Master's Thesis Computing Science

Radboud University Nijmegen

Evaluating Data Stream Write Performance in Lakehouse Systems

Author:
Vinícius Ribeiro Machado Schmidt
s1123702

University supervisor:
prof. dr. ir. Djoerd Hiemstra
hiemstra@cs.ru.nl

Company supervisors:
Chrystel Philipsen
chrystel.philipsen@sogeti.com
Marcel de Wit
marcel.de.wit@sogeti.com

Second assessor:
dr. Johannes Textor
johannes.textor@ru.nl

February 27, 2026

Acknowledgements

I would like to thank Djoerd Hiemstra and Johannes Textor for agreeing to supervise this project and providing support and feedback when necessary.

Further, I would like to thank Guillermo Sanchez and Pedro Holanda, whose technical knowledge helped me immensely when designing my experiments and testing my code. A special mention to Hannes Mühleisen for introducing me to them and helping me refine my research topic.

Thank you also to Chrystel Philipsen and Marcel de Wit, who provided useful tips and feedback, while I was carrying out this project at Sogeti.

Finally, I would like to thank my parents and my girlfriend, for all their support that carried me through the whole process of completing this thesis. I am also very grateful for all the sacrifices my parents made, so that I could be where I am now.

Abstract

The so-called data lakehouse systems were introduced for facilitating management and analytics of data contained in a data lake. Lakehouses removed the need of having (proprietary) data warehouses on top of those data lakes, which often required expensive ETL pipelines for ingesting data.

Lakehouse systems have become a popular technology, with the ability of integrating with multiple data sources. One common type of data source used in analytics tasks is streaming data, which can be integrated with lakehouses through modern stream processing engines. Although there is a need for integrating lakehouses and streaming data within the industry, there has been no previous research evaluating the throughput of lakehouses when ingesting that data.

Our work focuses on that research question by developing a benchmarking tool that evaluates the throughput of lakehouses when ingesting streaming data. We evaluated ingestion of individual events, as well as batched data. In our experiments, we included Iceberg and DuckLake and we also looked at DuckLake's "data inlining" feature to understand how it impacts performance. The results show that DuckLake is able to outperform Iceberg's throughput in both ingestion modes, although data inlining only provided performance gains when ingesting individual events.

We believe this work provides a first attempt at measuring the throughput of lakehouse systems when ingesting streaming data. Future works can use our benchmark as a baseline and extend it to other lakehouse systems.

Contents

Acronyms

SPE Stream processing engine
SUT System under test

Chapter 1

Introduction

1.1 Motivation and research questions

Many modern data storage architectures rely on so-called *data lakes*, which consist of (cloud) object storage, e.g. Amazon S3[1], Azure Blob Storage[2], for storing large amounts of (un)structured and semi-structured data in a centralized place [51]. Part of the data can then be copied into data warehouses through a process called "extract, transform, load" (ETL), so that it can be used for further downstream analytics tasks [33]. Although this two-tier architecture has become widespread among many enterprises, it has a few drawbacks. ETL pipelines used to load data into warehouses can be time-consuming and error-prone, causing the data in the warehouse to be outdated and sometimes inconsistent [40]. Additionally, maintaining the two systems is expensive and redundant, since part of the data is stored in both the data lake and the warehouse [33]. Finally, the flexibility of storing various file formats in the same place comes with the cost of managing different file versions and metadata, which with time can transform the data lake into a "data swamp" [9].

Following the limitations of data lakes and the two-tier architecture, Michael Armbrust et al. [33] proposed a new data storage architecture called *data lakehouse*. A lakehouse, also referred to as an open table format, builds on top of data lakes by structuring data into tables that are stored in immutable columnar formats, e.g. Apache Parquet[3]. These tables are managed with centralized metadata files that keep track of the current schema and the different versions of the table, all while ensuring that updates to tables occur in ACID [20] transactions [33]. Furthermore, the use of open formats for storage gives users control over their data, in contrast to proprietary formats of data warehouses [9], [33], [40]. These characteristics

[1] `https://aws.amazon.com/s3/`
[2] `https://azure.microsoft.com/en-us/products/storage/blobs/`
[3] `https://parquet.apache.org/`

allow analytical tasks to be performed directly on the lakehouse tables, thus eliminating the need of having a separate data warehouse, thus eliminating the need for a two-tier architecture.

Although open table formats have gained popularity since their introduction, some aspects of lakehouses have led to important questions in recent works. For instance, Paras Jain et al. [35] suggest that performing high frequent insertions into lakehouse tables can be challenging, because every insertion creates a new data file and requires updating the metadata files as well. Further, inserts in Delta Lake, a popular lakehouse format, can have up to hundreds of milliseconds of latency due to its reliance on object stores, which limits the throughput of streaming workloads, as mentioned in [8]. By contrast, the recent introduction of DuckLake [30] brought significant changes on how to handle write transactions to tables, while still ensuring ACID transactions. This is all thanks to the use of a traditional database management system (DBMS), e.g. PostgreSQL[4], for handling table metadata. In addition, DuckLake introduced the idea of "data inlining" [14], which is meant to improve performance of small insertions.

At the same time, many modern stream processing systems, e.g. Apache Spark[5], Arroyo[6] and RisingWave[7], can integrate with lakehouse systems, which emphasizes the demand for up-to-date, real-time data in industry use cases, as it was also pointed out by Michael Armbrust et al. [33]. Despite this demand for integrating stream processing engines (SPEs) with lakehouses, no work has been done yet to evaluate the throughput of open table formats when performing data stream insertions. Therefore, this thesis will fill that gap, by focusing on the following research question:

"*How can we evaluate the throughput of data streams in lakehouse systems?*"

In addition, the following sub-questions will be answered

1. What are the current state-of-the-art benchmarks for lakehouses?
2. What are common patterns used in benchmarks for stream processing systems?
3. How can we ensure our benchmark is actually fair?
4. What is the difference in performance of current lakehouse systems when evaluated with our benchmark?
5. What is the performance gain of the "data inlining" feature of DuckLake in comparison to other lakehouse formats?

[4] `https://www.postgresql.org/`
[5] `https://spark.apache.org/`
[6] `https://www.arroyo.dev/`
[7] `https://risingwave.com/`

1.2 Thesis structure

The rest of this thesis is structured as follows: Chapter 2 goes over the relevant background for this thesis, such as the different lakehouse technologies, how they are implemented and relevant features for this research. Chapter 3 talks about the existing work on benchmarking lakehouses and the different metrics employed by these benchmarks and the systems that they evaluate. Chapter 4 describes the methodology employed to develop our own benchmark and the reasoning behind those decisions. We also describe our experimental setup here. Chapter 5 presents the results of our experiments, which are discussed in Chapter 6. In Chapter 6 we also propose possible directions for future work. Finally, Chapter 7 concludes this thesis by looking back at the main research question and key findings of our experiments.

Chapter 2

Background

Before we can develop our benchmark, we need to first understand how lakehouses work and which characteristics are relevant to the goal of this thesis. We will look at Iceberg and Delta Lake (Sections 2.1 and 2.2), which are the two most popular and more mature lakehouse formats based on GitHub stars count and initial release date respectively [24], [45]. Moreover, we will look at DuckLake, which brings innovative and relevant features for our research (discussed in Section 2.3), despite it still being in version 0.3.

This chapter will also go over the definitions of data stream and throughput, as well as guidelines on performing fair benchmarking.

2.1 Iceberg

Iceberg was developed in 2017 at Netflix and in 2018 it was released as an open-source project within the Apache Software Foundation[1].

2.1.1 Architectural design

Data layer

At the base of Iceberg's architecture, there is a data layer, which consists of data files that constitute the Iceberg table. They can be stored in one of three formats: Parquet, ORC[2] or Avro[3], with Parquet being the most common due to its widespread adoption and performance gains [26].

Metadata layer

Besides the data layer, there is the metadata layer, which keeps track of the data files locations, the version history of the table, as well as its current

[1] `https://www.apache.org/`
[2] `https://orc.apache.org/`
[3] `https://avro.apache.org/`

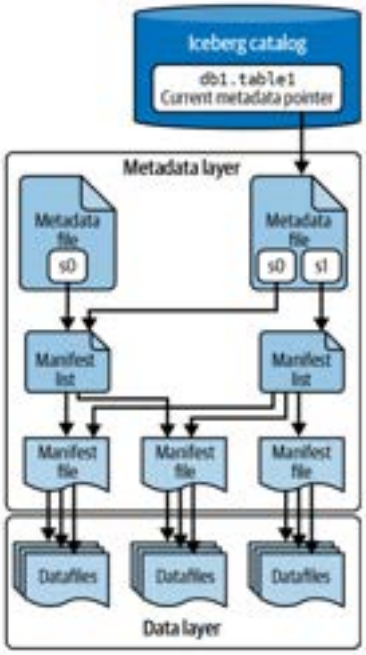

Figure 2.1: Architecture of an Iceberg table, as presented in [26].

schema. The files in the metadata layer are:

- **Manifest files**: Stored in Avro format, these files store pointers to multiple data files, as well as metadata about them, such as the minimum and maximum values of a file's columns and the number of records [26]. The metadata in manifest files is used for filtering data files during read operations, which helps improve read performance [26].
- **Manifest lists**: Manifest lists are another set of Avro files that individually point to a collection of manifest files. Together, the manifest files referenced by a manifest list constitute a specific snapshot, i.e. version, of an Iceberg table [26].
- **Metadata files**: These are JSON files that contain schema information about the table, as well as the snapshot history and a pointer to the current snapshot of that table [26]. A new metadata file is created after every update to the table, marking the creation of a new snapshot. Figure 2.1 depicts the architecture of an Iceberg table, where the newest metadata file has a pointer to both snapshots "s0" and "s1".

Catalog

In order to manage Iceberg tables, a catalog server is used. The goal of the catalog is to point to the latest metadata file of a table, as well as ensure that

this pointer is updated atomically [25], [26]. Consequently, all interactions with an Iceberg table must go through the catalog server. This avoids inconsistencies when clients are concurrently writing to the same table.

Although many technologies can be used as catalog, e.g. AWS Glue[4] and Hive Metastore[5], the developers of Iceberg provide a REST API spec [5] as an attempt to unify catalog access across different engines and languages. One prominent implementation of the Iceberg REST catalog is Apache Polaris[6], which uses a PostgreSQL database to store the latest metadata pointer.

2.1.2 Writing to Iceberg tables

Writing data to an Iceberg table, e.g. via `INSERT`, involves the following steps [26]:

1. First, the engine checks the catalog to get a pointer to the current metadata file, so that it can read the current table schema.

2. New data file(s) are written using the chosen storage format for the table.

3. The engine will then write the manifest file(s) that will contain metadata about the new data file(s), as well as a pointer to them.

4. Next, a new manifest list file is created, which will point to the new manifest file(s). In addition, existing manifest files are added to this new list, which together will correspond to the new snapshot of the table.

5. The engine writes a new metadata file which contains a pointer to the new snapshot (manifest list), as well as a history of all previous snapshots.

6. Finally, following the principle of optimistic concurrency [25], the engine will assume the table has not changed since the start of the operation and will try to atomically update the catalog to point to the new metadata file. However, if a concurrent write has created a new snapshot in the meantime, changing the metadata pointer fails and the engine has to retry. Iceberg tries to structure changes in such a way to minimize the cost of retries [4].

[4] `https://docs.aws.amazon.com/glue/latest/dg/what-is-glue.html`
[5] `https://hive.apache.org/`
[6] `https://polaris.apache.org/`

2.2 Delta Lake

Delta Lake was introduced in 2020 by Armbrust et al. [8]. It was originally developed at Databricks[7] and it was open-sourced in 2022 [43].

2.2.1 Architectural design

Figure 2.2: The architecture of a Delta Lake table, as presented in [8].

Data files

At its core, a Delta table consists of a directory containing data files, which are stored in the Parquet format. As shown if Figure 2.2, this data can be partitioned to improve efficiency of read queries, but this is beyond the scope of this thesis.

Delta log

The metadata of a Delta table is stored in a subdirectory, called "delta log". This directory contains JSON files, each named in increasing order according to the table version they correspond to, e.g. `00001.json`, `00002.json`, `00003.json`, etc. The log files themselves consist of actions, e.g. `add` and `remove`, that were applied to the data files in order to obtain the respective version [8].

Furthermore, Delta also keeps track of so-called "checkpoint" files. These are Parquet files consisting of all the actions in the JSON files up until that point in time. These checkpoint files are created periodically (the default being every 10 transactions), so that Delta can stay performant when reconstructing the current state of a table from the log files [8]. The name of the most recent checkpoint file is stored in a `_last_checkpoint` file, as depicted in Figure 2.2.

[7] `https://www.databricks.com/`

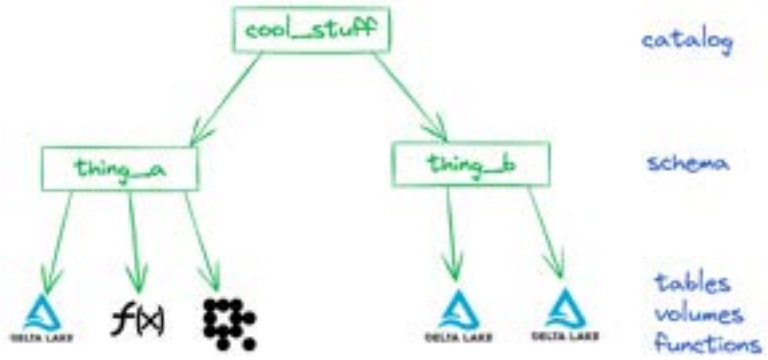

Figure 2.3: Overview of a Unity Catalog, which can be used to manage multiple Delta Lake tables [49].

Catalog

Delta Lake can be connected with a catalog, allowing the management of multiple tables and creating interoperability between other open table formats, such as Iceberg. The most popular catalog used to manage Delta tables is Unity Catalog[8]. There are two versions of this catalog: A proprietary version maintained by Databricks and an open-source version. We focus on the latter, since it is freely available. Similar to the Polaris catalog, the Unity Catalog server can be accessed through a REST API and it uses a relational database to store information about the Delta tables that it knows about.

Although a catalog server is not part of the Delta Lake specification, as it is in Iceberg's case, we include it in our research to make sure we perform a fair comparison between the systems we are analyzing (see also Section 2.5).

