LST-Bench: Benchmarking Log-Structured Tables in the Cloud

Jesús Camacho-Rodríguez
jesusca@microsoft.com
Microsoft
USA

Ashvin Agrawal
ashvin.agrawal@microsoft.com
Microsoft
USA

Anja Gruenheid
anja.gruenheid@microsoft.com
Microsoft
Switzerland

Ashit Gosalia
ashit.gosalia@microsoft.com
Microsoft
USA

Cristian Petculescu
cristp@microsoft.com
Microsoft
USA

Josep Aguilar-Saborit
jaguilar@microsoft.com
Microsoft
USA

Avrilia Floratou
avrilia.floratou@microsoft.com
Microsoft
USA

Carlo Curino
carlo.curino@microsoft.com
Microsoft
USA

Raghu Ramakrishnan
raghu@microsoft.com
Microsoft
USA

ABSTRACT

Data processing engines increasingly leverage distributed file systems for scalable, cost-effective storage. While the Apache Parquet columnar format has become a popular choice for data storage and retrieval, the immutability of Parquet files renders it impractical to meet the demands of frequent updates in contemporary analytical workloads. Log-Structured Tables (LSTs), such as Delta Lake, Apache Iceberg, and Apache Hudi, offer an alternative for scenarios requiring data mutability, providing a balance between efficient updates and the benefits of columnar storage. They provide features like transactions, time-travel, and schema evolution, enhancing usability and enabling access from multiple engines. Moreover, engines like Apache Spark and Trino can be configured to leverage the optimizations and controls offered by LSTs to meet specific business needs. Conventional benchmarks and tools are inadequate for evaluating the transformative changes in the storage layer resulting from these advancements, as they do not allow us to measure the impact of design and optimization choices in this new setting.

In this paper, we propose a novel benchmarking approach and metrics that build upon existing benchmarks, aiming to systematically assess LSTs. We develop a framework, LST-Bench, which facilitates effective exploration and evaluation of the collaborative functioning of LSTs and data processing engines through tailored *benchmark packages*. A package is a mix of use patterns reflecting a target workload; LST-Bench makes it easy to define a wide range of use patterns and combine them into a package, and we include a baseline package for completeness. Our assessment demonstrates the effectiveness of our framework and benchmark packages in extracting valuable insights across diverse environments. The code for LST-Bench is open source and is available at https://github.com/microsoft/lst-bench/.

1 INTRODUCTION

Parquet [63] and ORC [61] are widely popular columnar file formats designed to optimize data storage and retrieval, with a bias toward read-heavy workloads. These files are designed to be immutable: once created, they are read-only and optimized for efficient columnar reads. Modern analytical workloads, however, require frequent incremental updates to structured data, i.e., tables, in small batches in order to expedite insights and maximize the business value derived from data. Log-Structured Tables (LSTs) effectively cater to this requirement while leveraging the inherent strengths of columnar formats. They have become the industry standard and pervasively adopted in the field. Several implementations of LSTs have emerged, with Delta Lake [4, 20], Apache Iceberg [37], and Apache Hudi [28] being the most widely adopted ones. These LSTs add a metadata layer on top of immutable columnar files to represent versions of tables and specify how data processing engines and applications interact with them[1].

LSTs represent a significant paradigm shift in the storage layer design from traditional warehouse systems. Unlike traditional systems that manage their own storage [2, 60, 71], LSTs rely on non-POSIX APIs provided by object stores [3, 26, 52, 62] to enable their features, which can be shared across compute engines. LSTs use column-oriented log-structured immutable files instead of the row-oriented in-place updated page files used by traditional OLTP and warehouse database systems [1]. LSTs are designed specifically for processing large-scale data that receives continuous trickle updates [9], which, while not as frequent as in OLTP-style workloads, differ markedly from the infrequent, large-scale batch updates that traditional warehouses doing in-place updates are designed for [45, 76]. They provide single-table ACID transactions[2] using *multi-version concurrency control* [77], creating a new version of tables by 'depositing' a new immutable layer of files containing changes made to the dataset. This approach also enables features such as *time travel* queries. However, one drawback of having multiple layers of version files is the increase in IO operations required to retrieve data for query processing, which can result in slower query execution [14]. To tackle this issue and achieve desired outcomes, LSTs offer various controls, optimizations, and algorithms, including compaction, file sizing, clustering, and caching.

[1]While these implementations do not yet cover security and access control, we believe that they need to do so; in practice, most systems using them add a security layer.
[2]Multi-table transactions are a notable gap in comparison to traditional database systems; we expect this will also be addressed in future.

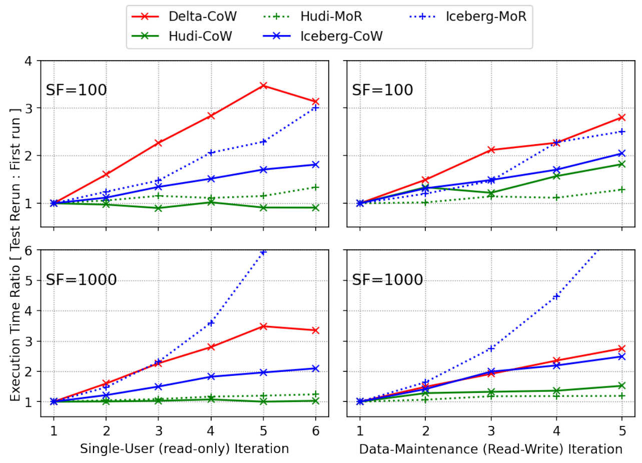


Figure 1: Execution time comparison of TPC-DS *single user* & *data maintenance* test iterations at different scale factors (SF=100, SF=1000) using various LSTs and strategies such as Copy-on-Write (CoW) and Merge-on-Read (MoR) on Spark.

1.1 Challenges in Benchmarking LSTs

The differences between LSTs and traditional warehouse systems introduce unique new challenges for users to learn how to operate and tune LSTs. Benchmarking is the go-to methodology to learn the characteristics of these new systems. However, there has been limited research on developing innovative evaluation mechanisms for LSTs aimed at formalizing the complexities of continuously changing performance in long-running deployments or the effect of concurrently running (and often expensive) maintenance operations on files stored on object stores.

As a result, users have been forced to rely solely on previously established benchmarks. Current evaluations typically rely on TPC-DS [57], which has long been the standard OLAP benchmark, and involve running a limited number of queries or using handcrafted queries to test a variety of operations [12, 44]. We observe that these evaluations suffer from two main limitations that restrict their ability to provide useful insights for LSTs. Firstly, they fail to uncover hidden characteristics inherent in LSTs and data processing engines that are crucial in real-world usage scenarios. Secondly, they lack specific evaluation metrics that are important for LSTs, such as performance degradation over time. Our goal is to propose a framework that complements a base workload such as TPC-DS to address these limitations.

Evaluation Scenarios. Current benchmarking workloads, such as TPC-DS, reflect a generalized understanding of OLAP tasks and do not consider characteristics such as (*i*) *longevity*, which involves handling frequent data modifications over a long period of time; (*ii*) *resilience*, which involves handling multiple data modifications of varying sizes in a regularly optimized table; (*iii*) *read/write concurrency*, which involves handling multiple sessions reading and writing data simultaneously, potentially using multiple compute clusters; or (*iv*) *time travel*, which involves querying data at different points in time. However, these characteristics are crucial for LSTs, which vary from traditional warehousing systems in building on immutable files and relying heavily on versions and mechanisms for compaction, version control, etc., and also in how they are deployed by customers interested in new features such as time travel.

Example 1.1. As we mentioned previously, performance degradation due to accumulation of version files over time is an important evaluation factor for LSTs. The standard TPC-DS workload involves two rounds of interleaved read and write queries, typically reported together [57]. Instead, we conducted experiments with an increased number of TPC-DS *single user* (read-only) query iterations, alongside *data maintenance* (read-write) steps between individual *single user* iterations. Figure 1 depicts the results, demonstrating consistent execution time deterioration for both *single user* and *data maintenance* tests (details in §5.1). More importantly, by analyzing the results, we can observe interesting trends beyond the second iteration, which would be overlooked if relying solely on the original TPC-DS workload.

Performance Metrics. Similar to other benchmarks [73], the TPC-DS specification focuses on a primary metric, namely queries per hour for decision support (QphDS), which provides a single performance measurement that is used for the comparison of systems. While this approach simplifies the ranking of multiple systems, it fails to capture essential dimensions that are relevant for evaluating LSTs. For instance, understanding the performance and efficiency degradation of LSTs over time is crucial to determining how system designers can effectively optimize platforms that rely on LSTs.

Example 1.2. The results of executing TPC-DS[3] (scale factor 1000) with various LSTs and Spark are presented in Table 1. However, *QphDS* does not capture the performance degradation of the second run relative to the first (i.e., the increase in latency), depicted in the 'Inter-test Degradation' column, which shows that well-performing LSTs can significantly degrade over time. These numbers indicate that running these LSTs without appropriate mediation will result in low-performance results.

Table 1: Latency increase between TPC-DS test iterations.

LST	Throughput-QphDS	Inter-test Degradation
Delta	511K	2.7 -> 5.2 hrs (**92%**)
Hudi-CoW†	262K	6.2 -> 6.5 hrs (**5%**)
Hudi-MoR‡	112K	23 -> 24 hrs (**6%**)
Iceberg-CoW†	549K	2.7 -> 4 hrs (**45%**)
Iceberg-MoR‡	493K	2.9 -> 5 hrs (**73%**)

† Copy-on-Write mode ‡ Merge-on-Read mode

Framework Flexibility. The overall performance and cost of LSTs are significantly influenced by the query engine and algorithmic components that orchestrate optimization tasks such as file clustering, small file compaction, caching, and more. For instance, in §5.1, we show that read queries exhibit up to an 85% improvement in execution time when running on Trino compared to Spark, for both Delta and Iceberg. This difference stems mainly from Spark's default distribution mode, which leads to generation of a large number

[3]Standard dataset and execution rules were followed, but the audit step was not performed.

of small data files. Consequently, it is important that a benchmarking framework for LSTs enables detection and analysis of such variations. In the aforementioned case, we draw our conclusion by correlating the benchmark run with telemetry collected from the cloud storage infrastructure using our framework. Moreover, as LSTs are a new proposal with continuously expanding use cases, it is critical that the framework supports extension to new engines, datasets, and scenarios that go beyond traditional OLAP tasks.

1.2 Contributions

This paper presents a benchmarking framework to evaluate open-source LSTs and overcome the limitations of existing benchmarks. Our contributions are as follows:

Conceptual Model. We introduce a conceptual model that allows us to understand performance in terms of three dimensions: metadata representations, algorithms associated with LSTs and their operations, and engine characteristics (how they carry out certain operations and how they use the underlying LSTs) (§2). Our model provides insights into the factors that influence query performance in LSTs and highlights the surprising amount of commonality across different table formats. We hope this can help the community to develop a more unified approach to building, using and benchmarking these emerging table implementations.

