the cardinality estimation has significantly more q-errors in DSB compared with that in TPC-DS. For median q-errors, DSB results in up to 4.5× q-errors as that from TPC-DS. For 90% q-errors, DSB results in up to 29.0× q-errors as that from TPC-DS. We also observe that the degree of q-errors is amplified after joining multiple tables.

Note that cardinality misestimates do not necessarily translate to suboptimality in plan quality, since the plan quality also depends on the complexity of the query template and the query execution engine of the database system (e.g., the plan quality may not be sensitive to certain misestimates due to techniques in query execution or "two wrongs make a right" [8, 10, 11, 17]).

## 6.5 Room for improvement in plan quality

We evaluate the room for improvement in plan quality on DSB by comparing the query performance of the original query plan (Original) and the best plan we manage to produce by injecting true cardinality during query optimization (Optimal), both on Microsoft SQL Server 2019. This indicates a lower bound of the room for improvement in plan quality on a commercial database.

We populate a workload with 20 query instances per query template using the default parameter distribution. Figure 2 shows the normalized elapsed time of Optimal compared with that of Original. On average, the best plans have 31% less elapsed time compared with the plans from the original Microsoft SQL Server 2019, indicating sufficient room for improvement in plan quality even with a mature commercial database. We also include the room for improvement (i.e., 41%) of the derived SPJ queries for the readers who are interested.

The improvement in plan quality also covers a wide range of query instances and query templates. Figure 3 breaks down the distribution of the elapsed time speedup ratio using Optimal compared with Original. For 25% of the queries, Optimal speeds up the elapsed time by at least 1.2×. For 10% of the queries, the speedup is

**Figure 2: Normalized elapsed time using best plans**

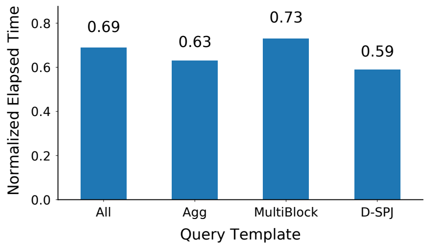


**Figure 3: Elapsed time speedup ratio using best plans**

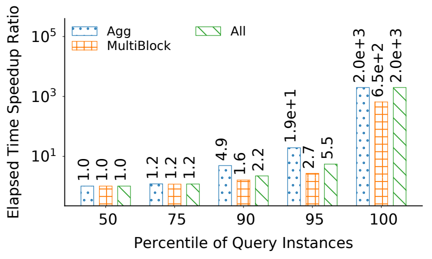
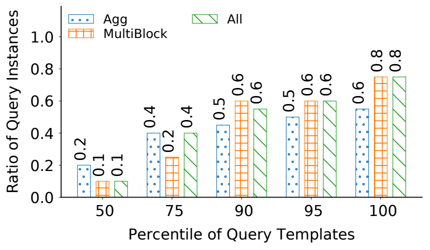


**Figure 4: Ratio of improved queries in elapsed time using best plans**



at least 2.2×, with a maximal speedup of more than three orders of magnitude. Figure 4 shows the distribution of the number of query instances per query template that get improved by at least 20% in elapsed time. For 50% of the query templates, at least 10% of the query instances have significant improvement in elapsed time. For 10% of the query templates, at least 60% of the query instances have significant improvement in elapsed time.

Our evaluation shows that there is significant room for improvement in plan quality on DSB both in aggregation and for a wide range of query instances and query templates.

## 7 EVALUATION CASE STUDY

In this section, we present a case study of how to evaluate workload-driven and traditional database systems using the DSB benchmark. Our goal is to demonstrate the methodology of evaluating database systems using DSB, as described in Section 5. The evaluation is performed on Microsoft SQL Server 2019, where we implement a variant of the cardinality estimator using ML. This case study does not necessarily reflect the best implementation of the specific ML-enhanced cardinality estimator we use, nor the performance of the state-of-the-art ML-enhanced database systems.