2.2.2 Writing to Delta tables

The process of writing data to a Delta table can be broken down into the following steps [8]:

1. Delta will first look for the `_last_checkpoint` file. Using the checkpoint file's ID, all subsequent log and checkpoint files are listed.
2. Using the listed files, the engine will read them to reconstruct the current state of the table, as well as statistics such as min and max values of columns and partitioning information.
3. Once all those files have been read, new data objects are written (possibly in parallel) to the underlying file storage.

[8] `https://www.unitycatalog.io/`

4. Then, assuming the latest log file has ID n, Delta attempts to create a new log file named $n + 1$`.json`. Since there can only be one file corresponding to a specific version of the table, creating the new log file needs to happen atomically. If this step fails due to a concurrent write operation, the transaction can be retried.

5. (Optional) Depending on the configured interval for writing checkpoints, a new checkpoint file will be created and the `_last_checkpoint` file is updated to reflect that change.

It is important to note that, like in Iceberg (Section 2.1.2), optimistic concurrency control is employed for write operations. However, Delta relies on the underlying object store's mechanism to ensure transaction atomicity [8]. For instance, Azure Blob Store, Google Cloud Storage and Amazon S3 support conditional write operations [23], [39], [41], thus ensuring that only one client can create a log file with a specific name.

2.3 DuckLake

DuckLake is an open table format introduced in 2025 as part of the DuckDB Foundation. Although it is currently an experimental release (as of writing, the current version is 0.3 [19]), we believe it will be a valuable addition to our research, as its specification proposes a new way of storing metadata and an interesting approach for handling changes to its tables.

2.3.1 Architectural design

Data layer

DuckLake's data layer consists of Parquet data files on the chosen storage system, which make up the table.

Catalog database

Similar to Iceberg, DuckLake also includes a catalog in its specification. However, DuckLake's catalog differs from Iceberg's in that it stores all of the table's metadata instead of storing the pointer to the current metadata file. In total, the catalog database contains 22 tables that together keep track of the schema, snapshots, data files, tables, statistics and partitioning information of DuckLake tables [16]. Currently, DuckLake supports DuckDB[9], SQLite[10], PostgreSQL and MySQL[11] as its catalog database [12]. Figure 2.4 shows the most relevant tables stored in the catalog [16]:

[9] `https://duckdb.org/`
[10] `https://sqlite.org/index.html`
[11] `https://www.mysql.com/`

Figure 2.4: Architecture of a DuckLake table, as presented in [30].

- `table`: This is where the DuckLake tables' name, path and schema are stored. In addition, it tracks the snapshot interval in which the table is valid.
- `table_stats`: This contains metadata about the DuckLake tables, such as the number of records and the total file size of the data files that compose this table.
- `column`: This table stores the columns that are part of the DuckLake table. The main attributes are the column's name, its type and default value.
- `table_column_stats`: Similar to `table_stats`, this table contains metadata about each column in a DuckLake table, e.g. min and max values and whether the column has `NULL` values.
- `partition_column`: Contains partitioning information about columns on which partitioning has been enabled. It also stores the transformation function that is applied to the column to obtain the partition key, if such a function is required. For instance, if a table contains a timestamp column, one can apply a transformation function to that column and use the resulting value as the partitioning key.
- `data_file`: This table stores information about the data files, such as the path, format (for now, only Parquet is allowed [16]), number of records in the file and the size in bytes.
- `file_column_stats`: Similar to `table_column_stats`, this table stores column statistics on a file level, e.g. amount of records, min and max values and number of `NULL` values.
- `delete_file`: Similar to `data_file`, this table stores the path and format of delete files[12], but it contains the number of records deleted by the file.
- `snapshot`: This table is used for representing the snapshots, i.e. versions, of the DuckLake instance. Every change to a table will result in a new snapshot with a corresponding timestamp.
- `snapshot_changes`: For each snapshot, this table stores a list of changes that were made to the DuckLake instance, e.g. `CREATE`, `INSERT`, `UPDATE`, `ALTER`.

[12]Delete files are used to mark which records have been deleted from a table without actually deleting any files. This is done because Parquet files are immutable and would have to be rewritten every time a record is updated or deleted. By using delete files, deleted records can be reconciled at read time in a process called merge-on-read (MoW) [35], which is beyond the scope of this thesis.

- `schema`: This table stores the different schemas that can be used to separate DuckLake tables into namespaces. The table stores the schema name, its path in the file system, as well as the snapshot interval in which that schema is valid.

2.3.2 Writing to DuckLake

Writing to DuckLake tables happens in the following steps [30]:

1. First, the Parquet files are written to storage, unless data inlining (2.3.3) is enabled and the number of inserted rows is lower than the set threshold.

2. Then, in a single database transaction, the following tables are updated to reflect the new changes:

   - `data_file`
   - `table_stats`
   - `table_column_stats`
   - `file_column_statistics`
   - `snapshot`
   - `snapshot_changes`

   DuckLake enforces a primary key constraint on the snapshot id in the `snapshot` table [13]. If two clients try to concurrently create a new snapshot, one of them will fail due to a `PRIMARY KEY` constraint violation [13]. In that case, DuckLake will try to solve the conflict without having to rewrite the data files. For that, it looks at the `snapshot_changes` table to see what changes were made by the transaction that succeeded. If there are no logical conflicts, e.g. both clients just added new data, the failed client can retry the transaction without having to rewrite the data files [13].

2.3.3 Data inlining

One feature that sets DuckLake apart from Delta and Iceberg is the so-called "data inlining". This feature allows DuckLake to be configured such that writes that insert fewer rows than a set threshold are stored in the catalog database, rather than written out as Parquet files [14]. This can increase the performance of DuckLake for frequent, small, inserts such as a streaming events. The "inlined" data is immediately visible for read operations and it can be flushed out to Parquet files once enough data has been accumulated in the catalog [14].

2.4 Data streams

Streaming data can be defined as data that is produced by one or more sources continuously and thus, is considered unbounded [17], [48]. As opposed to finite datasets that can be processed in their entirety by methods such as MapReduce [27], streaming datasets usually do not have a "final" data point, so streaming processing engines (SPEs) rely on approaches such as (micro-)batching and windowing to create finite groupings from the unbounded data for processing [48]. In addition, streaming events can also be processed individually as they arrive to the processing engine, such as in Spark's "real-time processing" mode [28]. Some use cases where stream processing is commonly applied include IoT sensor-monitoring, fraud detection, advertisement analytics and processing of logs for web services [53].

Modern streaming platforms such as RisingWave and Arroyo provide firsclass support for writing data to Iceberg and Delta Lake, through native connectors. In addition to that, we believe that the inlining feature of DuckLake would be very well-suited for writing streaming data. These characteristics of both modern SPEs and lakehouses indicate that there are industry use cases that require integration between them, which further highlights the relevance of our main research question.

2.5 Benchmarking

When it comes to evaluating systems' performance, it is crucial that the comparison is done fairly to avoid misleading results. In that regard, Raasveldt et al. [38] highlight some of common pitfalls when benchmarking database systems. These pitfalls were obtained from a literature review on best practices and recommendations for both benchmarking in general and benchmarking of database management systems (DBMSs). The authors then identified seven[13] different pitfalls that can occur during DBMS benchmarking, namely:

1. **Reporting non-reproducible results**: In scientific research in general, it is required that results are reproducible, so that they can be independently verified. Benchmarking is no exception to this rule.

2. **Sub-optimally optimizing systems**: In cases where the authors of a paper are comparing their own system with the state of the art, they might feel tempted to not properly optimize the systems they are comparing with, so that they get favorable results [38]. In some other

[13]In the original paper, the authors report "cold" vs "hot" runs and "cold" vs "warm" runs separately [38] , so they reported eight pitfalls in total. We believe however that they pertain to the same aspect of benchmarking, so we include them all in the same pitfall.

cases, failure to properly optimize a system might happen due to lack of familiarity with that system.

3. **Overly-optimizing a system**: Similarly to the previous pitfall, one might optimize their system to perform really well on a specific benchmark. This can be done by optimizing a system for the specific dataset or workloads present in a benchmark, which are all known prior to execution.

4. **Benchmarking code that contains bugs**: Sometimes, performance measurements can be influenced by bugs in the SUTs or in the code performing the benchmark. Bugs can lead to an unfair advantage by skipping a step of the benchmark workload or by only working with a specific dataset [38].

5. **Comparing systems that are fundamentally different**: Benchmark results can only be considered fair if they were obtained from systems that provide the same functionality. Otherwise, one of the systems might be given an unfair advantage by avoiding a set of constraints present in other systems. Raasveldt et al. [38] exemplify this by comparing a complete DBMS with an algorithm specialized in one task. The algorithm has an advantage because it takes fewer aspects into account than the DBMS [38].

6. **Ignoring preprocessing time**: When running benchmarks, the SUTs often have to be initialized with the dataset used in benchmark's workload. This initialization can sometimes include preprocessing steps that a system executes to speed up performance of subsequent tasks. The example given in [38] mentions the creation of database indexes, which if ignored, might favor a system that has efficient, but slow-to-create, indexes.

7. **Comparing "cold", "warm" and "hot" runs**: The concept of a "cold" run refers to executing a benchmark for the first time on a system for which no data has been cached on the host operating system yet. By contrast, subsequent runs of the benchmark are considered "hot", due to the creation of caches and other optimizations that the underlying operating systems might execute. Raasveldt et al. [38] also alert for "warm" runs, which can occur when the SUT is restarted to perform a "cold" run, but caches are still present in the OS memory [38].

In addition, the authors in [38] suggest practices on how to prevent these pitfalls from happening. In some cases, such as that of non-reproducible experiments, it can be easily avoided by including a detailed specification of the configuration parameters used, hardware specifications and source code.

However, other pitfalls are not so trivial to avoid. For instance, making sure the SUTs are not running sub-optimally can be hard depending on the amount of configuration available for each system. Nonetheless, we will follow the checklist provided in [38, Appendix A] and mention it whenever relevant throughout this thesis.

Chapter 3

Related Work

3.1 Benchmarking lakehouses

Lakehouses are a novel technology when compared to the more established DBMSs like PostgreSQL or MySQL. Therefore, not many studies have been published evaluating the performance of open table formats. For that reason, we focus on two notable works in this section: LHBench [35] and LST-Bench [10].

The authors in [35] leverage the TPC-DS benchmark [47] to evaluate three lakehouse systems, namely Apache Iceberg, Delta Lake and Apache Hudi[1]. Although the authors focused on query performance of lakehouses when reading data, they also presented some results for insert operations. They looked at the creation time of TPC-DS tables with 3 TB of bulk data. Their results show that Iceberg and Delta have similar loading times around 2,700 and 2,300 seconds respectively, whereas Hudi takes around 20,000 seconds to load all the data. They attribute this difference to Hudi performing more preprocessing at insertion than Delta and Iceberg. The authors performed a second experiment where they loaded 100 GB of TPC-DS data into each lakehouse and the results were proportional to the first experiment: Delta took around 420 seconds, Iceberg took around 320 seconds and Hudi took upwards of 2,400 seconds to load all the data. It is important to stress that these experiments were run on Spark clusters of 16 workers, each with 8 virtual CPUs and 61 GB of memory [35].

Camacho-Rodríguez et al. [10] presented LST-Bench, which also uses the TPC-DS benchmark, but extends it to include workloads that measure specific characteristics of lakehouses. For instance, the authors looked at the impact that merging many data files into fewer, larger files had on read performance. They also looked at how *not* performing those merge operations can degrade metrics such as CPU usage, memory consumption and disk I/O.

[1]Hudi is another open table format that we excluded from this thesis due to the lack of a dedicated catalog for it.

Their findings show that merging data files significantly helps maintain performance of Delta Lake and Iceberg, whereas Hudi, again, did not benefit so much from those merge operations. This result agrees with [35], as discussed previously. Furthermore, the results of LST-Bench showed that not merging data files can degrade the performance up to 6.8× [10]. Finally, they also utilized clusters of 17 nodes, where each node consisted of 8 virtual CPUs and 64 GB of memory [10].

Both papers, though quite comprehensive in their analyses, do not investigate the performance of writing streams of data into lakehouses. To the best of our knowledge, no other work has been done on that regard either. Therefore, we will address that gap in this thesis.

3.2 Common streaming benchmark patterns

Although our work does not focus on SPEs, we go over relevant characteristics of SPE benchmark systems, so that we can draw ideas for designing our own benchmark.

Yue et al. [53] performed an extensive survey of benchmarks of SPEs, focusing on four characteristics: (i) Dataset type, (ii) data ingestion method, (iii) SUT, (iv) metrics collected. We focus here on dataset type, the data ingestion method and metrics collected. The SUTs mentioned in [53] are not relevant, since our thesis does not look at the performance of SPEs.

3.2.1 Dataset type

As Yue et al. [53, Table 2] show in their paper, datasets are classified into either synthetic or real-world data. Their results indicate that there is a relatively even usage of both types of datasets. In cases where a synthetic dataset is used, e.g. the Linear Road benchmark [7], it is due to the benchmark workload requiring a specific schema for the data. Another example, though not mentioned in [53], is the PGVal benchmark [42], which required a deterministic dataset of web server logs, so the authors had to generate their own data.

In terms of real-word data, Yue et al. [53] found a variety of datasets in the compiled benchmarks. These datasets include IoT sensor data, Twitter data, network traffic data, e-commerce data, online game data, etc.

As discussed in Section 2.4, there are few requirements that characterize data streams. Therefore, many datasets are suitable for simulating streams of data.

3.2.2 Data ingestion

The vast majority of benchmarks gathered in [53] make use of Apache Kafka[2] to ingest data. Kafka is a "distributed event streaming platform" [6] that acts as an event broker between data producers, e.g. an IoT sensor, and consumers, such as Apache Spark, Apache Flink[3] or client APIs for various programming languages. An event is always written to a Kafka topic, so that different types of events can be easily organized and consumers can consume events from the topics they are interested in.

The few papers compiled in [53] that do not use Kafka for data ingestion mostly use a self-designed ingestion method. For instance, Linear Road [7] uses the MIT Traffic Simulator [52], which generates data into flat files that are read by a data driver to simulate incoming traffic data.

In other cases, such as in [29], the authors argue that using an event broker such as Kafka can lead to bottlenecks in the benchmark pipeline. In short, they suggest that the process of storing and partitioning incoming data by the broker can slow down the rate at which SPEs can consume that data [29]. They instead use an in-memory generator, which they tuned to make sure it can produce data faster than the fastest consumer in their benchmark [29]. However, they do not provide any source code for their generator, nor do they supply any data comparing Kafka with their own generator, so it is unclear whether Kafka would have been a bottleneck in their system.