LST-Bench: A Benchmarking Framework for LSTs. We design a benchmarking framework, LST-Bench, that focuses on key characteristics of LSTs and measures fundamental metrics for a thorough understanding of their relative performance in different scenarios (§3). Building upon TPC-DS, our benchmark proposes new extensions, relevant to LSTs, and offers the ability to define *packages* of workload patterns, inspired by real-world analytical workloads. We include a baseline package for completeness.

LST-Bench Implementation. We implement the framework, tying our novel benchmark and proposed metrics together (§4). It automates the process of running the workloads and collects the required telemetry from the engine and various cloud services to compute the metrics necessary for evaluation. LST-Bench is available as an open-source contribution[4].

Evaluation. We use LST-Bench to evaluate the performance, efficiency, and stability of out-of-the-box Delta Lake, Apache Iceberg, and Apache Hudi. Through our analysis of the results (§5), we provide insights into their strengths and weaknesses. Additionally, we conduct experiments using two widely adopted data processing engines, Apache Spark and Trino, and showcase their significant impact on the overall performance and efficiency of LSTs.

Our primary focus in this work is to develop a fair, comprehensive, and consistent framework for evaluating LSTs. While we provide insights into the current state of LSTs, it is important to note that different results could accrue for several reasons in practice: (*i*) using a different base workload or benchmark package, (*ii*) impact of overall system, including aspects such as security and access control, or (*iii*) impact of engine and its level of integration with the LST. We have designed LST-Bench as a modular, easy-to-extend framework and welcome contributions to our open-source codebase to enhance the framework and methodology proposed in this paper.

[4]https://github.com/microsoft/lst-bench/.

2 A CONCEPTUAL MODEL FOR LSTS

Delta Lake, Hudi, and Iceberg are rapidly evolving independently while sharing core characteristics, such as versioned table data stored as immutable files, a metadata layer describing the data, various controls and optimizations, and integration with popular engines. In this section, we propose a conceptual model that utilizes these common characteristics and represents an LST configuration along a three-dimensional space. Specifically, the first dimension pertains to the metadata layer representation introduced by each LST (§2.1), the second dimension encompasses algorithmic components and LST parameters for controls and optimizations (§2.2), and the third dimension focuses on the behavior of the data processing engines (§2.3). The performance and efficiency of query processing on LSTs are determined by the combination of these dimensions. In §5, we rely on this model to analyze and compare the observations, similarities, and differences among the various LSTs tested.

2.1 Metadata

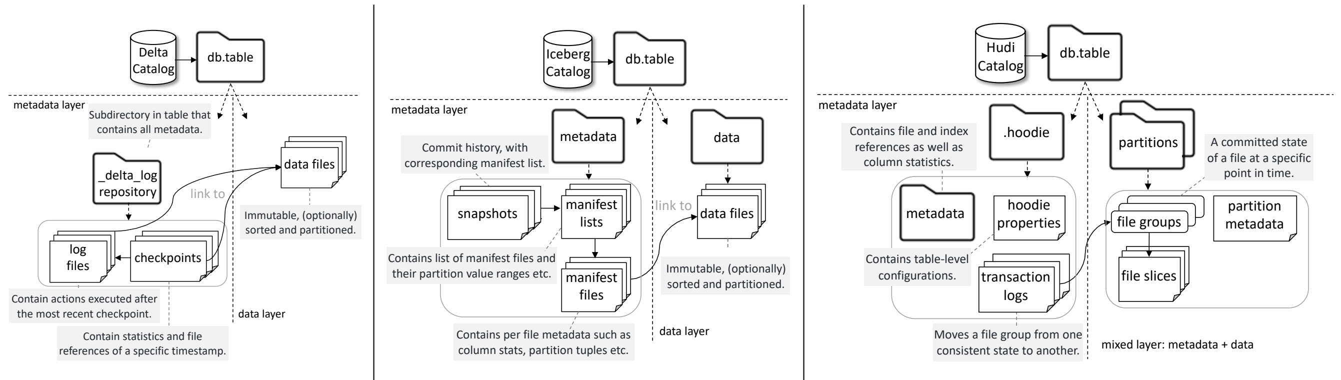
The metadata layer plays a crucial role in determining how engines interact with the format and serves as one of the key distinguishing components of LSTs[5]. LSTs store metadata within the corresponding file structure of a table. Specifically, these formats maintain a commit log of operations performed on each table, such as adding or removing data files, or modifying the schema. How the (meta)data files are modified may change depending on the LST, and the way the metadata is laid out in the storage system is a key factor that affects the performance and features of different LSTs. A summary of the various metadata layouts is included in Figure 2.

For each commit, Delta stores a log file identified by a monotonically increasing ID. This file includes an array of actions that were applied to the previous version of the table, along with statistics such as the minimum and maximum values for each column. The metadata subdirectory also includes *checkpoint files* that store non-redundant actions. These files are generated every 10 transactions by default and are referenced in the table metadata for quick access to the last checkpoint. In contrast, Iceberg takes a different approach by organizing files hierarchically to represent the table's state. The top-level structure consists of a *metadata file* that is replaced atomically whenever changes occur. This file contains references to *manifest list files*, each of which represents a snapshot of the table at a specific point in time. The manifest list files, in turn, reference *manifest files*, which track the data files and provide statistics about them. Lastly, Hudi creates a *timeline* by storing the actions performed on the table as files identified by their start commit time. It also uses a nested metadata table [27], which is a Hudi table itself, to store physical file paths and indexed files that belong to the table, thereby enabling efficient file pruning.

2.2 Algorithms

Implementations of LSTs incorporate various algorithms and configurations that give rise to different behaviors. These algorithms

[5]Notably, translating between different metadata representations enables representing one LST as another without costly data rewriting [13, 56].


Delta Catalog
db.table
metadata layer
Subdirectory in table that contains all metadata.
_delta_log repository
data files
link to
Immutable, (optionally) sorted and partitioned.
log files
checkpoints
Contain actions executed after the most recent checkpoint.
Contain statistics and file references of a specific timestamp.
data layer
Iceberg Catalog
db.table
metadata layer
metadata
data
Commit history, with corresponding manifest list.
snapshots
manifest lists
link to
data files
Contains list of manifest files and their partition value ranges etc.
manifest files
Immutable, (optionally) sorted and partitioned.
Contains per file metadata such as column stats, partition tuples etc.
data layer
Hudi Catalog
db.table
metadata layer
.hoodie
partitions
Contains file and index references as well as column statistics.
A committed state of a file at a specific point in time.
metadata
hoodie properties
file groups
partition metadata
Contains table-level configurations.
transaction logs
file slices
Moves a file group from one consistent state to another.
mixed layer: metadata + data

(a) Delta (b) Iceberg (c) Hudi

Figure 2: File layouts for LSTs under study.

and configurations encompass protocols for engine interaction, impacting concurrency and isolation guarantees, as well as configuration parameters that influence higher-level aspects like the data layout within the table, with significant implications for query performance. Additionally, LSTs introduce diverse algorithms for bookkeeping and cleanup operations. Importantly, all these aspects are accompanied by configuration parameters that allow for fine-tuning the behavior of a particular LST implementation.

2.2.1 Concurrency and Locking. LSTs leverage multi-version concurrency control (MVCC) [23, 58, 69] to allow for concurrent transactions to access the same data in the storage system without interfering with each other. This is achieved by creating multiple versions or *snapshots* of the same logical data.

All three LSTs allow transactions to be executed within the context of a single table by implementing optimistic concurrency control [21, 34, 38]. However, they differ in their approaches to conflict resolution and the level of isolation they provide. Hudi and Iceberg support *snapshot isolation*, which means that even if other transactions are concurrently modifying the table, a transaction reads a consistent snapshot of the data as it existed at the start of the transaction. In turn, Iceberg and Delta provide a stricter isolation level by default, which guarantees that writes to the table will occur in a serial order.

The lock management requirements, and consequently their implementation, differ across the three LSTs due to their different designs. Delta and Iceberg have minimal lock management needs and rely only on atomic *put-if-absent* or *rename* operations provided by the underlying object store or file system [4, 42]. Hudi has a greater reliance on locking, particularly when using the metadata table to track table files [30].

2.2.2 Data Layout Configuration. Since LSTs operate on the assumption that data files are immutable, they require mechanisms for updating and deleting rows. The two supported strategies are Copy-on-Write (CoW) and Merge-on-Read (MoR). CoW creates a new copy of the data files for each update or delete operation, while MoR writes changes to a separate file (often referred to as *delta file*) that is merged into the dataset during read operations. CoW is preferred for read-heavy workloads, while MoR is the preferred strategy for write-heavy workloads. Iceberg and Hudi currently support both CoW and MoR, while Delta only supports CoW[6].

2.2.3 Table Maintenance Operations. LSTs, similar to previous MVCC implementations [77], provide operations that ensure the data stored in a table is optimized and efficient. These operations include (*i*) *compacting* data files within a table, which consolidates smaller files into larger ones and optionally sorts the data based on specific column values, reducing metadata overhead and improving query performance, and (*ii*) *vacuuming* data files within a table, which deletes expired data files after the retention period or ones that are no longer referenced by the table metadata because they are deemed useless after the compaction process.

Engines relying on LSTs typically provide an API to perform these maintenance operations on-demand. For instance, commands to execute them are often available to users as SQL extensions [22] or stored procedures [32, 41]. LSTs have different default settings for using these maintenance operations, for example, Hudi enables compaction and vacuum out-of-the-box while Iceberg and Delta require users to specify them. Users may execute these operations after modifying table data or schedule them to run automatically based on predetermined criteria such as the age or size of the data files. Additionally, some commercial platforms offer the automation of table maintenance [17, 59, 70].

2.3 Engines

Engines and their configurations have a significant impact on the performance of all LSTs. Clearly, the efficiency of engine internals directly affects the speed of data read and write operations to the LST. However, there are additional aspects related to engine configuration that may not be immediately apparent but have substantial implications for performance.

For example, the cluster configuration, parallelism settings, and the chosen execution plan by the engine can have effects beyond the execution of individual queries. Concretely, these factors can impact the fragmentation at the storage level, as they can influence the number of files generated by the engine during write operations (recall our discussion comparing the behavior of Spark and Trino

[6]MoR support is an upcoming feature in Delta Lake.

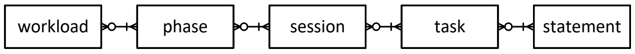


Figure 3: Workload components and their relationships.

in §1). This, in turn, has ripple effects on subsequent operations, affecting the performance of both reads and writes, including those executed during table maintenance operations.