### 7.1 Setup

We populate a DSB database with the scale factor 100GB. We implement the secondary B+ tree indexes as described in Section 3.3. In Microsoft SQL Server 2019, statistics are auto-created, including single column histograms and multi-column density vectors. We populate a training workload $W_{train}$ of 200 query instances per template, as well as a test workload $W_{test}$ with 20 query instances per query template, both using the default parameter distribution. In addition, we populate an alternative training workload $W'_{train}$ of 200 query instances per query template using a Gaussian distribution of (0, 2.0) for parameterization as described in Section 4.2. The duplicate ratio of our workloads is 0 for $W_{test}$, 0.03 for $W_{test} \cup W_{train}$, and 0.05 for $W_{test} \cup W'_{train}$.

The evaluation is performed on the hardware shown in Table 6. The queries are executed individually in isolation, i.e. at concurrency level of 1. Table 6 summarizes the setup.

We evaluate five variants of Microsoft SQL Server 2019. Our evaluation includes two variants of the traditional database systems:

- **Original:** The unmodified version of Microsoft SQL Server 2019, which uses Volcano / Cascades style query optimization.
- **NoPartAgg:** The original Microsoft SQL Server 2019 with transformation rules for partial aggregates push down disabled to simulate a database system without advanced query rewriting for aggregates. Compared with Original, NoPartAgg has a smaller plan search space for some aggregation queries.

We also evaluate three variants of workload-driven database systems by integrating a supervised ML-based cardinality estimator into Microsoft SQL Server 2019 [13]. Given a set of query templates, the ML-based cardinality estimator extracts expressions from the templates and builds models to predict the cardinality for these expressions. The technique uses a lightweight approach to estimate cardinality of SPJ expressions with range or point look-up predicate filters by leveraging supervised ML. And it trains a separate model for each SPJ expression. The technique incrementally generates training data for the ML models based on a given training workload (i.e., query templates or query instances), where the training data is labeled approximately and efficiently with sampling. We adapt [13] for the evaluation of the full DSB benchmark, where we use the technique for cardinality estimation of SPJ expressions and fall back to Original for unsupported expressions.[2]

We create three variants of the ML-enhanced database system based on the data used for training the models:

- **MLData:** The ML-enhanced cardinality estimator automatically generates appropriate training query instances based on the data in the database.
- **MLSame:** The ML-enhanced cardinality estimator uses the training query workload $W_{train}$ populated with the DSB benchmark to generate training data, where the parameter distribution of $W_{train}$ is the same as that of $W_{test}$. In this case, the parameterization uses the default parameter distribution.
- **MLDiff:** The ML-enhanced cardinality estimator uses the alternative training query workload $W'_{train}$ populated with the DSB benchmark to generate training data, where the parameter distribution of $W'_{train}$ is different from that of $W_{test}$, i.e., Gaussian distribution (0, 2) vs. the default parameter distribution.

### 7.2 Preparation stage

In the preparation stage, since Original and NoPartAgg do not create any additional statistics, their resource consumption is 0. The

[2]Our implementation of the technique has limitations in parsing complex predicate filters and optimizing time and space consumption of the technique. The case study focuses on demonstrating the evaluation process. The performance of the ML-enhanced optimizers in the study does not necessarily represent that of the technique's best implementation nor the state-of-the-art ML-enhanced query optimizers.

**Table 6: Summary of information and metrics in the evaluation case study**

| Category | Item | Optimizer | | | | | |
|---|---|---|---|---|---|---|---|
| | | Original | NoPartAgg | MLData | MLDiff | MLSame | Optimal |
| Runtime environment | Hardware | Azure D48ds v4 virtual machine instance, with 48 vCPUs, 192GiB RAM, and 1TB local SSD | | | | | |
| | Software | Microsoft SQL Server 2019 | | | | | |
| | Concurrency level | 1 | | | | | |
| Data and query generation | Database scale | 100GB with 56 secondary B+ tree indexes | | | | | |
| | Query templates | All | | | | | |
| | Training query workload | N/A | N/A | N/A | 200 query instances with default distribution | 200 query instances with Gaussian distribution | N/A |
| | Test query workload | 20 query instances with default distribution | | | | | |
| | Duplicate ratio of the workload | 0 | 0 | 0 | 0.03 | 0.05 | 0 |
| Preparation stage | Time | 0 | 0 | 55 hours w/ CPUs | 36 hours w/ CPUs | 36 hours w/ CPUs | N/A |
| | Resource | 0 | 0 | Model: 1.4GB | Model: 1.4GB | Model: 1.4GB | N/A |
| Test stage | Average query elapsed time (normalized) | 1 | 1.06 | 0.98 | 1.04 | 0.93 | 0.69 |
| | Average query CPU time (normalized) | 1 | 1.07 | 1.03 | 1.15 | 0.96 | 0.72 |
| | Other overhead | None | | | | | |