3.2.3 Metrics

In terms of performance metrics compiled by Yue et al. [53], most benchmarks focus on throughput (Section 3.2.4), processing latency, CPU/memory/network usage and disk I/O. Some benchmarks focus on other aspects of SPEs, such as processing guarantees [42] and scalability capacities [21].

In this thesis, we focus primarily on throughput of data streams, which is the main objective of our research question. In addition, we collect CPU usage and disk I/O metrics for a more fine-grained overview of the systems' resource utilization. We do not look at latency, since no processing will be done on the data streams before writing them to the lakehouses. Further, we will not look at memory usage because we will not be writing large amounts of data at once. Rather, we will look at small, constant writes, so we expect memory consumption to be relatively low. Finally, we will not collect network usage metrics, since our experiments will be run locally. We believe this might help to more accurately measure throughput because having a network connection to the storage layer would add noise to our results when writing events in the lakehouses.

[2] `https://kafka.apache.org/`
[3] `https://flink.apache.org/`

3.2.4 Throughput

When benchmarking streaming processing engines (SPEs), a common metric used is the system's throughput, which is defined as the amount of events that the system under test (SUT) can process in a given unit of time. In terms of execution time, Henning et al. [22] presented a thorough approach for measuring throughput, where they run their experiments for 15 minutes and repeat it three times. In addition, Henning et al. [21] run their experiments for five minutes, but the first minute is considered "warm up", which relates to the idea of "cold" vs "warm" runs discussed in [38]. Both of these approaches will be adapted for our experiments.

Chapter 4

Methodology

In this chapter we discuss the implementation of our benchmarking tool. We go over the data ingestion process, data formats used and the metrics that we collect. We also give an overview of the experimental setup, making sure to include as many details to facilitate reproducibility of our results.

4.1 Benchmark design

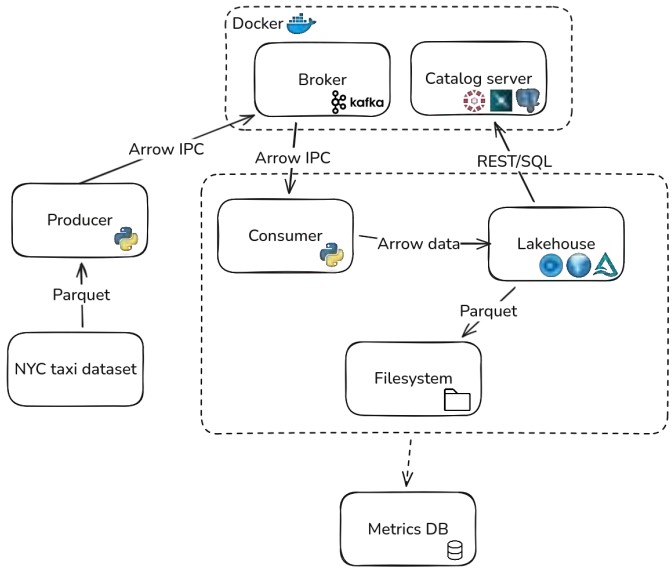

Figure 4.1: Overview of the components in our benchmarking tool.

4.1.1 Dataset

Our benchmark uses the TLC Trip Record Data [44], which is a publicly available dataset of taxi trips in New York City. We use specifically the

trip data of yellow taxi cabs during November 2025, totaling 4,181,444 trip records. This dataset provides a good representation of real-world stream data, since taxi trips are constantly occurring. Further, the dataset can be easily extended to include data from previous years, in case there is the need to scale up the benchmark.

As a preprocessing step, we added a single column to the dataset with an integer id for each trip. This helps us track statistics for each trip processed by our system and it also increases reproducibility of our results.

4.1.2 Data ingestion

Following the approach of previous works, we ingest the dataset into a Kafka broker instance, using the Kafka Python client API[1]. For reproducibility purposes, the broker runs in a Docker container. A single topic is created for the events being produced and we delete that topic after every experiment, to ensure we do not mix events from different runs. The Kafka producer runs in a separate Python process, so that our benchmark can consume the Kafka events independently.

4.1.3 Data format

We use the Arrow format for data representation in our benchmark. Apache Arrow[2] is a standardized, in-memory columnar format, which facilitates data exchange between systems implementing it. The columnar layout of Arrow data also makes it very efficient for writing it in Parquet format [1]. Furthermore, the Arrow specification includes an inter-process communication (IPC) mechanism [2], which can be used to stream arrow data while avoiding the use of serialization or extra copies to/from the in-memory Arrow layout. Therefore, we use the IPC mechanism for producing data to the Kafka broker, so that the consumer can read it as Arrow data directly.

4.1.4 Lakehouses

Our benchmark can interact with the different lakehouse systems in two ways. The default way uses a dedicated library[3] for each system, which is optimized for that specific lakehouse format and is compatible with Arrow data. Alternatively, the benchmark can be run with DuckDB as the client for all three lakehouse formats, since it has first-class support for all three formats [11]. In both cases, the client used to interact with the lakehouse is considered part of SUT and is reported explicitly for clarity of results.

[1] `https://docs.confluent.io/platform/current/clients/confluent-kafka-python/html/index.html`

[2] `https://arrow.apache.org/`

[3] PyIceberg, delta-rs and DuckDB

Additionally, all catalog servers are run on Docker containers, so that the benchmark can be run on any machine, thus contributing to reproducibility of experiments. The only exception is when using DuckDB as a catalog for DuckLake. In that case, the catalog is stored in a local file. This combination is only used when running experiments that involve the "data inlining" feature (Section 2.3.3).

4.1.5 Metrics

All collected metrics are stored in a local database for further analysis. The throughput is calculated as the number of events written to the given lakehouse system, divided by the total number of seconds that the experiment was executed. In addition, the following metrics are collected:

- **Time taken to read each event from the Kafka broker:** We expect read times to be low and consistent, since our benchmark reads events one at a time. Nonetheless, we collect this data to ensure that reading the data does not cause any bottlenecks in our experiments. This time is measured using the `perf_counter_ns` function from the `time` module, which provides the "highest resolution for measuring short durations" [46].
- **Time taken to write each event to the lakehouse table:** On top of looking at the throughput, we analyze the individual writing time of each event to see if systems can keep consistent writing times. Again, we use `perf_counter_ns` to measure write times.
- **CPU usage:** We run a separate Python thread that collects the total CPU usage percentage every second. For that, we use the `psutil` [18] Python library.
- **Disk I/O:** Similar to CPU usage, we collect the amount of bytes written to the machine's main disk at a one-second interval. `psutil` returns the total amount of bytes written since system startup, which strictly increases over time. Instead, we focus on the difference in bytes written between every second, so we can relate the average number of bytes written per second with the throughput of each system.

4.2 Experimental setup

Combining the approaches in [21], [22], we perform three runs of five minutes for each lakehouse, where we consider the first run to be cold and the others to be hot. To ensure cold runs, we restart the machine running the experiments after the three runs for each lakehouse.

Regarding the lakehouse systems evaluated, we only evaluate Iceberg and DuckLake. During the setup of our experiments, we encountered a bug in the `delta-rs` library, which prevents access to tables through Unity Catalog. We opened a GitHub issue[4] for this, but as of writing this thesis, the problem has not been solved yet. Although it is still possible to access and write to Delta tables without Unity Catalog, we believe this would give an unfair advantage to Delta, since the other two systems are connected to catalogs.

4.2.1 Insertion modes

We insert data in two different ways, namely individually and batched. In the first mode, as soon as an event is read by the Kafka consumer, it is inserted into the SUT and the insertion time is measured. This mode of insertion will give us insight in the raw throughput of each system, in a scenario where events cannot wait for window/batch operations.

When inserting batches, we accumulate 1,000 events in an Arrow table before inserting them at once into the given lakehouse table. Here, we evaluate a scenario where a streaming engine would use a window function for grouping incoming data and processing them in some way before writing them to the lakehouse.

4.2.2 Configuration

For DuckLake and Iceberg we use the default configuration of each system, except for the Parquet compression codec and the number of threads used by each system. We set the former to use the snappy [34] algorithm, which provides fast compression speeds. We explicitly set the number of threads to 1, since our benchmark inserts a single event in the SUT each time. Even in the case of Iceberg, where multiple metadata files are written at every insert, these files must be written sequentially, because each metadata file holds a reference to another data- or metadata file below it (Figure 2.1). We confirmed there is no benefit to using multiple threads by running some initial tests, where we noticed no increase in the number of events inserted to the lakehouse table when going from 1 to 16 threads. In fact, in the case of DuckLake, there was a significant decrease in the number of events inserted, as shown in Figure 4.2.

In addition, we perform runs using the "native" client for each lakehouse, DuckDB for DuckLake and PyIceberg for Iceberg, as well as using DuckDB for both. This will give us insight into how much of the performance is impacted by the client used for interacting with the lakehouse tables. PyIceberg does not have the option of interacting with other open table formats, so we cannot use it to write to DuckLake.

[4] `https://github.com/delta-io/delta-rs/issues/3966`

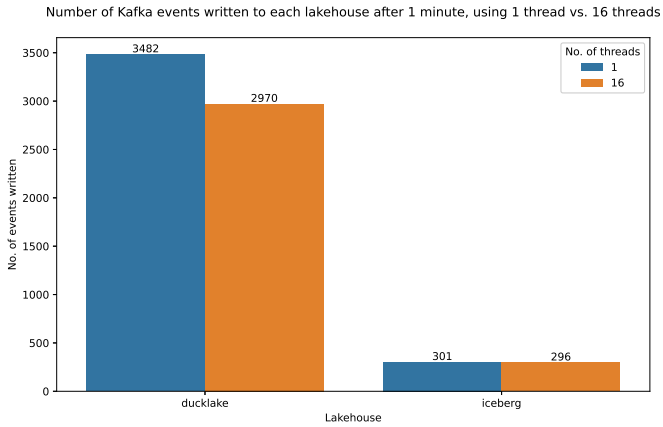

Figure 4.2: Comparison of using 1 thread vs. 16 threads for writing events to DuckLake and Iceberg in a time span of 1 minute.

Finally, when running experiments with the inlining feature of DuckLake enabled, we set the write threshold to be one more than the number of elements being inserted, i.e. 2 for writing individual events and 1001 for writing batches. The values do not matter, as long as they are bigger than the amount we are inserting into DuckLake.

4.2.3 Hardware and software specs

All experiments were executed on a local machine for which the hardware specification can be found in Table 4.1. We believe that running experiments locally significantly reduces complexity and noise that would otherwise be present in a cloud environment.

OS	Linux Mint 22.3 - Cinnamon 64-bit
CPU	AMD Ryzen 7 5700U 1.8 GHz base clock 8 cores and 16 threads
Memory	2× 8 GB DDR4 @ 1600 MHz
SSD	Intel 660p 1 TB NVMe M.2 Up to 1,800 MB/s for sequential writes Up to 220,000 IOPS for random writes

Table 4.1: Overview of the hardware specs used for running our benchmark

All software, as well as their versions can be found in Table 4.2. We hope to make our experiments as reproducible as possible, following the guidelines presented in [38]. We also made our source code available on GitHub[5].

[5] `https://github.com/schmidtvinicius/thesis/tree/main/benchmark`

Software	Version
Python	3.12
Kafka (Docker)	4.1.0
PostgreSQL (Docker)	18.0
Polaris (Docker)	1.3.0-incubating
DuckDB (Python)	1.4.4
PyIceberg (Python)	0.10.0
PyArrow (Python)	22.0.0
confluent-kafka (Python)	2.12.2

Table 4.2: Overview of the software and libraries used by our benchmark, as well as their specific versions.

Chapter 5

Results

5.1 Writing events one by one

5.1.1 Total events written

Figure 5.1 shows how many events were written to each lakehouse table. We can see that the first run of each SUT performed slightly better, before stabilizing at lower levels in the subsequent hot runs. Therefore, we focus on results from hot runs throughout the rest of this thesis. Other results can be found in Appendix A.

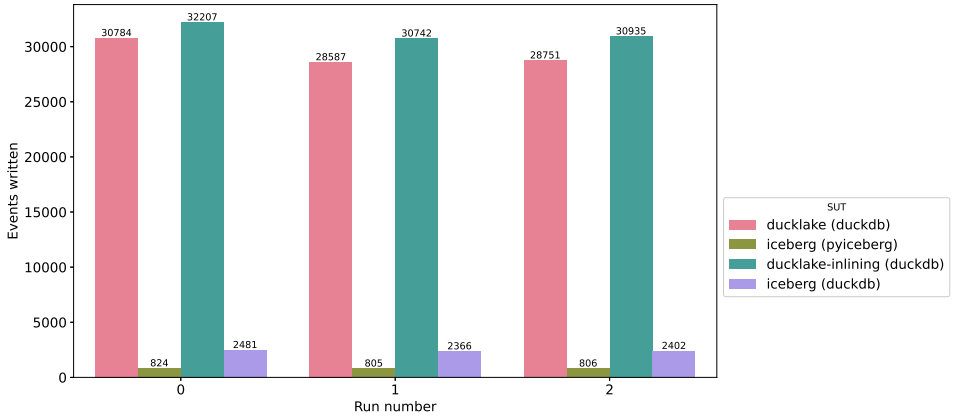

Figure 5.1: Throughput of each SUT in a time span of 5 minutes when writing events one by one. The run numbers correspond to the repetitions of each experiment, where run 0 is considered cold and runs 1 and 2 are considered hot.

Looking at the amount of events written to the different SUTs in Figure 5.1, we can see that DuckLake had a much higher throughput than Iceberg. Using DuckDB did improve Iceberg's throughput, but it was nonetheless much lower than DuckLake's.