3 LST-BENCH BENCHMARKING FRAMEWORK

TPC-DS [72] is widely used to evaluate decision support systems, covering various aspects such as data loading, query execution, sustained throughput, and data updates. Researchers and practitioners are also extensively using its dataset and workload to assess the efficiency of LSTs [12, 16, 44, 46], often measuring query latency. In this section, we propose a new benchmark for LSTs that builds upon TPC-DS as a "base workload", using its data set generator as well as its query set but modifying how the benchmark execution is structured. Specifically, we create several workload patterns that invoke tasks of the base workload along with other LST specific tasks such as compaction or time travel (§3.1). Furthermore, we propose new metrics intended to capture performance characteristics specific to LSTs (§3.2).

3.1 Workload Patterns

We divide the work of extending the base workload into two parts. First, we enhance the benchmark's customizability by proposing new tasks specific to LSTs, as well as making it easier to use existing ones to create custom workloads. This is discussed in detail in §3.1.1. Second, we utilize the proposed extensions to create a baseline *package* consisting of workload patterns useful to gain insights into LST aspects overlooked by the base workload, such as *stability*, *resiliency*, *read/write concurrency*, and *time travel*. These workload patterns are presented in §3.1.2. Before discussing these extensions in detail, we introduce the model employed for workload representation and, for completeness, briefly describe the original TPC-DS workload.

Workload Representation. Figure 3 depicts the components of a workload and their relationships. A *task* is a sequence of SQL *statements*, while a *session* is a sequence of tasks that represents a logical unit of work or a user session. A *phase* is a group of concurrent sessions that must be completed before the next phase can start. If a phase consists of a single task, we may refer to it interchangeably by its task name. Lastly, a *workload* is a sequence of *phases*.

We choose this flexible representation to ensure its adaptability to both standard and custom workloads prevalent in practice, offering a comprehensive solution. Although not all scenarios require all abstractions, our approach was driven by diverse and extensive user feedback, including (*i*) facilitating the mapping of existing workloads such as TPC-DS, (*ii*) aligning with the concept of sessions in JDBC, and (*iii*) ease of reusability. For instance, a session initiates a fresh connection to the engine, whereas a task simply groups a collection of SQL statements. Consequently, a workload designer can either execute multiple tasks in one session or initiate numerous concurrent sessions, each handling a subset of tasks.

W0. *Original TPC-DS.* According to these definitions, the TPC-DS benchmark executes multiple phases as shown in Figure 4a. These phases include (*i*) a Load phase where data is loaded into the tables used in the experiment, (*ii*) a Single User phase which runs a series of complex queries to determine the upper limit of the engine's performance, (*iii*) Throughput phases involve running multiple sessions in parallel, each executing a Single User task with a different permutation of the query set, to assess the engine's ability to handle multiple users and queries simultaneously, and (*iv*) Data Maintenance phases that are executed to test the engine's ability to handle data inserts and deletes.

3.1.1 *Workload Composability.*

Next we describe our extensions to enhance TPC-DS customizability. Note that although we will propose specific workload patterns in §3.1.2, our extensions offer flexibility for developing new patterns that can highlight characteristics that may have been overlooked in previous evaluations.

Configurable Sequence of Phases. As mentioned previously, the TPC-DS standard defines a strict sequence of phases that must be executed in a specific order. To evaluate specific aspects of LSTs, for example their longevity, we require a more flexible approach to the order of phases that is not captured by the standard sequence. For this reason, the sequence of phases that the benchmark executes should be configurable.

Ability to Run Multiple Tasks Concurrently within a Phase. TPC-DS sequences are linear, meaning that different tasks never overlap with each other (even though in the *throughput* phase multiple Single User tasks run in parallel). However, as other works have previously reported [5], a common use case for LSTs is querying the data while background operations, such as the incremental maintenance of downstream tables or materializations, are concurrently executing. Therefore, we want to be able to evaluate LSTs while running multiple, possibly different, tasks concurrently.

Optimize Task. Given that table maintenance operations are frequently executed concurrently with other queries, it is critical to include them in the evaluation of LSTs to determine whether they (*i*) can restore the LST to its initial non-degraded performance state, and (*ii*) impact the performance of other queries running concurrently. To address this, we introduce a new Optimize task that involves running *compaction* on LSTs. While LSTs offer various *compaction* strategies to optimize file layout and size, such as bin-packing or sorting, we opt to use the default strategy for each LST in our task definition. However, note that selecting an alternative strategy would be as straightforward as modifying the SQL associated with the task definition.

Time Travel Task. In §1.1, we mentioned that a benchmark should evaluate new features provided by LSTs, such as *time travel*, which allows querying of historical versions of a table based on timestamp or version. Therefore, we introduce a Time Travel task that executes the same queries as a Single User task but as of a given point in time.

3.1.2 *Baseline Package for Evaluation of LSTs.*

Based on these extensions, we propose a package consisting of four workload patterns

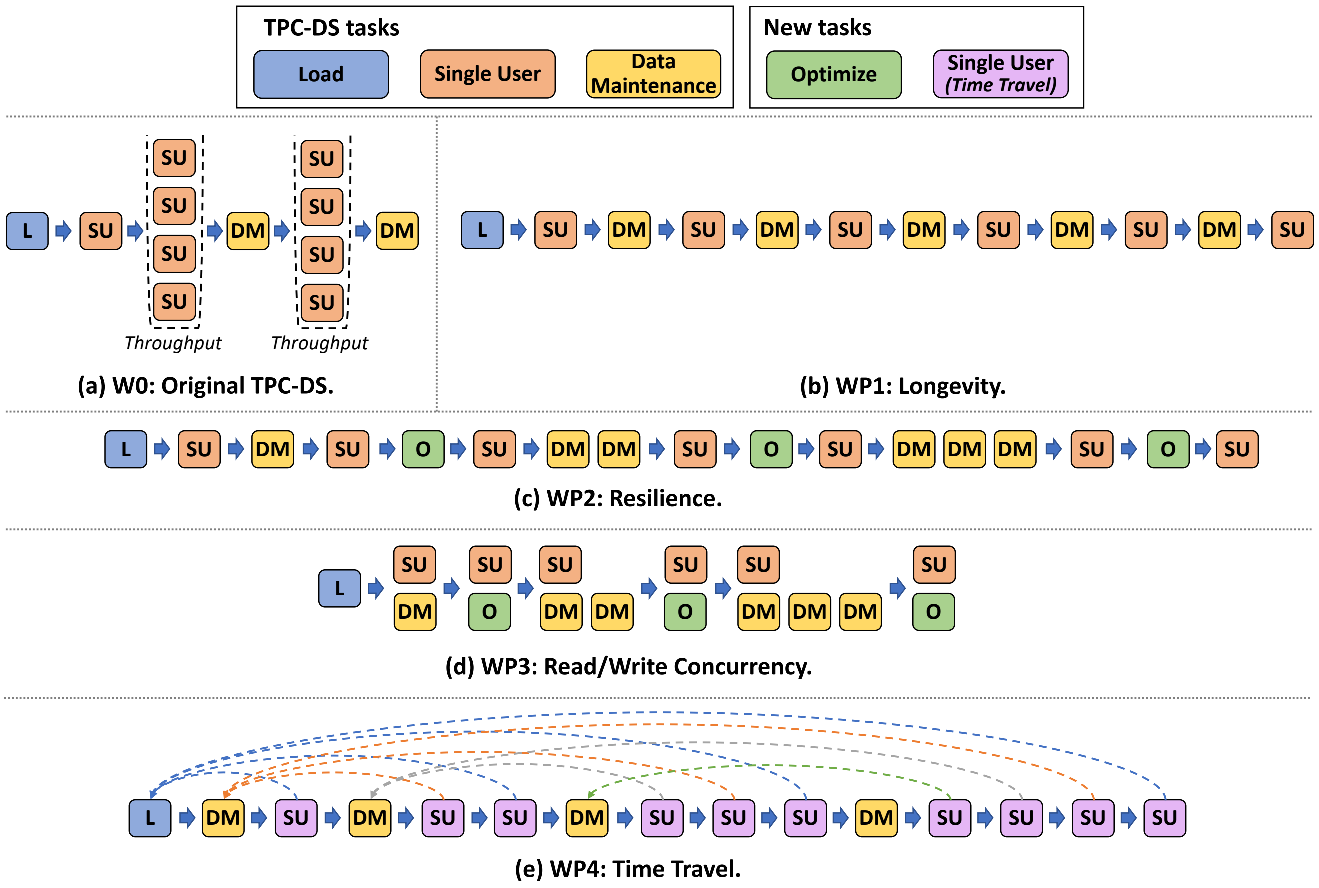


Figure 4: TPC-DS and extensions to evaluate LSTs characteristics.

to gain insights that cannot be obtained by executing the original TPC-DS workload. To design our experiments, we carefully selected various parameter values, including experiment length, based on empirical observations from customer workloads.

WP1. ***Longevity.*** This workload evaluates the performance, cost, and IO stability of LSTs over time. The proposed sequence is shown in Figure 4b. The experiment involves six Single User phases, each followed by a Data Maintenance phase to add new metadata and data to the table. Repeating this process multiple times allows us to observe how the LST behaves over time and identify significant trends, if there are any.

WP2. ***Resilience.*** This workload, shown in Figure 4c, evaluates the impact of table maintenance operations such as compaction on degradation over time. Each Optimize phase is executed subsequent to an increase in the number of write statements executed on the source tables, thus measuring the performance of Optimize operations as the ratio of refreshed data in a table increases.

WP3. ***Read/Write Concurrency.*** This workload evaluates the impact of the concurrent execution of read and write statements. As shown in Figure 4d, we run Single User phases concurrently with the Data Maintenance and Optimize phase respectively to simulate this scenario. Note that by leveraging the separation of storage and compute and the on-demand availability of cloud computing resources, our framework has the ability to run concurrent operations on separate compute clusters, which allows us to evaluate the storage layer's impact without the complication of interleaving these operations at the compute layer.

WP4. ***Time Travel.*** LSTs introduce *time travel*, which enables querying data at specific points in time by leveraging SQL extensions to specify the desired version of the table [35, 43, 66]. The workload shown in Figure 4e evaluates this new feature. We execute multiple Data Maintenance phases on the original data, followed by the same number of Time Travel phases, each executed on a version of the table produced by a previous Load or Data Maintenance phase.

Discussion. We could enhance our proposal through additional extensions, such as new tasks to cover dimensions that are still overlooked in the current proposal, e.g., *vacuum* operations, schema evolution [33, 36, 65], partition evolution [40], or deep and shallow table cloning [18]. Moreover, we could introduce tasks containing SQL statements that modify engine behaviors at the session level (e.g., Spark's *set* statements [68]), facilitating the evaluation of important engine features and configurations, including parallelism settings. Lastly, incorporating workload patterns consisting of a more concurrent, diverse, and intricate mix of tasks would further assist in evaluating LSTs. For instance, this could help to evaluate scenarios characterized by increased concurrency, which is common when dealing with LSTs, and diverse conflict resolution and isolation level configurations. It could also help to assess whether background operations like file consolidation and clustering, which typically run automatically and continuously in fully managed platforms without explicit invocation during a workload's execution, are indeed triggered and yielding their intended effects. Implementing these extensions into our framework is straightforward, and we plan to explore it in the future.