ML-enhanced database systems create samples from the database based on the expressions extracted from the query templates by the ML-based cardinality estimator. They then run training queries against the samples to collect execution statistics, with which the ML models are trained. These training queries are either automatically generated by the ML-enhanced database systems based on the expressions extracted from the query templates and the data in the database, i.e,. MLData, or they are provided as training query instances by the DSB benchmark, i.e., MLSame and MLDiff.

The overhead of the ML-enhanced databases comes from three sources: creating the samples, executing queries to collect statistics for labeling the data, and training the ML models. We set a time limit for creating samples and training the models to avoid excessive resource consumption. Most models are successfully trained to converge under this time limit. We observe that most of the time is spent on executing queries to collect training data. The space consumption of the ML models trained is 1.4GB. However, we observe that the technique can generate very large intermediate results while creating samples from join results, i.e., >650GB, because of the many-to-many joins among fact tables in the new query templates (Section 4.3). Since such sample creation will time out, the corresponding expressions are considered unsupported, and the ML-enhanced cardinality estimator will fall back to Original for cardinality estimation for these expressions. The ML models are trained with CPUs.

Table 6 summarizes the resources consumed by the five database systems in the preparation stage.

## 7.3 Test stage

In the test stage, we run query instances from $W_{test}$ on the five variants of Microsoft SQL Server 2019 and measure their performance. Table 6 summarizes the performance metrics as specified in Section 5. The query execution statistics, i.e., elapsed time and CPU time, includes the query optimization time, query execution time, and model inference time if applicable. The model inference is performed on CPUs. Since the three ML-enhanced variants do not perform additional model adaption during the test stage, we only report the aggregated query performance. The Optimal represents the performance of the best plans as described in Section 6.5, and it is used for reference purpose only.

Figure 5 shows the normalized elapsed time of the test workload breaking down by query types. Overall, the best ML-enhanced database system, i.e., MLSame, is 7% faster in query elapsed time than Original. Among the three ML-enhanced database systems, MLDiff, which is trained with a query workload populated from a different parameter distribution than that of the test query workload, shows 4% slower query elapsed time and performs the worst. The variant of the traditional database systems, NoPartAgg, which uses a simpler plan search space without partial aggregates push down, shows 6% slower query elapsed time and performs the worst among all the database systems in the evaluation.

We break down the queries by their degree of improvement and regression for the best performed ML-enhanced variant, i.e., MLSame. Figure 6 and Figure 7 show the speedup and slowdown ratios in query elapsed time with MLSame. For 10% of the query instances, MLSame leads to significant reduction in query elapsed