Focusing now on the first hot run of each SUT, we see from Figure 5.2

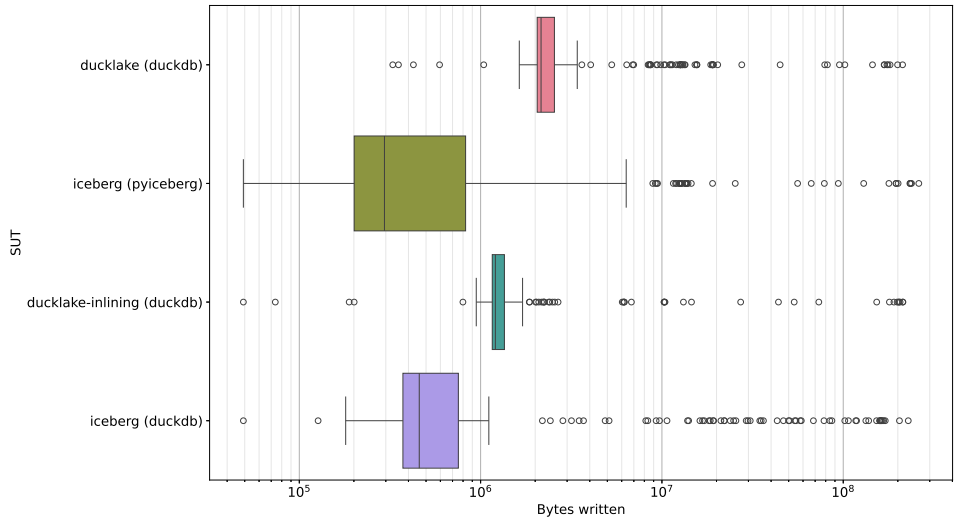

Figure 5.2: Boxplot of bytes written to disk at every second when inserting individual events to each SUT (hot run). The measurements were taken at intervals of 1 second.

SUT	Median (MB)	IQR (MB)
DuckLake	2.154	0.502
Iceberg (PyIceberg)	0.291	0.594
DuckLake + Inlining	1.204	0.193
Iceberg (DuckDB)	0.459	0.380

Table 5.1: Median and interquartile range, in megabytes, of the box plots presented in Figure 5.2.

that DuckLake was often able to write more bytes per second than Iceberg, which is in line with the total events written, shown in Figure 5.1. We can also see from Table 5.1 that the median number of bytes written to DuckLake was about an order of magnitude larger than that of Iceberg with PyIceberg. In addition, Iceberg + PyIceberg seems to have a more unstable rate of bytes/second, which is reflected by the highest interquartile range, i.e. spread, presented in Table 5.1.

5.1.2 Write times

Looking at Figure 5.3, we can see that DuckLake has more consistent write times than Iceberg + PyIceberg, based on the difference in the spread between them. Using Iceberg with DuckDB produces a smaller spread, and thus more consistent write times, but its median is still around an order of magnitude slower than both DuckLake runs. The median write times of each SUT are consistent with their respective amount of bytes/second and events

written (Figures 5.2 and 5.1).

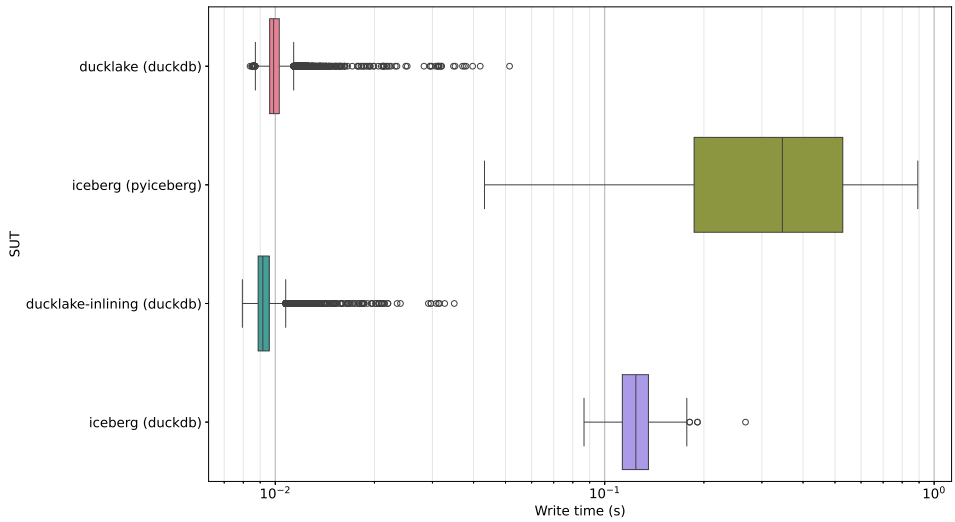

Figure 5.3: Box plots of the time spent writing each individual event to the lakehouse tables during the first hot run, in seconds.

SUT	**Median (ms)**	**IQR (ms)**
DuckLake	9.889	0.667
Iceberg (PyIceberg)	346.208	341.194
DuckLake + Inlining	9.169	0.711
Iceberg (DuckDB)	124.303	22.650

Table 5.2: Median and interquartile range, in milliseconds, of the box plots presented in Figure 5.3.

We observed that the write times for Iceberg increase over time, as shown in Figure 5.4, which explains the larger spread in the data when compared to other SUTs. Although using Iceberg with DuckDB help control the spread of write times, we can see from Figure 5.4 that they still get larger, as more data is written. By contrast, DuckLake's write times seem to stay more or less constant throughout the whole run.

Regarding the DuckLake instance with inlining enabled, we find that the time taken to flush the inlined events to a Parquet file on disk was around 200 milliseconds, as shown in Table 5.3. This is about 20× slower than the median write time of DuckLake without inlining, and about 2× slower than the median of Iceberg with DuckDB. However, DuckLake with inlining enabled was able to outperform the median write time of Iceberg with PyIceberg by around 1.5×.

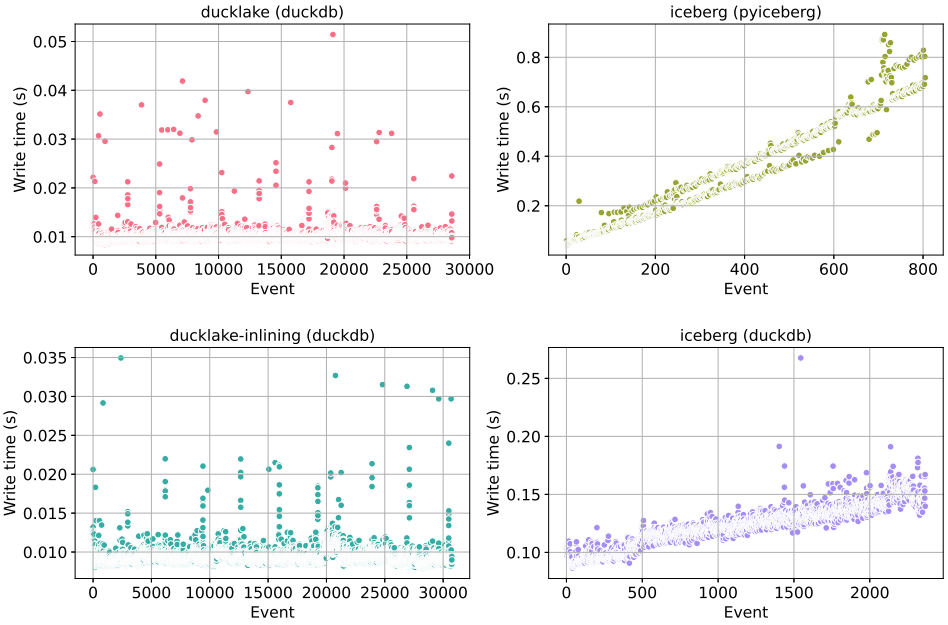

Figure 5.4: Scatter plots of time spent writing individual event to each SUT during the first hot run, in seconds.

Run number	Flush time (ms)
0	219.874
1	205.867
2	193.615

Table 5.3: Time taken by each DuckLake + Inlining run to flush inlined data to a single Parquet file, in milliseconds.

5.1.3 Read times

The box plots in Figure 5.5 go over the time spent reading each individual event from the Kafka broker. We can see that the median read times of DuckLake runs were slightly higher than Iceberg's, but the almost all read times were lower than 1 millisecond, with most times staying between 0.1 and 0.2 milliseconds. These measurements indicate that no system had a major advantage over others due to significantly higher read times.

5.1.4 CPU usage

We observed that Iceberg with DuckDB consistently utilized a higher percentage of the CPU than the other three SUTs. Nonetheless, all systems were mostly stable throughout the runs, except for a few spikes and dips, as shown in Figure 5.6. These observations can be attributed to the noise

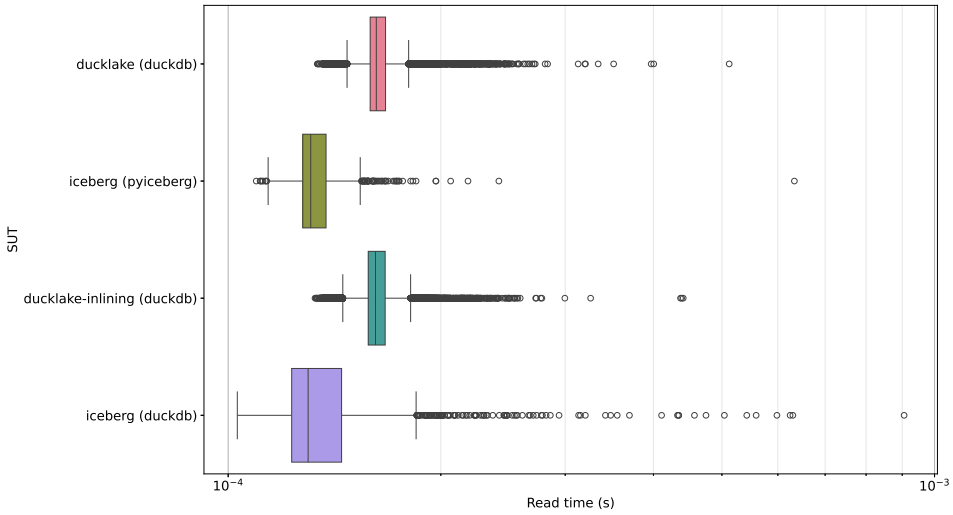

Figure 5.5: Box plots of the time spent reading individual events from Kafka during the first hot run of each SUT, in seconds.

caused by other processes running in the background, such as the Docker containers or even OS processes.

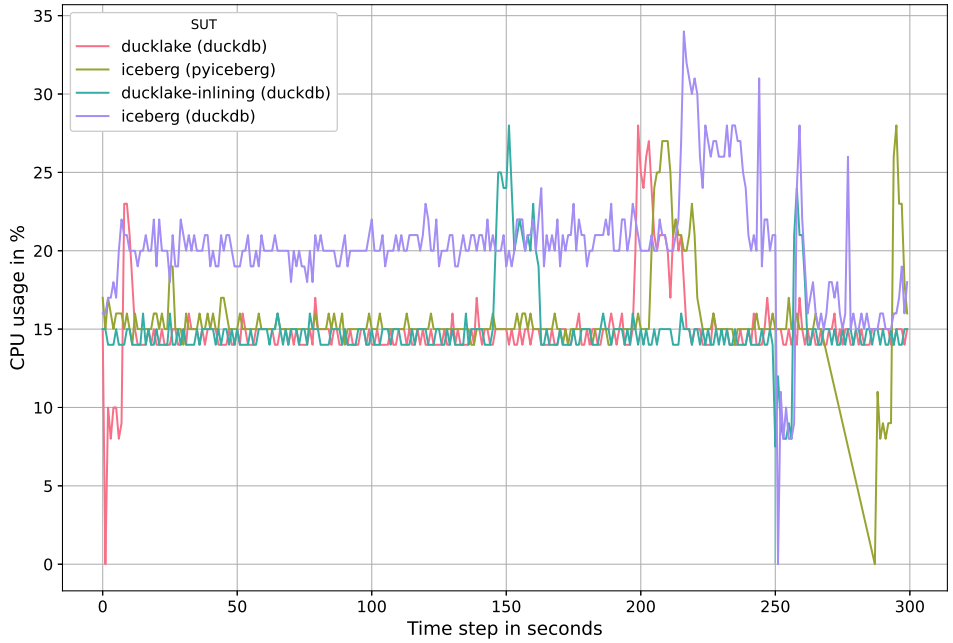

Figure 5.6: CPU usage percentage measured at every second throughout the first hot run of writing individual events to each SUT.

5.2 Writing events in batches of 1,000

5.2.1 Total events written

Figure 5.7 shows the total number of events written to each SUT when writing data in batches of 1,000 events. As observed in the previous experiments, the cold run was slightly better than the hot runs, so we focus on the latter instead.

We can see that all systems performed significantly better than when events were written one by one to the lakehouse tables. The biggest improvement can be noticed when using Iceberg with DuckDB, which managed to surpass DuckLake with inlining enabled. We also observed that inlining led to a decrease of in the amount of events written to DuckLake. That is in contrast to the previous experiments, where inlining had in fact improved the throughput of DuckLake.

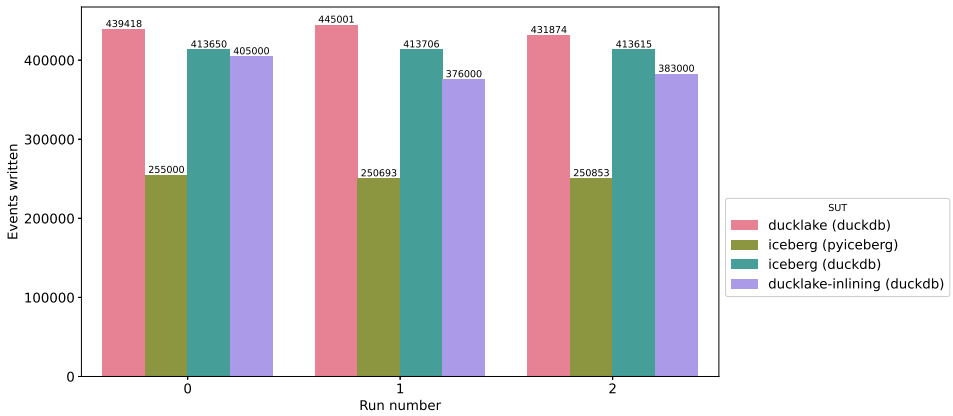

Figure 5.7: Throughput of each SUT in a time span of 5 minutes when writing events in batches of 1,000. The run numbers correspond to the repetitions of each experiment, where run 0 is considered cold and runs 1 and 2 are considered hot.

When looking at the amount of bytes written at every second in Figure 5.8, we see that all systems wrote fewer bytes per second than when writing events one by one. From Table 5.4 we can see that median number of bytes written to disk differed less between the SUTs than in the previous experiment. The exception here was DuckLake + inlining, which had a median about twice as big as the second highest median.

In terms of the spread of the data, both DuckLake instances wrote bytes at a more consistent rate than the Iceberg instances. In contrast to the previous experiment, DuckDB worsened Iceberg's data spread when compared to PyIceberg. In addition, DuckLake with inlining and Iceberg + DuckDB had less stable rates of bytes/second than their counterparts in the previous experiment. Overall, Iceberg + PyIceberg differed the least between writing individual and batched events, whereas all other three SUTs had a

lower median number of bytes written per second.