3.2 Metrics

This section explores metrics for a comprehensive and fair evaluation of LSTs unique features. We first discuss traditional metric categories applicable to LSTs, such as performance, and storage and compute efficiency (§3.2.1). We then introduce a stability metric that builds upon the aforementioned metrics to reveal crucial degradation characteristics of cloud data warehouses (§3.2.2). Our multi-metric approach draws from prior research [15, 73] and our own observations, and importantly, it can be easily extended to cover unexplored dimensions.

3.2.1 Traditional Metrics. In a cloud environment, several categories of metrics are important to consider when evaluating LSTs. First, data warehouse performance is traditionally evaluated using two measurements, latency, i.e., measuring the round-trip of queries, and throughput, i.e. , measuring the capacity of the system. Second, the interaction of a LST with the storage layer is an important aspect as LSTs specifically rely on cloud (object) storage. Unlike local disks, cloud IOPS are charged on a pay-as-you-go basis. This means that managing storage utilization is not the only factor to consider, but also total API operations and data transfers, and peak rates. Finally, a compute efficient LST achieves high performance while using a small amount of resources which is especially crucial on shared clusters. Key metrics include CPU utilization, memory utilization, and disk utilization, which measure the amount of CPU, memory, and local disk used for processing the workload, respectively.

Note that some of these (types of) metrics are captured as part of the TPC-DS standard. For example, QphDS is a throughput metric while load time, system availability and price per QphDS are additional metrics that contribute to a more comprehensive understanding of the evaluated systems.

3.2.2 Stability. LSTs are designed to receive continuous trickle updates, which, over time, can result in accumulation of delta files in the object store. Intuitively, the oftentimes smaller delta files degrade the system's efficiency as it causes the compute to consume additional resources to successfully execute a workload. The extent of the degradation depends on several factors, including the number of new files and data layout in the files. A well designed system is less susceptible to the degradation as more data updates are performed, or may have features to auto-mitigate adverse side effects of updates. To measure degradation, we introduce a new metric category, *stability*, which examines a system's ability to sustain its performance and efficiency (e.g., latency) consistently and

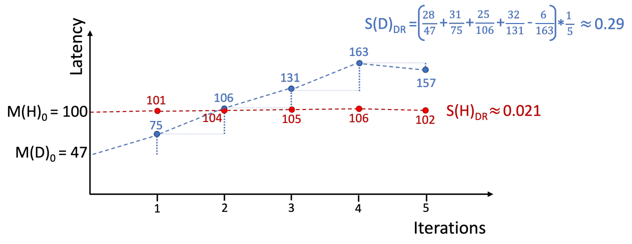


Figure 5: Example for *stability* computation.

exhibit minimal degradation. The process of calculating degradation involves dividing a workload's timeline into different phases, as described in §3.1, and then comparing the performance and efficiency measurements taken during each phase of the same type. For example, Single User phase SU-i would only be used for the computation of stability pertaining to Single User phase performance. We formally define the degradation rate below.

$$S_{DR} = \frac{1}{n} \sum_{i=1}^{n} \frac{M_i - M_{i-1}}{M_{i-1}} \tag{1}$$

where

- M_i is metric value of the i^{th} iteration of a workload phase,
- n is the number of iteration of the phase, and
- S_{DR} is the degradation rate.

Intuitively, S_{DR} is the rate at which a metric is growing or shrinking, due to the cumulative effects of changes in the underlying system's state. It provides information about how quickly a system degrades. A *stable* LST exhibits low S_{DR}. Note that M can be selected from the metrics introduced in §3.2.1; for metrics where higher values indicate better performance (e.g., throughput), the same function can be used by replacing M with its reciprocal $1/M$.

Example 3.1. Figure 5 shows S_{DR} evaluation of a system D, $S(D)_{DR}$, and a system H $S(H)_{DR}$ base on latency measurements over time. For D, we calculate the degradation rate as $(\frac{28}{47} + \frac{31}{75} + \frac{25}{106} + \frac{32}{131} - \frac{6}{163}) * \frac{1}{5} \approx 0.29$, while $S(H)_{DR} \approx 0.021$. This indicates that system D is less stable than system H, and, without any mitigation actions, it will under perform over time.

Discussion. Other reasonable metrics can be incorporated in this scheme. One example is relative standard deviation, which remains impartial to the sequence in which the measurements are presented. We opted for the current metric because it forms an essential building block for making predictions and taking appropriate actions. Over and above the particular metric used to capture stability, we want to emphasize the importance of stability as a new characteristic to measure and optimize for.

4 LST-BENCH BENCHMARKING TOOL

This section presents the implementation details of LST-Bench, a tool designed to benchmark and compare LSTs in the cloud, building on the ideas discussed in §3. Similar to existing benchmarking systems like BenchBase [8, 24] and DIAMetrics [19], LST-Bench includes an application written in Java that executes SQL workloads

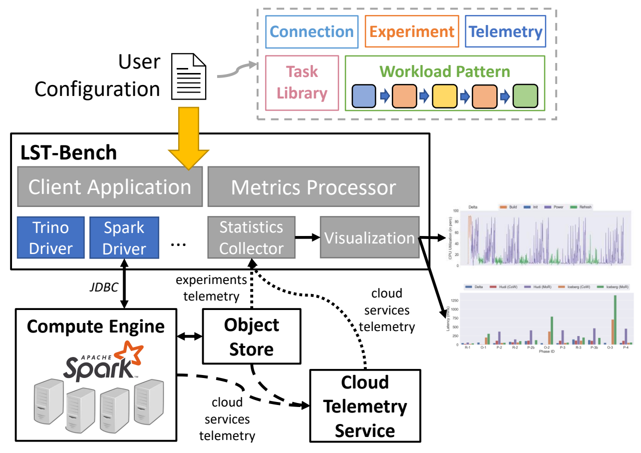


Figure 6: LST-Bench components and execution model.

against a database management system using JDBC (§4.1). Moreover, LST-Bench features a processing module written in Python that consolidates experimental results and calculates metrics to provide insights into LSTs and cloud data warehouses (§4.2).

4.1 Client Application

LST-Bench's client is a flexible and modular benchmarking application that enables users to easily combine various configurations, LSTs, and workloads. As depicted in Figure 6, it follows a *configuration-driven approach* that allows users to define (*i*) clusters connection details, (*ii*) specific options for the experiment, including System-Under-Test (SUT) and LST-Under-Test (LUT), (*iii*) telemetry collection configuration, and (*iv*) the workload to execute.

Experiment Definition. The configuration APIs enable users to define the workload for an experiment as a series of phases, with each phase identified by a unique, user-defined name. LST-Bench provides a library containing the TPC-DS and new tasks described in §3.1. Each task consists of a sequence of SQL statements stored in files in a folder hierarchy; there are different files for the various SQL dialects supported. In addition, certain tasks, such as Optimize, have separate files for each LST under each engine, as LSTs have distinct syntax for executing their table maintenance operations. LST-Bench selects the appropriate files for an experiment based on the workload configuration provided by the user.

Customizability and Extensibility. The SQL files can contain variables that LST-Bench replaces before executing a particular task. Users can use this mechanism to pass configuration values to a task, such as the catalog name, database name, or desired table location in the object store. Additionally, LST-Bench utilizes this feature internally to inject necessary information to run certain tasks, such as the timestamp value for Time Travel.

Incorporating new tasks into the LST-Bench library is straightforward. Users can add the files with the SQL statements that need to be executed and include the new task in the LST-Bench library. Once done, the new task can be referenced from the workload definition files.

Experiment Execution. It is important to note that LST-Bench does not automatically deploy and configure the compute cluster; we are exploring this extension as future work. Instead, it is the user's responsibility to deploy the engine with the corresponding LST libraries. When conducting an experiment, LST-Bench utilizes JDBC and engine-provided drivers to connect to the SUT. LST-Bench creates a *JDBC connection pool* of a size equal to the maximum number of concurrent sessions required by any phase in the experiment. The workload configuration contains the phases that need to be executed sequentially during the experiment. Moreover, the default library includes tasks definitions that create external tables in the engine catalog. These tables point to object store directories containing TPC-DS data generated by the benchmark tool at the specified scale factors. These tasks are typically invoked at the start of an experiment and the external tables they create are currently used in Load and Data Maintenance tasks.

4.2 Metrics Processor

The metrics processor relies on telemetry collected by LST-Bench as well as external, cluster-level telemetry that the user enables up through cloud services. Specifically, LST-Bench collects and stores a detailed breakdown of the start and end times of each experiment, phase, task, and statement, as well as other configuration values during the execution of a given workload. This telemetry allows us to reason about the latency and throughput of the evaluated LSTs. In addition, we rely on time series data that captures resource utilization, storage API calls, or network I/O volume gathered by cloud service providers which is accessible via dedicated APIs[7]. The metrics processor package provides generalized drivers for extracting external telemetry as well as specific implementations of those cloud setups that we use for our evaluation. It also provides notebooks and templates that allow users to plot the same (types of) figures capturing both internal and external telemetry that we will discuss next in our evaluation.

5 EVALUATION

In this section, we present benchmarking results for the workloads described in §3.1 using LST-Bench with Delta Lake, Apache Iceberg, and Apache Hudi, running on Apache Spark [67] and Trino [74]. *We used default parameter settings (e.g., isolation level) and did not perform any special tuning for the evaluated LSTs. Tuning for optimal performance is beyond the scope of this paper, whose focus is on how to benchmark performance over LSTs.* We chose Spark and Trino as the compute engines for the evaluation because they are widely adopted, open-source, and offer the most mature integration with the LSTs examined in this study, based on our own experience. We note that our benchmarking framework can readily be used with other engines, and the extent to which those engines are integrated with different LSTs will materially impact the performance across LSTs. In brief, our results show:

[7]LST-Bench leverages standard auditing telemetry, making its insights-gathering approach broadly applicable. However, in scenarios where access to telemetry from cloud services is restricted, such as when the platform is offered through a vendor, certain insights might be unavailable.

- The accumulation of data files significantly degrades LST's performance, up to 6.8*x* in our study, unless maintenance is performed to mitigate its impact.
- DML operations on Spark can result in a significantly higher number of delta files than Trino, causing up to a 2.4X degradation in performance in our tests. Additionally, the baseline query workloads run at nearly double the speed on Trino for both Delta and Iceberg tables.
- CoW and MoR modes have significant trade-offs regarding their read/write interaction with the storage layer. For example, Hudi and Iceberg MoR on Spark lead to high I/O volume and calls, respectively, resulting in higher read query latency than CoW.
- Table maintenance has a big impact on Delta and Iceberg performance stability, whereas Hudi maintains stable performance without periodic maintenance by doing more upfront work.
- Tuning LSTs involves trade-offs depending on user goals. For example, Iceberg's default file group-by-group compaction reduces disruption on read queries running on the same cluster, but significantly increases compaction time.
- Concurrent read/write sessions have non-trivial impact on query performance. Combining maintenance operations with read queries on the same cluster can improve resource utilization without affecting read latency. Running multiple compute clusters concurrently can reduce execution time by leveraging compute and storage decoupling in cloud engines.