**Figure 5: Normalized average elapsed time for all queries by query types**

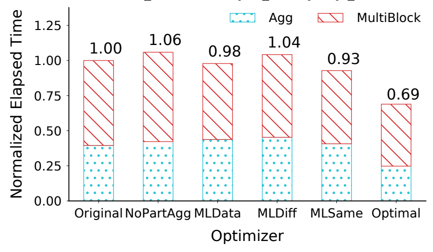


**Figure 6: Elapsed time speedup ratio with MLSame**

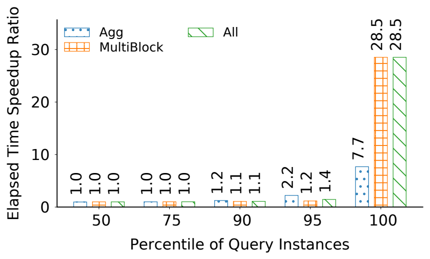


**Figure 7: Elapsed time slowdown ratio with MLSame**

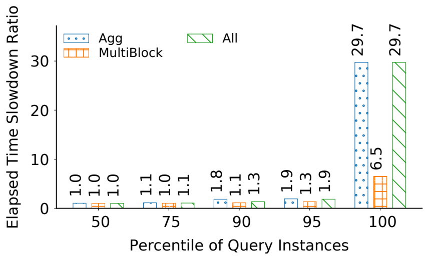


**Figure 8: Normalized average elapsed time for top 20 most expensive query templates**

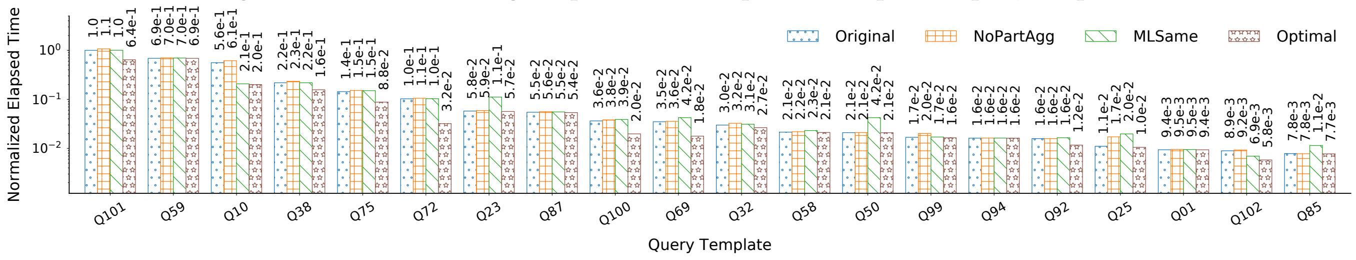
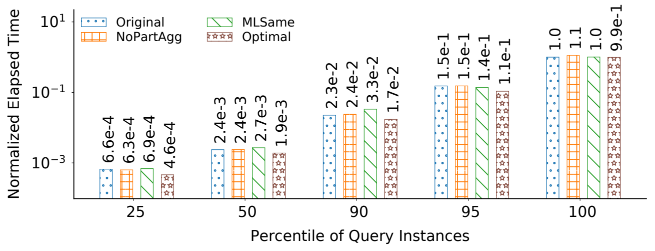


**Figure 9: Distribution of normalized elapsed time over test query instances**



time. In particular, for 5% of the query instances, the MLSame speeds up the query elapsed time by more than 1.4×, with the maximal speedup ratio of 28.5. However, for 5% of the query instances, the MLSame slows down the query elapsed time by more than 1.9×, with the maximal slowdown ratio of 29.7.

Figure 8 shows the normalized average query elapsed time aggregated per query template for the top 20 most expensive query templates on the original Microsoft SQL Server 2019. We also observe both improvement and regression with MLSame for the expensive queries. Figure 9 shows the percentiles of the query elapsed time for all test query instances. We observe that the query performance of Optimal improves over Original across all the percentiles.

The trends for CPU time are similar to those of the elapsed time, and the results are omitted due to space limit.

## 8 CONCLUSION

We propose a new benchmark, DSB, to evaluate both workload-driven and traditional database systems on modern decision support workloads. The DSB benchmark can generate thousands of distinct query instances and dynamic workloads. To compare the performance of workload-driven and traditional database systems, we provide guidelines on evaluation methodology and metrics. We show that, compared with the TPC-DS benchmark, the DSB benchmark results in higher cardinality misestimates due to the skews and correlations in the data distribution. The queries in the DSB benchmark also lead to more query optimization complexity compared with these from the TPC-DS benchmark on both Microsoft SQL Server 2019 and Postgres 13. In our case study, we demonstrate how to evaluate workload-driven and traditional database systems using the DSB benchmark. Our case study shows that, there is significant room for improvement in plan quality with DSB on a mature commercial database. The query performance can be impacted by the accuracy of cardinality estimates, complex query rewriting rules, and adapting the database systems under workload shifts. In addition, our emphasis on end-to-end query performance aims to encourage innovations on workload-driven database systems with low maintenance and adaptation overhead. Finally, we expect that reporting the distribution of query performance comparison can draw attention to both query performance improvements and regressions.