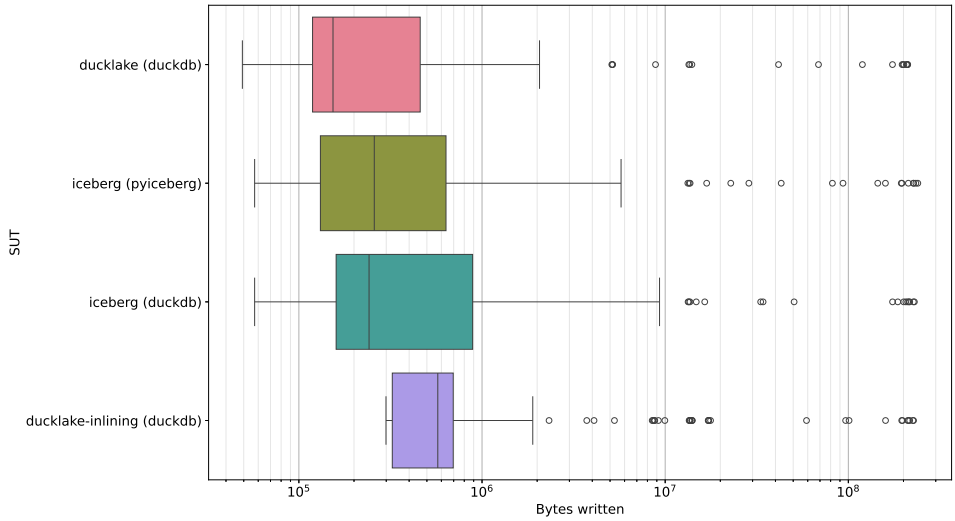

Figure 5.8: Box plots of bytes written to disk at every second when inserting batches of 1,000 events to each SUT (hot run). The measurements were taken at intervals of 1 second.

SUT	Median (MB)	IQR (MB)
DuckLake	0.154	0.341
Iceberg (PyIceberg)	0.258	0.502
Iceberg (DuckDB)	0.242	0.727
DuckLake + Inlining	0.573	0.372

Table 5.4: Median and interquartile range, in megabytes, of the box plots presented in Figure 5.8.

5.2.2 Write times

The write times of the SUTs shown in Figure 5.9 are in line with the results presented in Figure 5.7. We see that Iceberg + PyIceberg had the highest write times of all SUTs, with a median write time about 3× higher than the second highest median (DuckLake + Inlining). For all SUTs the median write time was higher than the previous experiments because every write operation involved batches of 1,000 event, in contrast to individual events in the previous experiment.

Regarding stability of write times, Iceberg + PyIceberg was also least stable system, given by its higher IQR. All the three other systems showed a similar, lower spread in their write times.

With respect to the flush times of DuckLake + inlining, we see in Table 5.6 that it took around 2 seconds to flush the inlined data. This represents a tenfold increase compared to the previous experiment. Furthermore, the flush times were about 3× slower than the highest median write time.

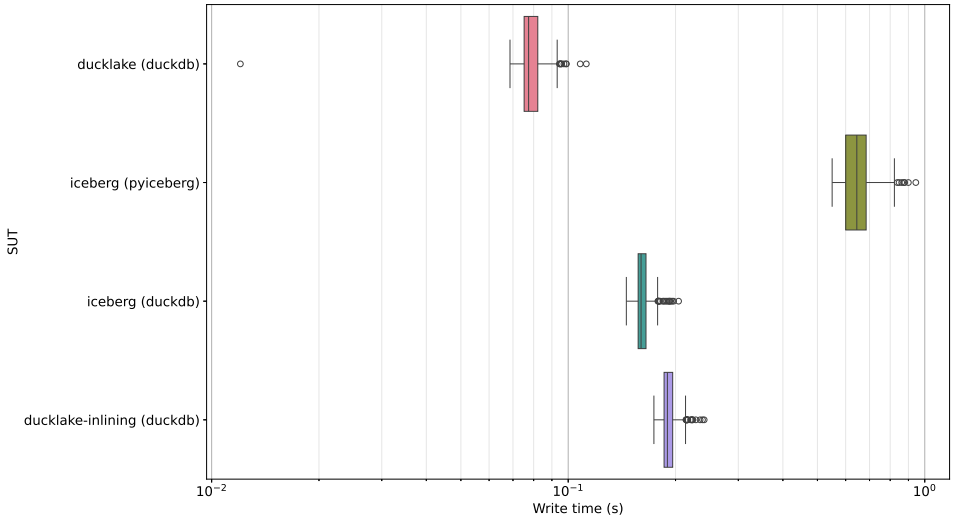

Figure 5.9: Box plots of the time spent writing each batch of 1,000 events to the lakehouse tables during the first hot run, in seconds.

SUT	Median (ms)	IQR (ms)
DuckLake	77.439	6.942
Iceberg (PyIceberg)	644.842	84.695
DuckLake + Inlining	189.746	10.625
Iceberg (DuckDB)	160.086	8.189

Table 5.5: Median and interquartile range, in milliseconds, of the box plots presented in Figure 5.9.

Run number	Flush time (s)
0	2.287
1	1.992
2	1.974

Table 5.6: Time taken by each DuckLake + inlining run to flush the inlined batches of data to a single Parquet file, in seconds.

Looking now at Figure 5.10, we can see that write times behave in a similar way as the previous experiment. When using Iceberg with PyIceberg,

we see the same effect where write times get larger as more events are added to the Iceberg table, which again explains the bigger spread of Iceberg's write times presented in Table 5.5. DuckLake + DuckDB had the lowest and more consistent write times, with the other two SUTs presenting occasional spikes in write times, but staying mostly consistent throughout the experiment run.

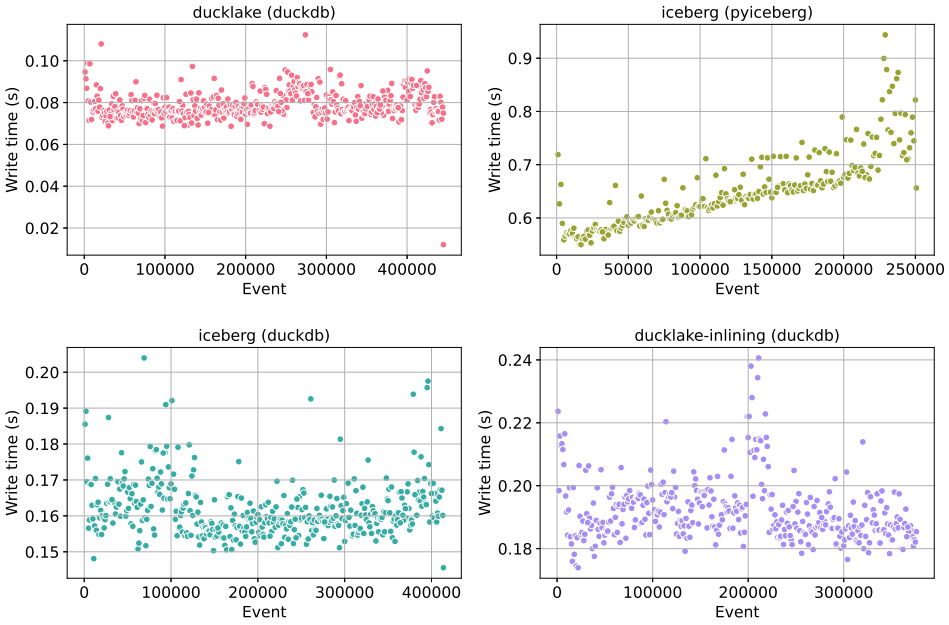

Figure 5.10: Scatter plots of the time spent writing each batch of 1,000 events to each SUT during the first hot run, in seconds.

5.2.3 Read times

The read times of all runs (Figure 5.11) were very similar to each other, with the Q3 for all box plots being below 0.1 milliseconds. We can again confirm that no SUT had an unfair advantage caused by significantly faster read times of our benchmark tool.

5.2.4 CPU usage

In terms of CPU usage, all systems presented very similar percentages throughout their runs, as we can see in Figure 5.12. DuckLake instances showed slightly higher CPU percentages overall than Iceberg. We also see that all four runs had a dip and spike around halfway through, which is likely due to some underlying operation running in the local machine's OS.

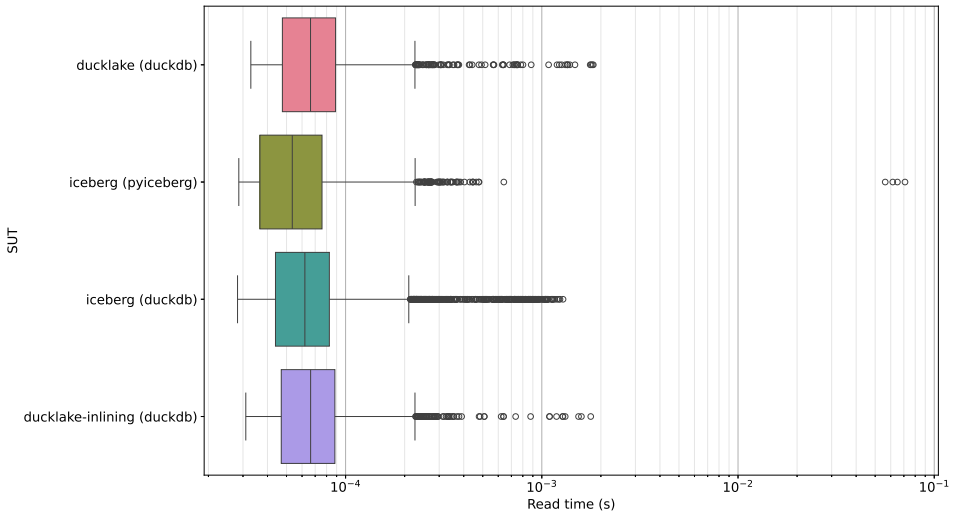

Figure 5.11: Box plots of the time spent reading individual events from Kafka during the first hot run of each SUT, in seconds.

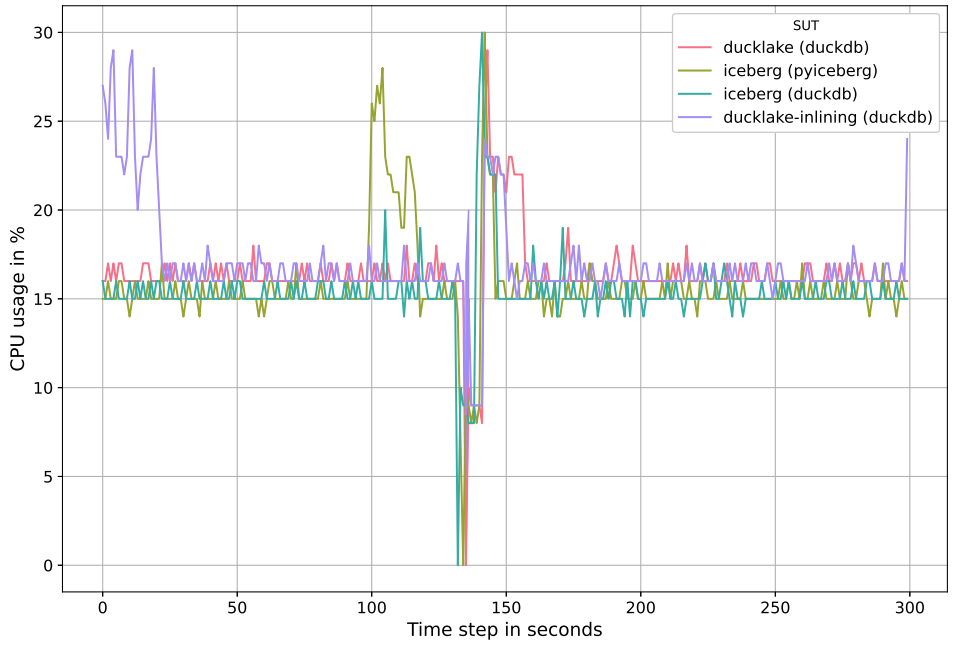

Figure 5.12: CPU usage percentage measured at every second throughout the first hot run of writing batched events to each SUT.

Chapter 6

Discussion

6.1 DuckDB vs. PyIceberg

We would first like to discuss the differences observed between DuckDB and PyIceberg, when running experiments with Iceberg. The difference in throughput between the two clients was considerably large when writing both individual events and batches of events.

6.1.1 Faster write operations

If we look at the experiments where events were written one by one to the Iceberg table, we see that Parquet files written by DuckDB are 2.9 kB in size, while PyIceberg produced files of 8.2 kB. This difference of almost 3× in size is surprising, given that both clients were configured to use snappy compression (Section 4.2.2). We believe this difference might be caused by PyIceberg not applying compression to files that contain few rows. Our reason for this claim is that when writing batches of 1,000 events, both DuckDB and PyIceberg produced Parquet files of similar size, ranging between 37 kB and 40 kB.

We expect similar internal differences to be present in the implementation of DuckDB's and PyIceberg's JSON and Avro writers as well. For instance, Iceberg's metadata JSON files will grow larger with time because they contain all the data from previous metadata files. As a consequence, write times will increase after each append, as we observed in Figure 5.4. However, the write times of DuckDB increased at a much slower rate than PyIceberg's did. We believe this difference is thus linked to a more efficient implementation of DuckDB's metadata writer. This would also explain why DuckDB was still faster than PyIceberg when writing batches of 1,000 events, given that the Parquet file sizes were similar between both clients in that case.

Based on the performance difference between DuckDB and PyIceberg, we focus the rest of our discussion on Iceberg + DuckDB, following the guidelines of fair comparison presented in [38].

6.2 Single events vs. batches

6.2.1 Throughput

Writing events in batches significantly improved the throughput of all systems. This is also reflected by the increase in the median write time relative to the increase in events written per file. While we increased the latter by a factor of 1,000×, the former increased by 20× at most (DuckLake + Inlining). This performance increase is directly tied to how Parquet files work: The file format is meant for efficiently storing compressed columnar data in row groups, so that it can be used for analytics tasks [37]. It also stores metadata to further improve read efficiency. Consequently, the overhead associated with writing a Parquet file makes it more well suited for multiple rows (events) per file, rather than a single row. In fact, the recommended row group size is between 512 MB and 1 GB [36].