We want to emphasize that the results we report are specific to the versions and configurations that we tested, and their performance can be subject to change and improvement due to further tuning and future developments. Our main objective in sharing these findings is to demonstrate LST-Bench's ability to quantify noteworthy trade-offs across combinations of engines and LSTs.

Our results highlight an important point–each LST offers opportunities to tune performance by making careful choices, e.g., when to do maintenance. In general, these choices depend upon the target workload.

Hardware and Software Setup. Our experiments were conducted on clusters running Spark 3.3.1 and Trino 420. Each cluster comprised 1 head and 16 worker nodes. Furthermore, for the evaluation of concurrency (§5.3), additional clusters consisting of 1 head and 7 worker nodes were used. All clusters were provisioned by Azure VMSS [54] and their nodes were Azure Standard E8as v5 instances with AMD EPYC™ 7763 CPU @ 2.45GHz (8 virtual cores) and 64GB RAM. For Spark, we used Delta Lake v2.2.0, Apache Iceberg v1.1.0, and Apache Hudi v0.12.2. In contrast, in Trino, the LST implementation is integrated within the engine; currently, Trino only supports read and write operations for Delta (CoW) and Iceberg (MoR). The data sets for evaluation were stored in Azure Data Lake Storage Gen2 (ADLS) [52]. We leveraged Azure Monitor [53] to collect telemetry from the compute cluster and data storage, and relied on Logs Analytics [55] to execute queries against the collected data.

Experimental Setup. To generate data at different scale factors (SF100, SF1000), we used the *dbgen* tool in the TPC-DS benchmark [72] and stored the generated data in ADLS. Data streams for Data Maintenance were also generated using the same tool and stored in ADLS. Our Single User task is a permutation of the 99 queries in the benchmark. For our evaluation, we use the workload patterns described in §3.1, running WP1 on Spark and Trino, and the remaining patterns solely on Spark; running them on Trino is left for future work. We discuss the results of our evaluation next.

5.1 Longevity

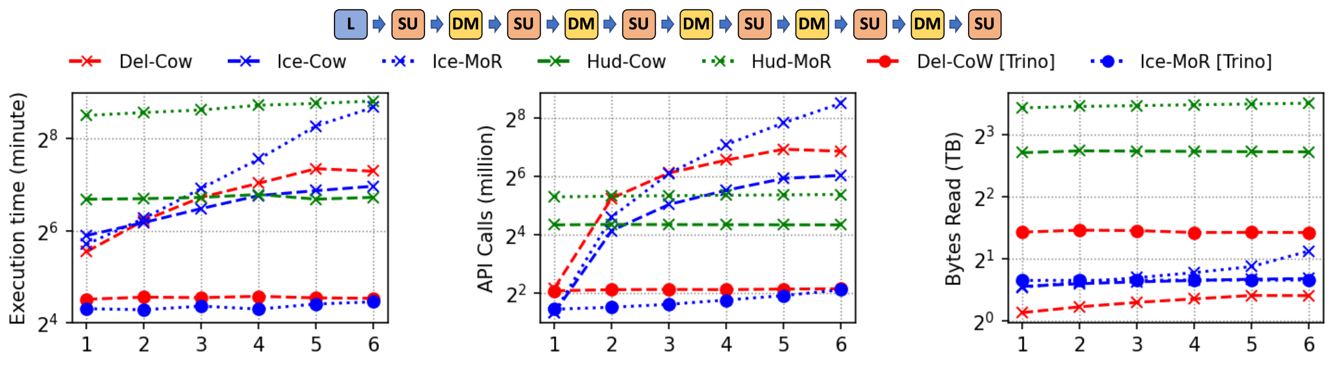
The aim of the *longevity workload* (WP1, §3.1.2), consisting of six Single User and five interleaved Data Maintenance phases, is to evaluate an LST's ability to maximize performance, efficiency, and stability metrics in scenarios that involve frequent data updates. Our findings demonstrate that by executing the workload using LST-Bench, we can detect crucial configurations across all dimensions of the LST model that significantly influence these objectives.

We ran WP1 against seven variations of LSTs derived from representative selections across the three dimensions of the LSTs (§2); Spark and Trino engines, CoW and MoR data algorithms, and Delta, Iceberg, and Hudi as metadata layouts. Two scale factors, 100GB and 1TB, were used against each variation to account for the impact of scaling. We recorded latency of each phase within a run, along with storage and compute tier metrics (§3.2).

(*i*) *Compared to Trino, Spark-related variations exhibit low stability.* Figure 7a illustrates the execution time of all Single User phases in WP1. We observe that in Spark with Delta and Iceberg, the execution time of a Single User phase is always higher compared to its previous run. This behavior is due to *Spark's default distribution-mode* parameter, which results in distribution of writes in a table partition up to 200 times [50] , creating hundreds of thousands of new delta files during Data Maintenance phase runs (Figure 8a). Consequently, there are over 5*x* as many API calls to fetch the delta files for read queries (Figure 7b). Figure 1 shows the ratio of runs of Single User and Data Maintenance phases for two scale factors, confirming the performance degradation observation for both scale factors and phase types. However, this behavior differs from Trino with Delta and Iceberg. Data Maintenance runs on Trino create up to 40*x* fewer files (Figure 8a). We confirmed our hypothesis by running Spark Single User phase on Delta files created by Trino and observed the disappearance of previous degradation.

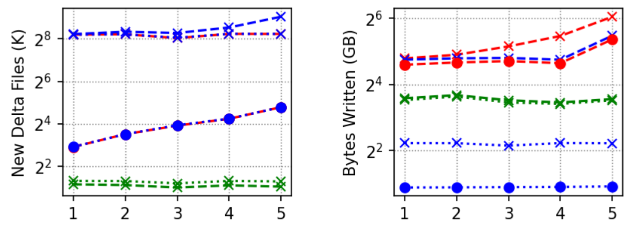
(*ii*) *Trino demonstrates faster query execution compared to Spark.* During the build phase, both Spark and Trino generate an equal number of files. Consequently, in the initial Single User phase before any Data Maintenance phase, both Spark and Trino process the same number of files. However, our results in Figure 7 indicate that Single User 1 completes nearly twice as fast on Trino for both Delta and Iceberg. It is difficult to claim that Trino is consistently faster than Spark, as it depends on various factors like configuration, query type, plan, and data size. For instance, one contributing factor to Trino's speed advantage over Spark in our tests is its absence of checkpointing. While checkpointing enhances fault tolerance, it also introduces significant latency [49].

(*iii*) *Spark with Hudi variation shows unmatched stability.* We now analyze the execution of Single User and Data Maintenance phases using Spark with Hudi (Figures 1, 7 and 8). It exhibits distinct behavior compared to Delta and Iceberg, as the execution time, API calls, and data bytes read and written of the phases remains stable. Through our investigation, we discovered that Hudi has several default optimization-related parameters enabled, such as



(a) Performance of WP1 SU phases. **(b) Total ADLS API calls by SU phases.** **(c) Bytes read from ADLS during SU runs.**

Figure 7: Evaluation of runtimes, network round trips, and storage utilization of SINGLE USER phases for 7 LSTs setups using WP1 ($SF1000$). The results highlight how increase in network round trips affects phase execution times.



(a) Files created by DM phases. **(b) Total size of new files.**

Figure 8: Evaluation of storage usage of WP1 DATA MAINTENANCE phases ($SF1000$).

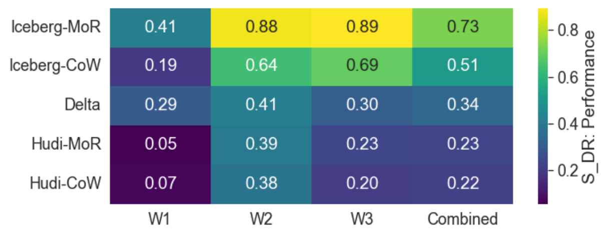


Figure 9: Performance Degradation, S_{DR}, Evaluation. As lower S_{DR} is desired, Hudi emerges as most *stable* LST.

automatic cleanup and compaction [29]. Moreover, a crucial design choice in Hudi is to avoid creating small files by automatically adding sufficient records during the writing process to achieve the desired file size [31]. While these features contribute to stability in performance and efficiency, which is highly desirable, they come with trade-offs. We observe that, unlike Iceberg and Delta, Hudi-related variations read up to ~6x more data and exhibit higher execution latencies. This observation is consistent with findings from other recent studies [44].

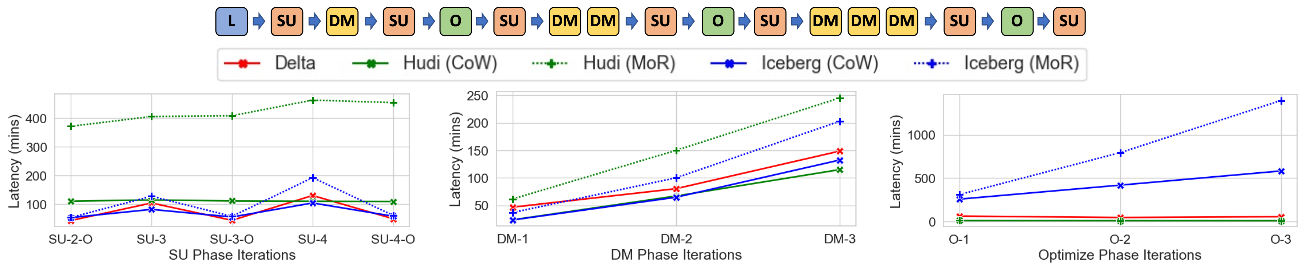
(*iv*) *Read-Write tradeoff in MoR mode.* MoR optimizes frequent table updates, reducing data file rewriting costs and write latencies. However, it introduces a tradeoff: Increased computational cycles and network IO during read operations, impacting performance. To compare, we tested Hudi and Iceberg on Spark with CoW and MoR modes. Figures 7a and 7b demonstrates that Iceberg and Hudi MoR versions consistently have slower performance than CoW variants due to the overheads mentioned earlier.