## REFERENCES

[1] [n.d.]. *The DSB benchmark.* https://aka.ms/dsb, last accessed on 2021-09-10.
[2] [n.d.]. *IMDB dataset.* https://www.imdb.com/interfaces/, last accessed on 2021-09-10.
[3] [n.d.]. *A parallel zipf-skewed data generator for TPC-H benchmark.* https://github.com/SrikanthKandula/tpch_dbgen_zipf_skew, last accessed on 2021-09-10.
[4] [n.d.]. *TPC-H data generation with skew.* https://www.microsoft.com/en-us/download/details.aspx?id=52430, last accessed on 2021-09-10.
[5] Ashraf Aboulnaga and Surajit Chaudhuri. 1999. Self-Tuning Histograms: Building Histograms without Looking at Data. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD '99).* Association for Computing Machinery, New York, NY, USA, 181–192. https://doi.org/10.1145/304182.304198

[6] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. 2005. Database Tuning Advisor for Microsoft SQL Server 2005: Demo. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*. Association for Computing Machinery, New York, NY, USA, 930–932. https://doi.org/10.1145/1066157.1066292

[7] Peter Boncz, Angelos-Christos Anatiotis, and Steffen Kläbe. 2017. JCC-H: Adding join crossing correlations with skew to TPC-H. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 103–119.

[8] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. 2015. Smooth Scan: Statistics-oblivious access paths. In *2015 IEEE 31st International Conference on Data Engineering*. 315–326. https://doi.org/10.1109/ICDE.2015.7113294

[9] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. 2011. The Mixed Workload CH-BenCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems (DBTest '11)*. Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages. https://doi.org/10.1145/1988842.1988850

[10] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2007. *Adaptive query processing*. Now Publishers Inc.

[11] Bailu Ding, Surajit Chaudhuri, and Vivek Narasayya. 2020. Bitvector-Aware Query Optimization for Decision Support Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2011–2026. https://doi.org/10.1145/3318464.3389769

[12] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1241–1258. https://doi.org/10.1145/3299869.3324957

[13] Anshuman Dutt, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2020. Efficiently Approximating Selectivity Functions Using Low Overhead Regression Models. *Proc. VLDB Endow.* 13, 12 (July 2020), 2215–2228. https://doi.org/10.14778/3407790.3407820

[14] Andrey Gubichev and Peter Boncz. 2014. Parameter Curation for Benchmark Queries. In *6th TPC Technology Conference on Performance Evaluation and Benchmarking*. Springer/Verlag, 113–129.

[15] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. *CIDR* (2019).

[16] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[17] Guy Lohman. 2014. Is query optimization a "solved" problem. In *Proc. Workshop on Database Query Optimization*, Vol. 13. 10.

[18] Lin Ma, Bailu Ding, Sudipto Das, and Adith Swaminathan. 2020. Active Learning for ML-Enhanced Database Systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 175–191. https://doi.org/10.1145/3318464.3389768

[19] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data*. Association for Computing Machinery, New York, NY, USA, 1275–1288. https://doi.org/10.1145/3448016.3452838

[20] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (July 2019), 1733–1746. https://doi.org/10.14778/3342263.3342646

[21] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 982–993. https://doi.org/10.14778/1687627.1687738

[22] Mark EJ Newman. 2005. Power laws, Pareto distributions and Zipf's law. *Contemporary physics* 46, 5 (2005), 323–351.

[23] Meikel Poess and Chris Floyd. 2000. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Rec.* 29, 4 (Dec. 2000), 64–71. https://doi.org/10.1145/369275.369291

[24] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. 2002. TPC-DS, Taking Decision Support Benchmarking to the next Level. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 582–587. https://doi.org/10.1145/564691.564759

[25] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *Proc. VLDB Endow.* 14, 1 (Sept. 2020), 61–73. https://doi.org/10.14778/3421424.3421432