Despite the performance gains brought by writing data in batches, we would like to stress the significant differences between Iceberg and DuckLake when writing individual events. The results presented in Figure 5.1 highlight the impact of each system's design on their throughput: Every insert operation in Iceberg will create four files (one data file, one manifest file, one manifest list and one metadata file), and a catalog transaction, that all need to be written sequentially. In addition, the metadata file will become larger each time, thus causing longer write times after each insert (Figure 5.4). By contrast, writing in DuckLake involves writing a Parquet file and performing a transaction to the catalog server. This difference in the design of each system leads to a throughput that is about 12× higher (Figure 5.1).

6.2.2 Total bytes written

When comparing Tables 5.1 and 5.4, we noticed the median for all SUTs was lower when writing batched events than when writing individual events, even though the total number of events was higher in the former setting. This difference can be explained by batched events producing fewer files overall than individual events, as well as Parquet files being compressed. To illustrate this, we looked at the amount of files, and their total size, produced across the three runs of Iceberg + DuckDB and compiled the results in Table 6.1.

Writing events in batches produced almost 6× fewer files, while data files only took roughly twice as much space on disk. These improvements highlight the advantage of Parquet's compression, considering that writing events in batches was able to process about $\frac{413{,}000}{2{,}400} \approx 172$[1] times more events than writing events one by one.

[1]These values were obtained by approximating the total events consumed by Iceberg + DuckDB in Figures 5.1 and 5.7

	Data files		Metadata files	
	Amount	**Size**	**Amount**	**Size**
Individual	7,249	21.4 MB	21,785	5400 MB
Batched	1,242	46.4 MB	3,732	183.5 MB

Table 6.1: Comparison between the amount and total size of data and metadata files produced by DuckDB in Iceberg when writing events one by one (**Individual**) and in batches of 1,000 (**Batched**).

Furthermore, the gains on metadata files were also substantial, where the disk space used dropped from 5.4 GB to 183.5 MB, representing a decrease of about 29× in size.

6.2.3 Data inlining

Interestingly, enabling data inlining for DuckLake worsened its performance when writing batched data. In fact, Iceberg + DuckDB was also able to consistently write more events than DuckLake + inlining. This is a surprising result, because inlining does not write any Parquet files, but rather just stores them in the catalog database as part of the transaction. However, the database still writes that data to disk and, in the case of PostgreSQL, that data is stored in a row-based way [50], whereas the data being inserted has a columnar format. Converting from one representation type to another can lead to overhead, especially for larger batches of data.

In addition, we saw in Table 5.4 that DuckLake with inlining had the highest median number of bytes written to disk. This difference could be explained by the catalog database (PostgreSQL in this case) not compressing the data or compressing it less than the Parquet data. As a consequence, inlining would take more bytes, and thus more time, than directly writing Parquet files for larger batches of data.

It is also worth mentioning that the inlining feature of DuckLake is recommended for small appends/changes by its developers [14]. So although this feature can be configured for any threshold, it is likely to yield better performance gains for lower amounts of inlined data.

6.3 Data inlining trade-off

Although inlining data in DuckLake did not yield better results for batched data, it did improve the throughput of DuckLake by about 7.5% when writing individual results. There is still the cost of eventually flushing the inlined data to disk, but that took only around 200 milliseconds for roughly 30,000 events (Table 5.3). By contrast, if we individually insert 30,000 events to a DuckLake table and then merge the resulting 30,000 files

using DuckLake's `ducklake_merge_adjacent_files` function [15], it takes about 13 seconds. It is important to realize that merging small files together is a recommended maintenance step in lakehouse tables, so that read performance is not hindered [3], [15], [31]. Therefore, the inlining feature of DuckLake offers a big advantage when small, frequent events are inserted into a DuckLake table, by minimizing the need of merging data files.

6.4 Research questions

Sub-question 1 **"What are the current state-of-the-art benchmarks for lakehouses?"** was answered by our literature review in Chapter 3. Our findings show that the current state-of-the-art benchmarks for lakehouse systems focus on read performance, as well as insertion performance of bulk data. In addition, we could not find any previous works that evaluated the throughput performance of lakehouse systems when inserting streams of data into them.

Sub-question 2 **"What are common patterns used in benchmarks for stream processing systems?"** was also answered in Chapter 3, where we found that most benchmarks for evaluating streaming systems use a Kafka broker for ingesting a variety of datasets, and measure the usage of system resources, such as CPU, main memory, network and disk. In addition, many benchmarks measure system throughput and processing latency.

When translating those findings to our research (Chapter 4), we concluded that throughput was the main metric to focus on, but we also collected usage metrics for CPU and disk I/O for a more in depth analysis of the results. We used Kafka to ingest the TLC Trip Record Data dataset into our benchmark, although we believe any dataset with enough records would have sufficed for the purposes of our research.

Sub-question 3 **"How can we ensure our benchmark is actually fair?"** was answered in Chapter 3, where we summarized the work of Raasveldt et al. [38]. We found that there are many aspects to ensure a fair benchmark, such as separating cold and hot runs, reporting hardware and software details for reproducibility and making sure the SUTs provide equivalent functionality.

Sub-questions 4 **"What is the difference in performance of current lakehouse systems when evaluated with our benchmark?"** and 5 **"What is the performance gain of the "data inlining" feature of DuckLake in comparison to other lakehouse formats?"** were both answered in Chapters 5 and 6. The throughput of DuckLake at inserting individual events was much higher than Iceberg's and enabling inlining only increased that difference. When inserting events in batches, Iceberg was able to close in the gap with DuckLake, but the latter still delivered a higher throughput. Inlining also turned out to not be advantageous when writing

batches of data, as it performed worse than DuckLake without inlining. Finally, we learned that using different clients for interacting with lakehouses can have a huge impact in performance due to internal implementation differences.

6.5 Limitations and future work

Although we did our best to make sure experiments were fair, it is important to acknowledge their limitations.

Firstly, the experiments were carried out on a single machine, where all services were running in parallel. This means that processes could interfere with each other when trying to acquire system resources, resulting in noise in the data. We believe this is why we observed outliers in our data, especially when looking at CPU usage (Figures 5.6 and 5.12) and bytes written per second (Figures 5.2 and 5.8).

Additionally, we believe that running experiments at a larger scale could add more value to our results. Lakehouses usually run in distributed cloud environments, where they are connected to a separate catalog server and storage system. However, running experiments in such an environment brings its own challenges, such as orchestrating multiple cloud services, scaling up services accordingly and accounting for network latency. This would be an interesting challenge to tackle by future work.

Furthermore, our research focused on a scenario where only one client inserts data into the lakehouse tables. However, in many real-life scenarios, data streams are produced by multiple clients, resulting in concurrent insert operations. Evaluating lakehouse systems under those conditions would give more insight into how each of them performs when there are frequent conflicting inserts.

Finally, due to unforeseen circumstances, we could not include Delta Lake in our comparisons. Benchmarking it in the future would provide a more complete picture of the current landscape of lakehouses at handling streaming data. Based on our results, we expect Delta to perform better than Iceberg, but worse than DuckLake when inserting individual events. This is based on the way that Delta handles inserts by creating a new log file, and possible checkpoint file, on top of writing Parquet data. This process involves less steps than when inserting into Iceberg tables, and the use of checkpoint files would help keep write performance at a constant rate, since the log files will not grow indefinitely as more data is appended.

Chapter 7

Conclusion

In this work we set out to evaluate the throughput of streaming data in modern lakehouse systems. The final product was a publicly available benchmark tool that can be used to reproduce the results reported in this thesis, as well as evaluate other lakehouses, e.g. Delta Lake, in the future. Other sub-questions have been summarized in Section 6.4.

Our results showed that DuckLake has a clear advantage over Iceberg when constantly inserting individual streaming events. The biggest gain in performance comes from DuckLake's architectural design, which avoids writing metadata to separate files, by using a catalog database for tracking metadata and handling transactions. In addition, Iceberg's throughput decreases over time, due to a constant increase in size of the metadata files, which store all previous snapshots of the Iceberg table.

When writing data in batches, Iceberg's throughput improved significantly, especially when paired with DuckDB, instead of PyIceberg. This last point also highlights the importance of fair benchmarking, since using different clients dramatically changed the results obtained with Iceberg.

When experimenting with the data inlining feature of DuckLake, we observed moderate improvements in its throughput. However, the biggest advantage of inlining is that the data can be flushed faster to a single Parquet file, than having to merge all individual Parquet files after they have been inserted. This reduces the need of running compaction of data files and benefits read performance. When it comes to inserting batched data however, the inlining feature decreased DuckLake's performance, which is likely related to the catalog database converting data from a columnar representation to a row-based one.

Based on these results, we would recommend using DuckLake with data inlining enabled instead of Iceberg for writing individual streaming events to a lakehouse table at the highest throughput. When writing data in batches, we would also advise using DuckLake, but without data inlining, as we found that Iceberg's metadata files will grow in size with more inserts, thus slowing

down the system over time.

Finally, we would like to point out that lakehouses are relatively new and evolving. Consequently, there are still few works that evaluate them, and the existing ones focus on their read performance and bulk data loading tasks. Our work is a first attempt at gauging the ability of lakehouse systems at consuming continuous streams of data. We hope to provide a baseline for similar benchmarks in the future.

Bibliography

[1] *Apache Arrow - Frequently Asked Questions*, English. Accessed: Feb. 2, 2026. [Online]. Available: https://arrow.apache.org/faq/

[2] *Apache Arrow - Streaming, Serialization, and IPC*, English. Accessed: Feb. 3, 2026. [Online]. Available: https://arrow.apache.org/docs/python/ipc.html

[3] Apache Iceberg, *Maintenance*, English. Accessed: Nov. 11, 2025. [Online]. Available: https://iceberg.apache.org/docs/latest/maintenance/

[4] Apache Iceberg, *Reliability*, English. Accessed: Oct. 22, 2025. [Online]. Available: https://iceberg.apache.org/docs/latest/reliability/

[5] Apache Iceberg, *REST Catalog Spec*, en, Oct. 2025. Accessed: Oct. 15, 2025. [Online]. Available: https://iceberg.apache.org/rest-catalog-spec/

[6] *Apache Kafka*, Nov. 2025. [Online]. Available: https://github.com/apache/kafka

[7] A. Arasu et al., "Linear road: A stream data management benchmark," in *Proceedings of the thirtieth international conference on very large data bases - volume 30*, ser. Vldb '04, Number of pages: 12, Toronto, Canada: VLDB Endowment, 2004, pp. 480–491, ISBN: 0-12-088469-0.

[8] M. Armbrust et al., "Delta lake: High-performance ACID table storage over cloud object stores," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3411–3424, Aug. 2020, ISSN: 2150-8097. DOI: 10.14778/3415478.3415560 Accessed: Jun. 27, 2025. [Online]. Available: https://dl.acm.org/doi/10.14778/3415478.3415560

[9] Avril Aysha, *Delta Lake vs Data Lake - What's the Difference?* English. Accessed: Jan. 22, 2026. [Online]. Available: https://delta.io/blog/delta-lake-vs-data-lake/

[10] J. Camacho-Rodríguez et al., "LST-Bench: Benchmarking Log-Structured Tables in the Cloud," en, *Proceedings of the ACM on Management of Data*, vol. 2, no. 1, pp. 1–26, Mar. 2024, ISSN: 2836-6573. DOI: 10.1145/3639314 Accessed: Sep. 8, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3639314

[11] *DuckDB Documentation - Lakehouse Formats*, English, Oct. 2025. Accessed: Feb. 2, 2026. [Online]. Available: https://blobs.duckdb.org/docs/duckdb-docs.pdf

[12] *Ducklake Documentation - Choosing a Catalog Database*, English, Oct. 2025. Accessed: Jan. 27, 2026. [Online]. Available: https://ducklake.select/docs/stable/duckdb/usage/choosing_a_catalog_database

[13] *DuckLake Documentation - Conflict Resolution*, English, Oct. 2025. Accessed: Jan. 27, 2026. [Online]. Available: https://ducklake.select/docs/stable/duckdb/advanced_features/conflict_resolution

[14] *DuckLake Documentation - Data Inlining*, English, Oct. 2025. Accessed: Jan. 27, 2026. [Online]. Available: https://ducklake.select/docs/stable/duckdb/advanced_features/data_inlining

[15] *Ducklake Documentation - Merge Adjacent Files*, English, Oct. 2025. Accessed: Feb. 19, 2026. [Online]. Available: https://ducklake.select/docs/stable/duckdb/maintenance/merge_adjacent_files

[16] *Ducklake Documentation - Tables*, English, Oct. 2025. Accessed: Jan. 27, 2026. [Online]. Available: https://ducklake.select/docs/stable/specification/tables/overview

[17] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, "A survey on the evolution of stream processing systems," en, *The VLDB Journal*, vol. 33, no. 2, pp. 507–541, Mar. 2024, ISSN: 1066-8888, 0949-877X. DOI: 10.1007/s00778-023-00819-8 Accessed: Sep. 15, 2025. [Online]. Available: https://link.springer.com/10.1007/s00778-023-00819-8

[18] Giampaolo Rodola, *Psutil*, Jan. 2026. [Online]. Available: https://github.com/giampaolo/psutil

[19] Guillermo Sanchez and Gabor Szarnyas, *DuckLake 0.3 with Iceberg Interoperability and Geometry Support*, English, Sep. 2025. Accessed: Nov. 21, 2025. [Online]. Available: https://ducklake.select/2025/09/17/ducklake-03/

[20] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," en, *ACM Computing Surveys*, vol. 15, no. 4, pp. 287–317, Dec. 1983, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/289.291 Accessed: Jan. 22, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/289.291

[21] S. Henning and W. Hasselbring, "Theodolite: Scalability Benchmarking of Distributed Stream Processing Engines in Microservice Architectures," en, *Big Data Research*, vol. 25, p. 100 209, Jul. 2021, ISSN: 22145796. DOI: 10.1016/j.bdr.2021.100209 Accessed: Sep. 15, 2025. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2214579621000265

[22] S. Henning, A. Vogel, M. Leichtfried, O. Ertl, and R. Rabiser, "ShuffleBench: A Benchmark for Large-Scale Data Shuffling Operations with Distributed Stream Processing Frameworks," en, in *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, London United Kingdom: ACM, May 2024, pp. 2–13, ISBN: 979-8-4007-0444-4. DOI: 10.1145/3629526.3645036 Accessed: Sep. 15, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3629526.3645036

[23] *How to prevent object overwrites with conditional writes*, English. Accessed: Jan. 26, 2026. [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/userguide/conditional-writes.html

[24] *Iceberg*, Dec. 2025. [Online]. Available: https://github.com/apache/iceberg

[25] *Iceberg Spec - Optimistic Concurrency*, English. Accessed: Jan. 26, 2026. [Online]. Available: https://iceberg.apache.org/spec/?h=concurren#optimistic-concurrency

[26] A. M. Jason Hughes, *Apache Iceberg: The Definitive Guide*, eng. O'Reilly Media, Inc, 2024, OCLC: 1425185829, ISBN: 978-1-0981-4861-4.