Performance Stability Analysis. To enable convenient stability comparisons despite the observed variability, we evaluate the effective performance degradation between two phases, S_{DR} (§3.2.2), for each experiment setup. The results are summarized in Figure 9. Each cell presents the combined performance S_{DR} value for SINGLE USER, DATA MAINTENANCE and OPTIMIZE phases across workloads WP1,2,3, as well as the average for each setup. Figure 9 is consistent with our prior discussion, confirming Hudi as the most stable LST, particularly in read-intensive scenarios as indicated by S_{DR} below 0.07. Conversely, except in one instance against WP1, Iceberg consistently exhibits lower stability, with a S_{DR} up to 0.89. Further analysis against WP2,3, which involves optimization and concurrency, is presented in §5.2 and §5.3.

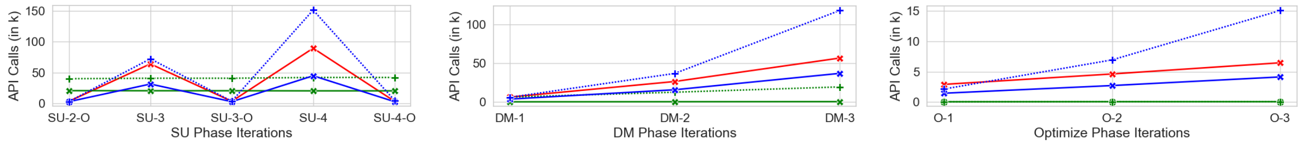
5.2 Resilience

Next, we consider how the performance of LSTs changes when maintenance operations are introduced into the workload. We use the *resilience workload* (WP2, §3.1.2), which evaluates the impact of the OPTIMIZE phase, i.e., compacting small data files into larger files for higher efficiency. The results for SF1000 are shown in Figure 10a. WP2 iteratively executes a sequence of SINGLE USER (labeled SU-*i*), DATA MAINTENANCE (with an increasing number of tasks), OPTIMIZE, and SINGLE USER (labeled SU-*i*-O) tasks. Note for a given *i*, SU-*i* and SU-*i*-O query the same logical version of the data. Results for SU-1 and SU-2 are not shown since they were discussed in §5.1.

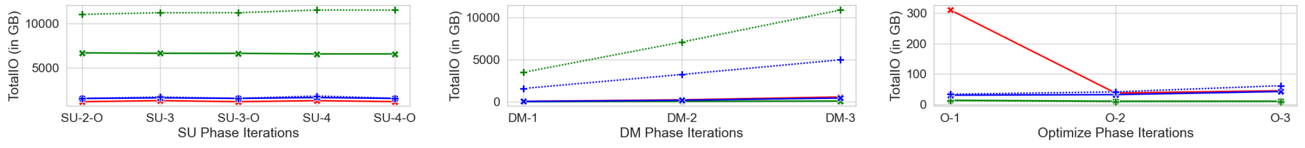
We observe significantly different behavior for the three LSTs. Similar to the performance development in previous workloads, we observe that the addition of OPTIMIZE phases does not impact the performance of Hudi since its performance remains stable and only degrades minimally. For both Iceberg and Delta, on the other hand,



(a) Phase Latency (in minutes). Single User **performance recovers to pre-**Data Maintenance **levels due to** Optimize**.**



(b) Storage API Calls (Millions). Single User **results show significant reduction in API calls after** Optimize**.**



(c) Total I/O Volume (in TB).

Figure 10: Performance and storage efficiency evaluation of WP2 phases (SF1000).

we observe that the Optimize phase has a significant impact on subsequent query execution of Single User phases, as shown by comparing SU-3 to SU-3-O and SU-4 to SU-4-O. We observe that the latency drops by 2.3x (1.5x for Iceberg-CoW and 2.2x for Iceberg-MoR) for the first pair and 2.6x (1.8x for Iceberg-CoW and 3.2x for Iceberg-MoR) for the second pair. This indicates that periodic data maintenance operations are crucial for these LSTs to reduce the number of storage layer access calls (see Figure 10b) which are significantly higher in 'unoptimized' phases. For Hudi, we observe a high I/O volume in the Single User phase, solely composed of file read operations. Furthermore, we see a relatively low amount of read or write activity for CoW in the Data Maintenance and Optimize phases (similar to Iceberg-CoW) while Hudi-MoR has a disproportionally high data volume in the Data Maintenance phase (similar to but more prominent than for Iceberg-MoR).

Interestingly, we observe a drastic increase in latency for Iceberg when executing Optimize phases, which cannot be observed for any other LST. Both I/O and CPU utilization remain comparatively low during these phases but by looking at the storage access calls in detail, we observe that Iceberg issues a large number of (sequentially executed) storage layer access calls. The reason is that for each table, Iceberg's data compaction operation is executed file group-by-group by default [39]. The operation parameters can be adjusted to run in parallel for n groups, which can result in a significant performance boost, but making such tuning adjustments requires additional use case context. Another interesting observation when looking at data maintenance cycles is a spike in I/O cost for the first Optimize phase executed by Delta. This suggests that the operation makes significant changes to the data layout of the table generated after Load, reducing latency and I/O cost for subsequent data maintenance operations.

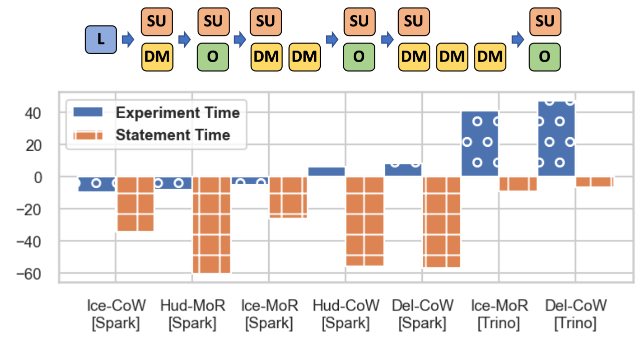
Finally, we observe that for this particular workload, CoW-based outperform MoR-based execution models, i.e., they have a lower latency and incur lower I/O cost.

5.3 Read/Write Concurrency

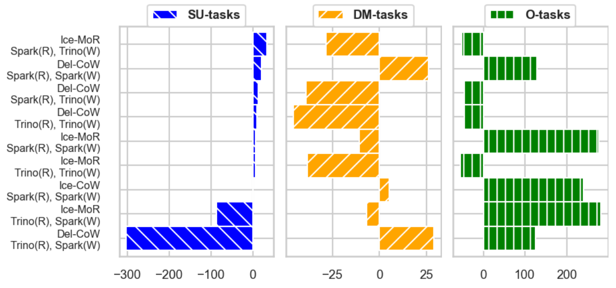
Thus far, the sessions have been executed sequentially, utilizing all available resources. However, LSTs are designed with concurrency in mind. In this section, we analyze the impact of running read and write sessions concurrently using WP3. Note that both WP2 and WP3 contain the same set and sequence of tasks. The only difference is that the phases of WP3 execute Single User task concurrently with either Data Maintenance or Optimize. Thus, we use WP2 as the baseline to evaluate execution speedup and overheads. The concurrent sessions can be executed either on a single cluster or on multiple clusters. In the latter setup, the Single User task is executed on our larger cluster, while the Data Maintenance and Optimize tasks are executed on our smaller one. We refer to this setup as WP3-Multi[8].

Figure 11a shows the comparison of the total execution time of the individual statements within all Single User, Data Maintenance, and Optimize tasks, referred to as *Statement Time*, as well as the total end-to-end experiment execution time, which we call *Experiment Time*, for a single cluster. In Spark, we observe that the *Experiment Time* for WP3 is within a margin of 10% compared to WP2, while *Statement Time* degrades by at least 25% across all

[8]Hudi excluded since we encountered issues where queries failed due to updated underlying data, which prevented us from executing the Data Maintenance and Single User tasks concurrently on separate clusters.



(a) Single cluster (WP3).



(b) Breakdown per task type on multiple clusters (WP3-Multi). (R) represents the reading engine (SU tasks), (W) the writing engine (DM and O tasks).

Figure 11: Performance gains and losses (in %) of WP3 and WP3-Multi relative to baseline WP2 (SF1000). Higher positive values indicate better performance.

LSTs. Consequently, the concurrent execution of sessions in this setup does not necessarily lead to significant performance improvements due to resource contention. In contrast, Trino demonstrates more efficient utilization of the cluster resources, resulting in gains of at least 40% in *Experiment Time*, with only minor degradation observed in *Statement Time*.

We now turn our attention to the breakdown of *Statement Time* per task type for the multiple cluster setup, depicted in Figure 11b. To begin, when Spark executes Data Maintenance and Optimize tasks, Trino execution of Single User experiences a substantial slowdown compared to WP2. Conversely, Spark execution of Single User shows a counterintuitive speed-up relative to WP2. These differences can be attributed to the data layout generated by Data Maintenance tasks. For instance, using a smaller Spark cluster for Data Maintenance results in fewer generated files compared to a larger Spark cluster, though it still generates more files than Trino, as we discussed in §5.1. In all scenarios, reduced file reads in the multi-cluster setup leads to a decrease in task execution time. In relation to this observation, Optimize tasks in Spark are notably faster compared to WP2 due to fewer file reads, while Optimize tasks in Trino tend to be slower, primarily due to constrained resources in the smaller cluster. In summary, these results highlight the significant impact of setup and configuration values for LSTs, table maintenance operations, and engines on the overall performance, and the importance of considering real-world scenarios that previous benchmarks may have overlooked, such as the use of different engine combinations, in which LST-Bench can help.

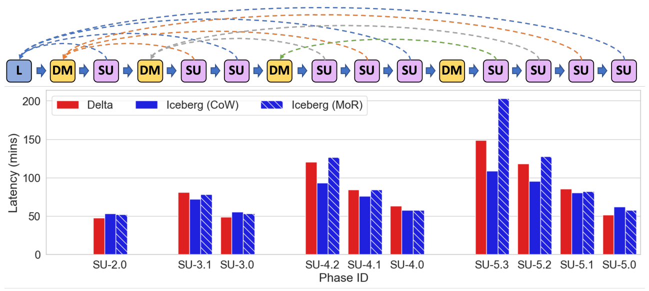


Figure 12: Latency measurements for WP4 (SF1000).

5.4 Time Travel

Lastly, we study the performance of time travel queries in Spark[9], depicted in Figure 12 for SF1000. In the figure, we use the notation SU-$i.v$ to represent the execution of a Single User task i that queries version v of the table. Here, $v = 0$ corresponds to the table after the Load phase, while $v = j$ (where $j > 0$) corresponds to the table after j iteration of Data Maintenance. The results are consistent with our expectations. We observe that the query latency increases as additional data files are written into the tables. This latency increase is similar to the increase observed in WP1 as new Data Maintenance operations are executed. In other words, writing new data after a specific version has been created appears to have minimal impact on querying that specific version. In addition, we analyze the storage efficiency (e.g. I/O and call counts) and find no significant difference between queries run on the latest version of the data and corresponding time travel queries on that version after data modifications have been applied. This indicates that these LSTs can efficiently support time travel queries without incurring any significant overhead in query performance or storage.

6 DISCUSSION

LST-Bench aims to serve various personas with distinct interests and requirements related to LSTs and their engine integrations, both existing and future ones. Notably, even in scenarios where access to infrastructure telemetry data is restricted, various personas can still leverage LST-Bench by using custom patterns tailored to their specific workloads and objectives.

(*i*) *Developers from Engine-Building Organizations.* A primary user persona targeted by LST-Bench is developers from organizations that build and commercialize platforms and query engines relying on LSTs. For instance, Microsoft has recently adopted LST-Bench for evaluating their system deployments, benefiting from valuable insights gained through these evaluations. They leverage LST-Bench to compare various engine and LST version combinations, and have plans to automate this process for tracking progress over time. Additionally, some of these organizations have recently focused on the development of a metadata translation layer from one LST to another [13, 56]. In such cases, developers can use LST-Bench to assess the effectiveness of the conversion process.

[9]Hudi excluded due to SQL extension bug for time travel queries: HUDI-7274.

(*ii*) *Developers from Engine-Using Organizations.* Another key user persona includes developers from organizations that manage their own engines, even if they do not build them. Additionally, prospective customers seeking to compare and choose across platforms for a variety of workloads fall into this category. Some of these users have expressed interest in constructing their own *benchmark packages* tailored to their specific scenarios, not necessarily building upon TPC-DS as the base workload. They can then employ LST-Bench to perform various tasks, such as engine selection and configuration optimization.

(*iii*) *Researchers and Data Professionals.* LST-Bench extends its utility to researchers and data professionals interested in studying best practices and tuning configurations across the three-dimensional space outlined in §2. For example, LST-Bench can help to automate the selection of the optimal configuration for a table between CoW and MoR modes, which result in different trade-offs between read and write performance as demonstrated in §5. By expanding the variety of datasets and packages in LST-Bench, these users can not only fine-tune configuration values for LSTs, table maintenance operations, or engines, but also investigate the existence of "no-regret" defaults that apply to these configuration parameters. Comparisons with the best defaults can be quantified to provide a clear understanding of the differences. While such evaluations are enabled by the LST-Bench framework, they are beyond the scope of this paper.

Community Engagement and Adoption. The decision to open-source LST-Bench invites participation from organizations relying on LSTs for data processing. The engagement with these organizations has led to discussions about extensions to the tool in various areas, with some already integrated into the code.

(*i*) *Workload Representation Model.* The existing model lacks a mechanism to define tasks *dynamically* based on the data stored in LSTs, a feature valuable in various scenarios. For instance, one could create tasks to optimize a partitioned table, with each task executed on a user-configurable number of partitions. In another scenario, one could split the data maintenance task into queries that affect smaller data batches, each composed of a user-configurable number of rows. However, these patterns do not fit easily into the workload representation framework described in §3.1 because the number of tasks to execute is not known in advance. To address this challenge, we introduced the concept of *parameterized custom tasks*, which expand the framework's capabilities by enabling the integration of custom user code for generating workflows dynamically.

(*ii*) *Libraries of Workload Components.* As explained in §4, *tasks* in LST-Bench are organized into libraries for convenient reuse across different workloads. Feedback from users indicated that this approach can lead to many redundant entries within workload pattern definition files. Consequently, we are investigating expanding our library model to incorporate definitions for other workload components, such as a *session* or a *phase*, which could then be shared and reused across multiple workloads as well.

(*iii*) *Workload Packages.* Users recommended expanding the scope to include other standard benchmarks, like TPC-H, that are commonly used to evaluate analytical systems, as well as additional scenarios in which LSTs are commonly used, such as data cleaning [10] and Change Data Capture (CDC) table mirroring with transactional consistency guarantees [9].

(*iv*) *Metrics.* In the context of the CDC scenario, which allows various workflow designs (potentially dependent on the LST capabilities), the choice of appropriate metrics for evaluating the scenario is still unclear. For example, users may prioritize *data availability delay* over *end-to-end execution time.*

7 RELATED WORK

The research and methods of evaluation of open LSTs are rapidly evolving, with new insights being published on a monthly basis. The bulk of literature is in the form of blog posts by vendors or users and is based on established benchmarks that were originally intended for a different category of systems. In this section, we first focus on works that compare LSTs both theoretically and empirically, then we discuss benchmarking methodology that is relevant for LSTs, and lastly, we discuss frameworks proposed in previous work.

Comparing LSTs. Blog posts and papers focusing on the comparison of LSTs can be split into two categories. The first category is a theoretical evaluation of the different approaches, looking at features such as transaction management, schema evolution, and time travel [6, 7, 44]. Blog posts also commonly mention open-source statistics such as number of committers, pull requests, and Github ratings [51, 75]. After examining features and statistics, these blog posts often endorse one LST over others. In contrast, our objective is to establish a standardized approach to evaluate the performance and stability of LSTs, and to offer a framework that allows for empirical comparisons rather than just theoretical discussions.

The second category comprises comparisons of LSTs based on experimental evaluations. For instance, in a recent paper, Jain et al. [44] used TPC-DS benchmarking strategies adapted for LSTs to identify the strengths and weaknesses of each of them: (*i*) They evaluate the impact of data updates on performance by running five sample queries, then merging changes into a table multiple times, and running the same queries again, and (*ii*) they create a synthetic micro-benchmark with varying data refresh sizes to test the impact of the update size on the performance of the LST. Similarly, other recent blog posts [16, 46] have also used the Load task (which has been modified to use Parquet as the source format [47]) and the Single User task from TPC-DS to compare performance of LSTs.

Drawing conclusions about the superiority of a specific LST from the aforementioned works is challenging. For instance, [44] found that (*i*) Delta outperforms both Iceberg and Hudi in TPC-DS query performance, (*ii*) Delta and Iceberg have significantly faster load times than Hudi, and (*iii*) Delta provides better query performance after data has been modified than either Iceberg or Hudi. Other works [16, 46] suggest that Hudi is competitive with Delta in these same dimensions. However, this does not mean that the results are misrepresented by any of those works, as engine setup and configuration parameters can significantly impact these evaluations [48, 78]. Additionally, most evaluations use different versions of the LSTs and underlying execution engines further complicating objective comparisons. To address these issues, we proposed an evaluation methodology and framework that is open source, customizable, repeatable, and easy to use, providing users with a one-stop solution to evaluate these formats objectively.

Benchmarking LSTs. Prior research has used various benchmarks to compare LSTs, but the most commonly used benchmark is TPC-DS. It is important to note that while TPC-DS V1 [57] was designed to evaluate monolithic RDBMSs, TPC-DS V2 [64] was specifically developed to cater to SQL-based big data systems. However, even though TPC-DS V2 already considered SQL engines running on a common storage layer (e.g., HDFS) that could be accessed by multiple systems, it ignored key elements in evaluating LSTs, such as data layout optimization, time travel, or the impact of data manipulations over extended time periods. Furthermore, the TPC-DS result consists of a single performance metric, which, although straightforward for ranking purposes, is insufficient in capturing critical LST-based concepts like *stability* (§3.2). We have therefore proposed metrics that complement the TPC-DS performance score and can help to evaluate a system across these additional dimensions.

As mentioned above, prior work [44] has taken a first step towards modifying TPC-DS by evaluating the performance difference of a set of queries before and after a series of SQL MERGE INTO statements were executed. With our work, we take the idea of long-term impact evaluation one step further and make TPC-DS composable, i.e., we allow users to create their own workloads based on TPC-DS by mixing and matching the different TPC-DS phases. This modification allows us to evaluate all of the previously unaddressed elements that are unique to LSTs.

Prior work has also examined customized micro-benchmarks designed to evaluate the read and write capabilities of different LSTs. For instance, these benchmarks include operations that append and remove data from an existing table, and mimic GDPR deletions [12]. Integrating these workloads into our benchmarking framework to further extend the evaluation should be a straightforward process.

Benchmarking Frameworks. It is important to note that previous research has developed several benchmarking frameworks mainly geared towards evaluating SQL systems. For example, OLTP-Bench [24] can execute several standardized workloads using different database backends such as PostgreSQL or SQL Server. Similarly, DIAMetrics [19] was designed to allow its users to compare and contrast the execution of different (customizable) benchmarks, also extending the idea of benchmarking to include other aspects such as data movement and data security. In this paper, we describe a framework that is specifically focused on LSTs. In addition to their specific SQL dialects, LSTs may be executed on different types of clusters, with different optimization parameters, for which we deploy different (and novel) benchmarks in their evaluation. DSB [25] focuses on workload-driven RDBMSs that adapt over time using ML techniques, extending TPC-DS with more complex data distribution, query templates, and dynamic workloads. In contrast, LST-Bench concentrates on LSTs but could easily integrate DSB's modifications to expand the range of evaluated scenarios.

YCSB [15] presents a framework to compare cloud data *serving* systems like Cassandra or HBase using tiers and workloads. YCSB and LST-Bench share similarities in their approach, but YCSB focuses solely on this category of systems, and does not cover aspects such as the composability of workloads, which is a key contribution of our work. Finally, PEEL [11] is designed for benchmarking distributed systems, with a focus on reproducibility and automated experiment processes. Our implementation also takes inspiration from PEEL, although we recognize the need for further automation in LST-Bench to run evaluations more effectively in cloud deployments, which we plan to explore in future work.

8 CONCLUSION

In this paper, we presented LST-Bench, our benchmarking framework for evaluation of LSTs using workload patterns that mimic real-world customer scenarios, such as those found within Microsoft, while at the same time providing means to fairly evaluate those workloads. We discuss in-depth how we enable users to create their own custom workloads using the extendable plug and execute functionality within LST-Bench and showcase how we use LST-Bench to evaluate and compare existing LSTs. Our extensive evaluation finds that these LSTs vary significantly in terms of their performance, storage efficiency, and stability. This demonstrates that LST-Bench can be used to evaluate LSTs effectively and comprehensively.

ACKNOWLEDGMENTS

We would like to thank Joyce Cahoon and Yiwen Zhu for their valuable discussions on the stability metric, and Jose Medrano for his feedback on the benchmarking framework. Additionally, we appreciate the early users and individuals who have shown interest in LST-Bench, as their feedback is helping us steer the project in the right direction.