[27] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," English, in *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, vol. 6, San Francisco, CA, USA: USENIX Association, Dec. 2004, pp. 137–150. Accessed: Nov. 12, 2025. [Online]. Available: https://www.usenix.org/legacy/events/osdi04/tech/full_papers/dean/dean.pdf

[28] Jerry Peng et al., *Introducing Real-Time Mode in Apache Spark™ Structured Streaming*, English, Aug. 2025. Accessed: Feb. 27, 2026. [Online]. Available: https://www.databricks.com/blog/introducing-real-time-mode-apache-sparktm-structured-streaming

[29] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking Distributed Stream Data Processing Systems," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, Paris: IEEE, Apr. 2018, pp. 1507–1518, ISBN: 978-1-5386-5520-7. DOI: 10.1109/ICDE.2018.00169 Accessed: Sep. 15, 2025.

[30] Mark Raasveldt and Hannes Mühleisen. "The DuckLake manifesto: SQL as a lakehouse format," DuckLake, Accessed: Jun. 27, 2025. [Online]. Available: https://ducklake.select/manifesto/

[31] Matthew Powers, *Delta Lake Small File Compaction with OPTIMIZE*, English, Jan. 2023. Accessed: Feb. 19, 2026. [Online]. Available: https://delta.io/blog/2023-01-25-delta-lake-small-file-compaction-optimize/

[32] M. Merli, S. Guo, P. Li, H. Chen, and N. Lu, "Ursa: A Lakehouse-Native Data Streaming Engine for Kafka," en, *Proceedings of the VLDB Endowment*, vol. 18, no. 12, pp. 5184–5196, Aug. 2025, ISSN: 2150-8097. DOI: 10.14778/3750601.3750636 Accessed: Sep. 18, 2025. [Online]. Available: https://dl.acm.org/doi/10.14778/3750601.3750636

[33] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia, "Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics," presented at the CIDR, 2021. Accessed: Jun. 26, 2025. [Online]. Available: https://vldb.org/cidrdb/2021/lakehouse-a-new-generation-of-open-platforms-that-unify-data-warehousing-and-advanced-analytics.html

[34] opensource@google.com, *Snappy*, Mar. 2025. [Online]. Available: https://github.com/google/snappy

[35] Paras Jain, Peter Kraft, Conor Power, Tathagata Das, Ion Stoica, and Matei Zaharia, "Analyzing and comparing lakehouse storage systems," presented at the CIDR, Amsterdam, The Netherlands, 2023. [Online]. Available: https://vldb.org/cidrdb/2023/analyzing-and-comparing-lakehouse-storage-systems.html

[36] *Parquet - Configurations*, English, Mar. 2024. Accessed: Feb. 18, 2026. [Online]. Available: https://parquet.apache.org/docs/file-format/configurations/

[37] *Parquet - Motivation*, English, Mar. 2022. Accessed: Feb. 18, 2026. [Online]. Available: https://parquet.apache.org/docs/overview/motivation/

[38] M. Raasveldt, P. Holanda, T. Gubner, and H. Mühleisen, "Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing," en, in *Proceedings of the Workshop on Testing Database Systems*, Houston TX USA: ACM, Jun. 2018, pp. 1–6, ISBN: 978-1-4503-5826-2. DOI: 10.1145/3209950.3209955 Accessed: Sep. 8, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3209950.3209955

[39] *Request preconditions*, English, Jan. 2026. Accessed: Jan. 26, 2026. [Online]. Available: https://docs.cloud.google.com/storage/docs/request-preconditions

[40] J. Schneider, C. Gröger, A. Lutsch, H. Schwarz, and B. Mitschang, "The lakehouse: State of the art on concepts and technologies," *SN Computer Science*, vol. 5, no. 5, p. 449, Apr. 18, 2024, ISSN: 2661-8907. DOI: 10.1007/s42979-024-02737-0 Accessed: Jun. 23, 2025. [Online]. Available: https://link.springer.com/10.1007/s42979-024-02737-0

[41] *Specifying conditional headers for Blob service operations*, English, Nov. 2025. Accessed: Jan. 26, 2026. [Online]. Available: https://learn.microsoft.com/en-us/rest/api/storageservices/specifying-conditional-headers-for-blob-service-operations

[42] J. Tahir, R. Mayer, C. Doblander, and H.-A. Jacobsen, "How Reliable are Streams? End-to-End Processing-Guarantee Validation and Performance Benchmarking of Stream Processing Systems," *Proceedings of the VLDB Endowment*, vol. 18, no. 3, pp. 585–598, Nov. 2024, ISSN: 2150-8097. DOI: 10.14778/3712221.3712227 Accessed: Sep. 18, 2025.

[43] Tathagata Das and Denny Lee, *Delta 2.0 - The Foundation of your Data Lakehouse is Open*, English, Aug. 2022. Accessed: Oct. 22, 2025. [Online]. Available: https://delta.io/blog/2022-08-02-delta-2-0-the-foundation-of-your-data-lake-is-open/

[44] Taxi & Limousine Comission, *TLC Trip Record Data*, English. Accessed: Feb. 3, 2026. [Online]. Available: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[45] The Delta Lake Project Authors, *Delta Lake*, Jan. 2026. Accessed: Jan. 23, 2026. [Online]. Available: https://github.com/delta-io/delta

[46] *Time — Time access and conversions*. Accessed: Feb. 13, 2026. [Online]. Available: https://docs.python.org/3/library/time.html#time.perf_counter

[47] *TPC BENCHMARK DS*, English, Nov. 2024. Accessed: Sep. 11, 2025. [Online]. Available: https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-DS_v4.0.0.pdf

[48] Tyler Akidau, *Streaming 101: The world beyond batch*, English, Aug. 2015. Accessed: Nov. 12, 2025. [Online]. Available: https://www.oreilly.com/radar/the-world-beyond-batch-streaming-101/

[49] *Unity Catalog Structure*, English, Jul. 2024. Accessed: Jan. 26, 2026. [Online]. Available: https://docs.unitycatalog.io/quickstart/

[50] User:Simon, *PostgreSQL Wiki - ColumnOrientedSTorage*, English, Apr. 2014. Accessed: Feb. 20, 2026. [Online]. Available: https://wiki.postgresql.org/wiki/ColumnOrientedSTorage

[51] *What is a data lake?* English. Accessed: Jan. 22, 2026. [Online]. Available: https://www.cloudflare.com/learning/cloud/what-is-a-data-lake/

[52] Q. Yang and H. N. Koutsopoulos, "A Microscopic Traffic Simulator for evaluation of dynamic traffic management systems," en, *Transportation Research Part C: Emerging Technologies*, vol. 4, no. 3, pp. 113–129, Jun. 1996, ISSN: 0968090X. DOI: 10.1016/S0968-090X(96)00006-X Accessed: Jan. 30, 2026. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0968090X9600006X

[53] W. Yue, M. Boissier, and T. Rabl, "A Survey of Stream Processing System Benchmarks," en, in *Performance Evaluation and Benchmarking*, R. Nambiar and M. Poess, Eds., vol. 15337, Series Title: Lecture Notes in Computer Science, Cham: Springer Nature Switzerland, Jul. 2025, pp. 24–43, ISBN: 978-3-031-93857-3 978-3-031-93858-0. DOI: 10.1007/978-3-031-93858-0_2 Accessed: Sep. 12, 2025. [Online]. Available: https://link.springer.com/10.1007/978-3-031-93858-0_2

Appendix A

Additional results

A.1 Writing events one by one

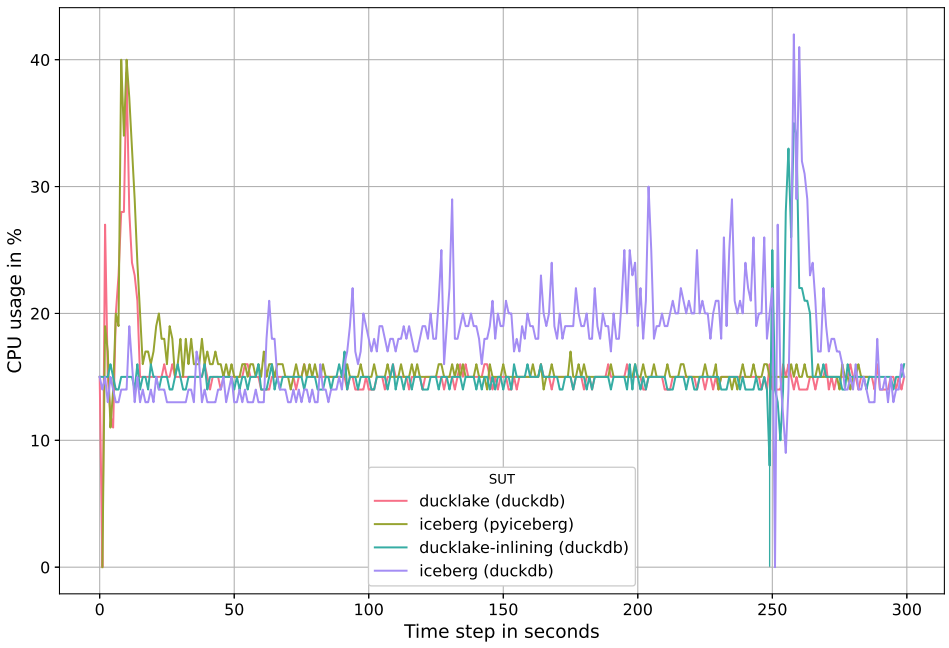

Figure A.1: CPU usage percentage measured at every second throughout the cold run of writing individual events to each SUT.

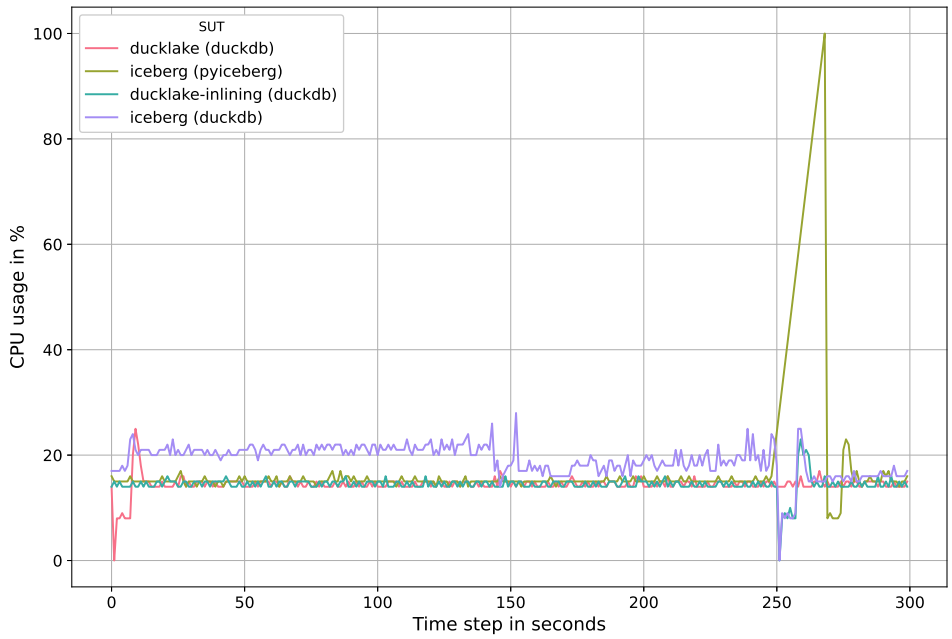

Figure A.2: CPU usage percentage measured at every second throughout the second hot run of writing individual events to each SUT.

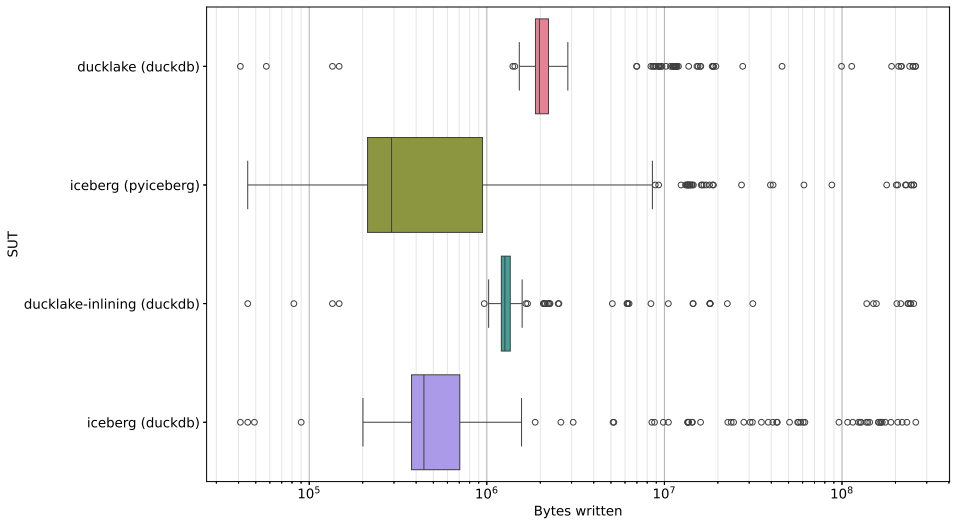

Figure A.3: Boxplot of bytes written to disk at every second when inserting individual events to each SUT (cold run). The measurements were taken at intervals of 1 second.