REFERENCES

[1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends® in Databases* 5, 3 (2013), 197–280.
[2] Amazon. 2023. Redshift - Cloud Data Warehouse. https://aws.amazon.com/redshift/.
[3] Amazon. 2023. S3 - Cloud Object Storage. https://aws.amazon.com/s3/.
[4] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (2020), 3411–3424.
[5] Michael Armbrust, Ali Ghodsi, Reynold Xin, Vuk Ercegovac, Sourav Chatterji, Eun-Gyu Kim, Paul Lappas, Yannis Papakonstantinou, Yingyi Bu, Yuhong Chen, Yijia Cui, Rahul Govind, Aakash Japi, Kiavash Kianfar, Xi Liang, Jon Mio, Mukul Murthy, Supun Nakandala, Andreas Neumann, Nitin Sharma, Yannis Sismanis, Justin Tang, Joseph Torres, Min Yang, Li Zhang, and Bilal Aslam. 2023. Making Data Engineering Declarative. In *Conference on Innovative Data Systems Research (CIDR)*.
[6] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *Conference on Innovative Data Systems Research (CIDR)*.
[7] Vladimir Belov and Evgeny Nikulchev. 2021. Analysis of Big Data Storage Tools for Data Lakes based on Apache Hadoop Platform. *International Journal of Advanced Computer Science and Applications* 12, 8 (2021).
[8] BenchBase. 2023. Multi-DBMS SQL Benchmarking Framework via JDBC. https://github.com/cmu-db/benchbase.
[9] Ryan Blue. 2023. *CDC Data Gremlins.* https://tabular.io/blog/cdc-data-gremlins/
[10] Ryan Blue. 2023. *Tutorial: Using Trino and Iceberg for data warehousing.* https://tabular.io/tutorials/using-trino-and-iceberg/
[11] Christoph Boden, Alexander Alexandrov, Andreas Kunft, Tilmann Rabl, and Volker Markl. 2017. PEEL: A Framework for Benchmarking Distributed Systems and Algorithms. In *TPC Technology Conference (TPCTC)*.
[12] Brooklyn Data Co. 2023. *Setting the Table: Benchmarking Open Table Formats.* https://brooklyndata.co/blog/benchmarking-open-table-formats
[13] Tim Brown. 2023. *Announcing Onetable.* https://www.onehouse.ai/blog/onetable-hudi-delta-iceberg
[14] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth

Jayachandran, Siddharth Seth, Deepak Jaiswal, Slim Bouguerra, Nishant Bangarwa, Sankar Hariappan, Anishek Agarwal, Jason Dere, Daniel Dai, Thejas Nair, Nita Dembla, Gopal Vijayaraghavan, and Günther Hagleitner. 2019. Apache Hive: From MapReduce to Enterprise-grade Big Data Warehousing. In *ACM International Conference on Management of Data (SIGMOD)*. 1773–1786.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)*. 143–154.

[16] DataBeans. 2022. *Delta vs Iceberg vs Hudi : Reassessing Performance.* https://databeans-blogs.medium.com/delta-vs-iceberg-vs-hudi-reassessing-performance-cb8157005eb0

[17] Databricks. 2023. https://www.databricks.com/.

[18] Databricks. 2023. CREATE TABLE CLONE. https://docs.databricks.com/sql/language-manual/delta-clone.html.

[19] Shaleen Deep, Anja Gruenheid, Kruthi Nagaraj, Hiro Naito, Jeff Naughton, and Stratis Viglas. 2020. DIAMetrics: Benchmarking Query Engines at Scale. *Proc. VLDB Endow.* 13, 12 (2020), 3285–3298.

[20] Delta Lake. 2023. https://delta.io/.

[21] Delta Lake. 2023. Optimistic concurrency control. https://docs.delta.io/2.2.0/concurrency-control.html#optimistic-concurrency-control.

[22] Delta Lake. 2023. Optimizations. https://docs.delta.io/latest/optimizations-oss.html.

[23] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *ACM International Conference on Management of Data (SIGMOD)*. 1243–1254.

[24] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (2013), 277–288.

[25] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. *Proc. VLDB Endow.* 14, 13 (2021), 3376–3388.

[26] Google Cloud. 2023. Cloud Storage. https://cloud.google.com/storage.

[27] Apache Hudi. 2022. RFC - 15: Metadata Table. https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=147427331.

[28] Apache Hudi. 2023. https://hudi.apache.org/.

[29] Apache Hudi. 2023. Automatic async compaction. https://hudi.apache.org/docs/compaction.

[30] Apache Hudi. 2023. Deployment considerations. https://hudi.apache.org/docs/0.12.2/metadata/#deployment-model-c-multi-writer.

[31] Apache Hudi. 2023. File Auto Sizing. https://hudi.apache.org/docs/file_sizing.

[32] Apache Hudi. 2023. Optimization Procedures. https://hudi.apache.org/docs/procedures#optimization-table.

[33] Apache Hudi. 2023. Schema Evolution. https://hudi.apache.org/docs/0.12.2/schema_evolution/.

[34] Apache Hudi. 2023. Supported Concurrency Controls. https://hudi.apache.org/docs/0.12.2/concurrency_control/#supported-concurrency-controls.

[35] Apache Hudi. 2023. Time Travel Query. https://hudi.apache.org/docs/0.12.2/quick-start-guide#time-travel-query.

[36] Apache Iceberg. 2022. Schema evolution. https://iceberg.apache.org/docs/1.1.0/evolution/#schema-evolution.

[37] Apache Iceberg. 2023. https://iceberg.apache.org/.

[38] Apache Iceberg. 2023. Concurrent write operations. https://iceberg.apache.org/docs/1.1.0/reliability/#concurrent-write-operations.

[39] Apache Iceberg. 2023. MAX_CONCURRENT_FILE_GROUP_REWRITES. https://iceberg.apache.org/javadoc/1.1.0/org/apache/iceberg/actions/RewriteDataFiles.html#MAX_CONCURRENT_FILE_GROUP_REWRITES.

[40] Apache Iceberg. 2023. Partition evolution. https://iceberg.apache.org/docs/1.1.0/evolution/#partition-evolution.

[41] Apache Iceberg. 2023. Spark Procedures. https://iceberg.apache.org/docs/latest/spark-procedures/.

[42] Apache Iceberg. 2023. Specification. https://iceberg.apache.org/spec/.

[43] Apache Iceberg. 2023. Time travel. https://iceberg.apache.org/docs/1.1.0/spark-queries/#time-travel.

[44] Paras Jain, Peter Kraft, Conor Power, Tathagata Das, Ion Stoica1, and Matei Zaharia. 2023. Analyzing and Comparing Lakehouse Storage Systems. In *Conference on Innovative Data Systems Research (CIDR)*.

[45] Ralph Kimball and Margy Ross. 2013. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling* (3rd ed.). Wiley Publishing.

[46] Alexey Kudinkin. 2022. *Apache Hudi vs Delta Lake - Transparent TPC-DS Lakehouse Performance Benchmarks).* https://www.onehouse.ai/blog/apache-hudi-vs-delta-lake-transparent-tpc-ds-lakehouse-performance-benchmarks

[47] Delta Lake. 2023. Delta OSS TPC-DS Benchmark. https://github.com/delta-io/delta/tree/master/benchmarks.

[48] Chen Lin, Junqing Zhuang, Jiadong Feng, Hui Li, Xuanhe Zhou, and Guoliang Li. 2022. Adaptive Code Learning for Spark Configuration Tuning. In *IEEE International Conference on Data Engineering (ICDE)*. 1995–2007.

[49] Emma Lullo. 2023. *Starburst: Trino: The origins and development of fault-tolerant execution.* https://www.starburst.io/blog/trino-development-fault-tolerant-execution/

[50] Dipankar Mazumdar. 2023. *Dremio: Write performance in Data Lakes with Apache Iceberg & Spark.* https://www.linkedin.com/posts/dipankar-mazumdar_apacheiceberg-dataengineering-softwareengineering-activity-7085019704540418048-J4hG

[51] Alex Merced. 2022. *Comparison of Data Lake Table Formats (Apache Iceberg, Apache Hudi and Delta Lake).* https://www.dremio.com/blog/comparison-of-data-lake-table-formats-apache-iceberg-apache-hudi-and-delta-lake/

[52] Microsoft. 2023. Azure Data Lake Storage. https://azure.microsoft.com/products/storage/data-lake-storage.

[53] Microsoft. 2023. Azure Monitor. https://azure.microsoft.com/products/monitor.

[54] Microsoft. 2023. Azure Virtual Machine Scale Sets. https://azure.microsoft.com/products/virtual-machine-scale-sets/.

[55] Microsoft. 2023. Log Analytics in Azure Monitor. https://learn.microsoft.com/azure/azure-monitor/logs/log-analytics-overview.

[56] Microsoft. 2023. Universal Format (UniForm). https://learn.microsoft.com/en-us/azure/databricks/delta/uniform.

[57] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB '06)*. 1049–1058.

[58] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *ACM International Conference on Management of Data (SIGMOD)*. 677–689.

[59] OneHouse. 2023. https://www.onehouse.ai/.

[60] Oracle Exadata. 2023. http://www.oracle.com/exadata.

[61] Apache ORC. 2023. https://orc.apache.org/.

[62] Apache Ozone. 2023. https://ozone.apache.org/.

[63] Apache Parquet. 2023. https://parquet.apache.org/.

[64] Meikel Poess, Tilmann Rabl, and Hans-Arno Jacobsen. 2017. Analysis of TPC-DS: the first standard benchmark for SQL-based big data systems. In *ACM Symposium on Cloud Computing (SoCC)*. 573–585.

[65] Matthew Powers. 2023. *Delta Lake Schema Evolution.* https://delta.io/blog/2023-02-08-delta-lake-schema-evolution/

[66] Matthew Powers. 2023. *Delta Lake Time Travel.* https://delta.io/blog/2023-02-01-delta-lake-time-travel/

[67] Apache Spark. 2023. https://spark.apache.org/.

[68] Apache Spark. 2023. SET. https://spark.apache.org/docs/3.4.1/sql-ref-syntax-aux-conf-mgmt-set.html.

[69] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of POSTGRES. In *ACM International Conference on Management of Data (SIGMOD)*. 340–355.

[70] Tabular. 2023. https://tabular.io/.

[71] Teradata. 2023. https://www.teradata.com/.

[72] TPC. 2021. TPC-DS Specification Version 3.2.0. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v3.2.0.pdf.

[73] TPC. 2023. TPC Benchmarks Overview. https://www.tpc.org/information/benchmarks5.asp.

[74] Trino. 2023. https://trino.io/.

[75] Kyle Weller. 2023. *Apache Hudi vs Delta Lake vs Apache Iceberg - Lakehouse Feature Comparison.* https://www.onehouse.ai/blog/apache-hudi-vs-delta-lake-vs-apache-iceberg-lakehouse-feature-comparison

[76] Paul Westerman. 2001. *Data warehousing: using the Wal-Mart model.* Morgan Kaufmann.

[77] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (2017), 781–792.

[78] Yiwen Zhu, Subru Krishnan, Konstantinos Karanasos, Isha Tarte, Conor Power, Abhishek Modi, Manoj Kumar, Deli Zhang, Kartheek Muthyala, Nick Jurgens, Sarvesh Sakalanaga, Sudhir Darbha, Minu Iyer, Ankita Agarwal, and Carlo Curino. 2021. KEA: Tuning an Exabyte-Scale Data Infrastructure. In *ACM International Conference on Management of Data (SIGMOD)*. 2667–2680.