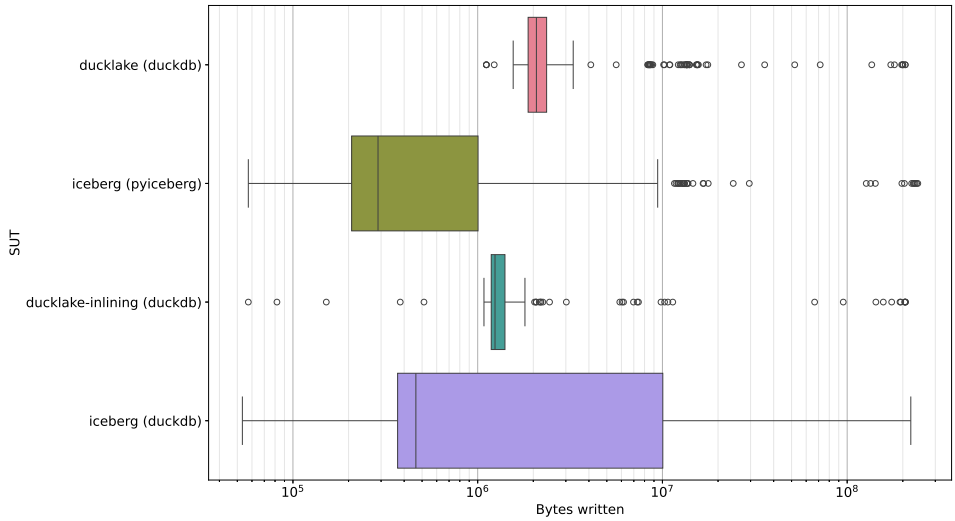

Figure A.4: Boxplot of bytes written to disk at every second when inserting individual events to each SUT (second hot run). The measurements were taken at intervals of 1 second.

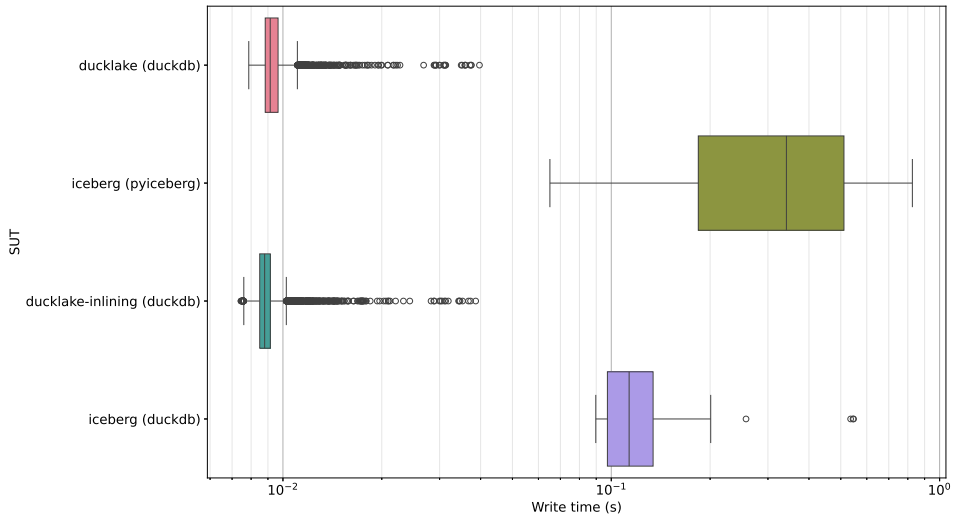

Figure A.5: Box plots of the time spent writing each individual event to the lakehouse tables during the cold run, in seconds.

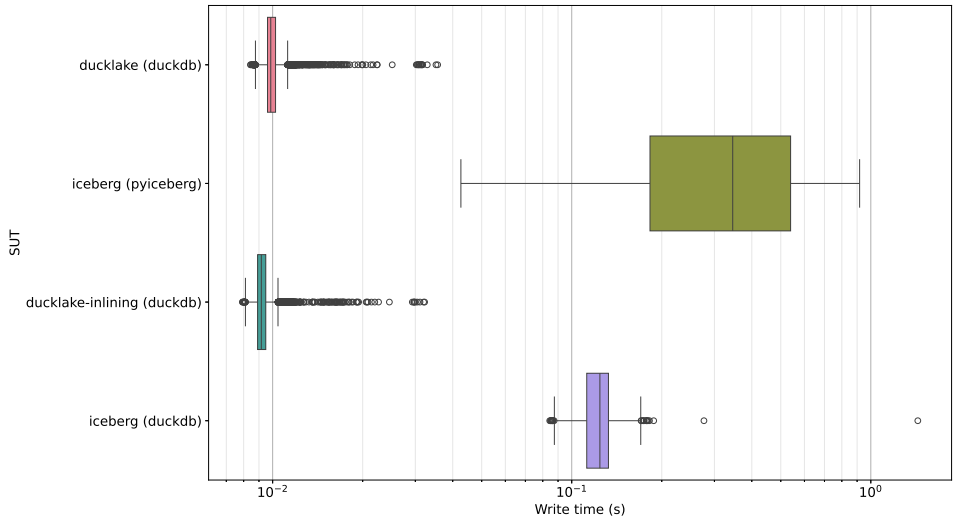

Figure A.6: Box plots of the time spent writing each individual event to the lakehouse tables during the second hot run, in seconds.

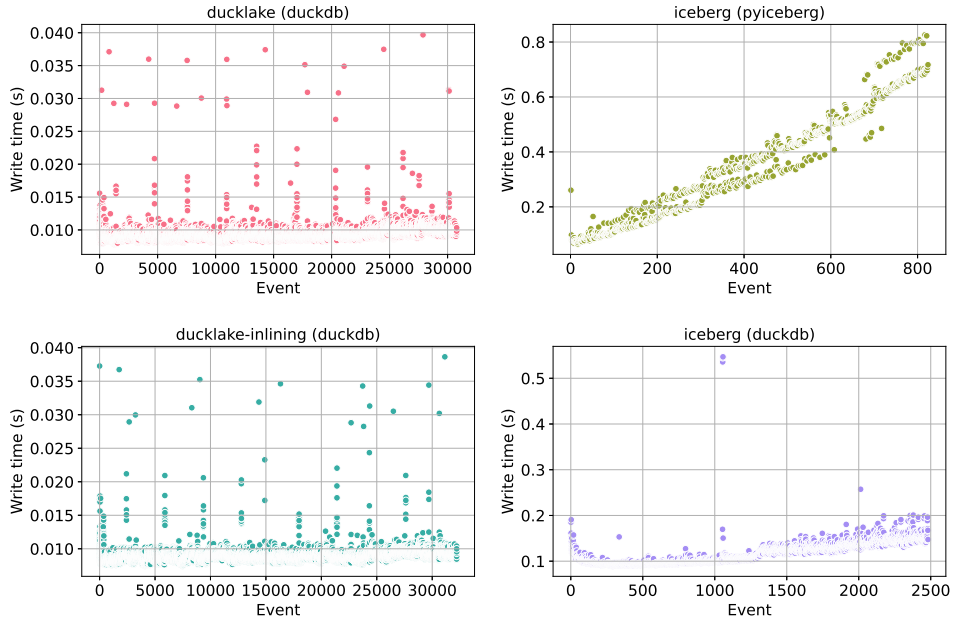

Figure A.7: Scatter plots of time spent writing individual event to each SUT during the cold run, in seconds.

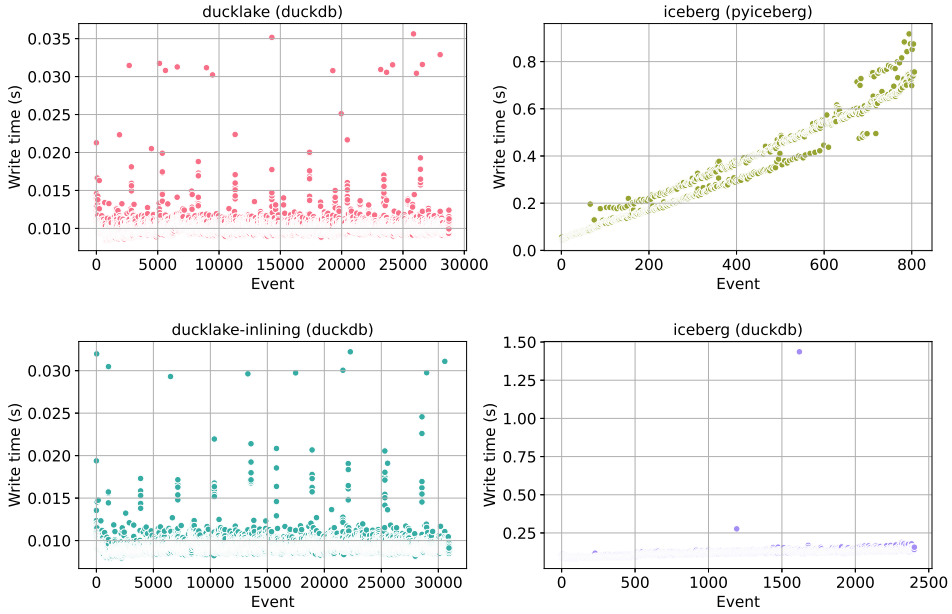

Figure A.8: Scatter plots of time spent writing individual event to each SUT during the second hot run, in seconds.

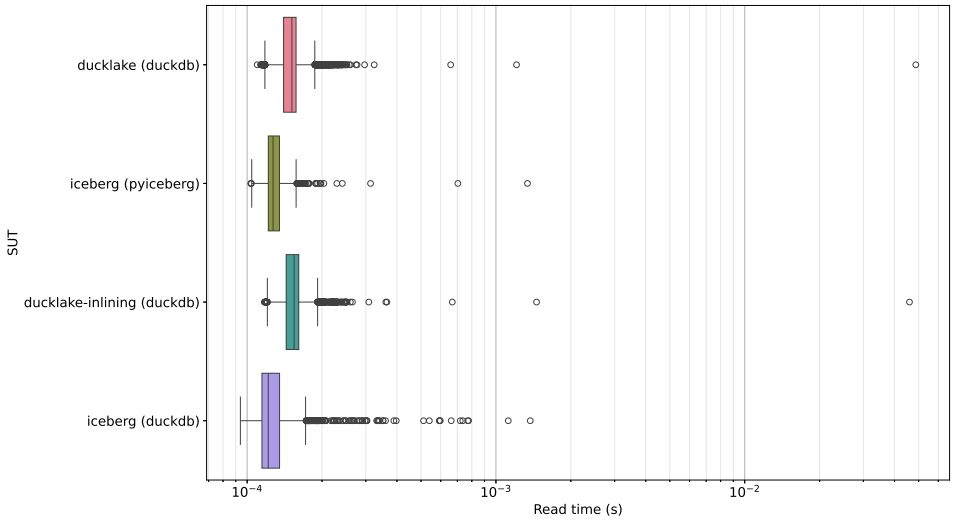

Figure A.9: Box plots of the time spent reading individual events from Kafka during the cold run of each SUT, in seconds.

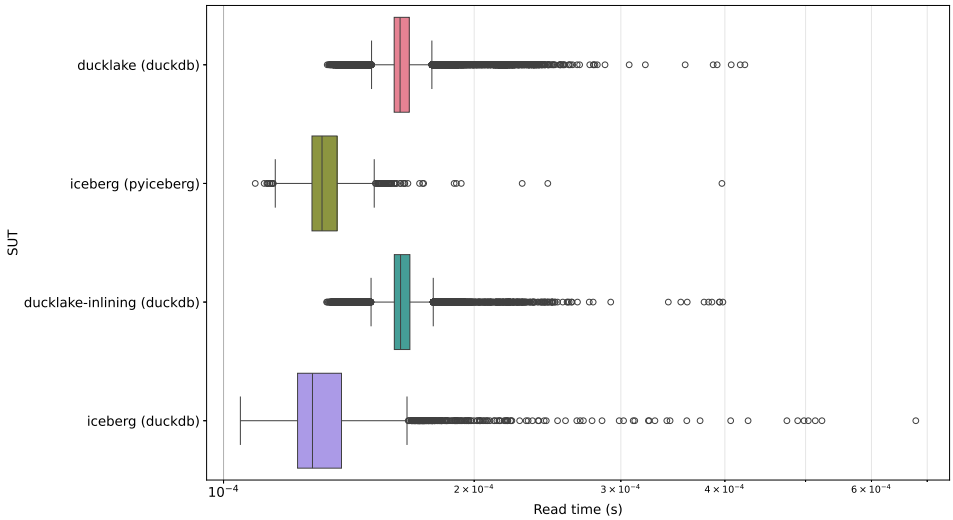

Figure A.10: Box plots of the time spent reading individual events from Kafka during the second hot run of each SUT, in seconds.

A.2 Writing events in batches of 1,000

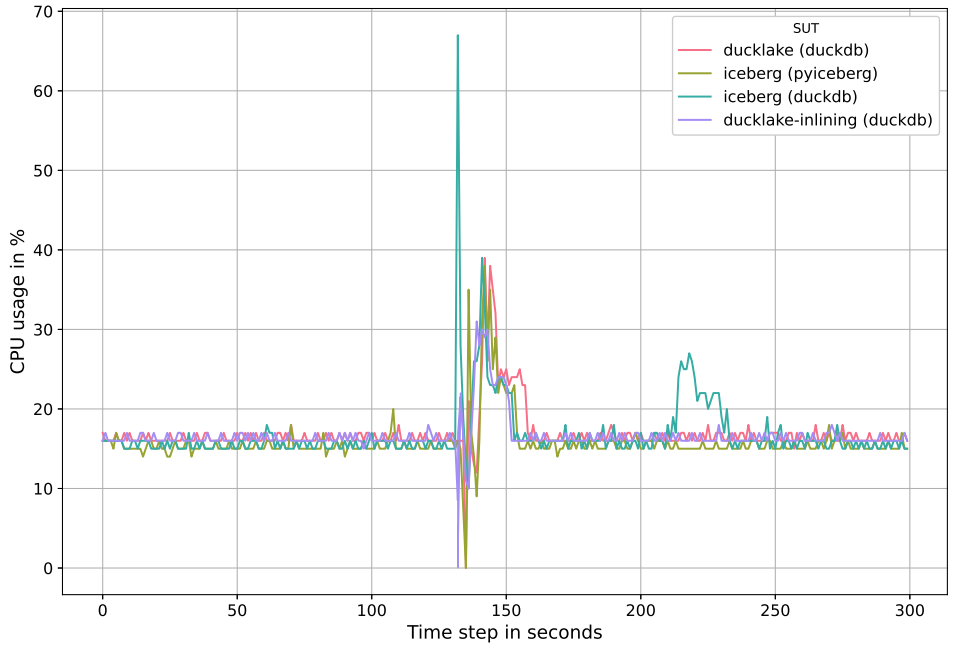

Figure A.11: CPU usage percentage measured at every second throughout the cold run of writing individual events to each SUT.

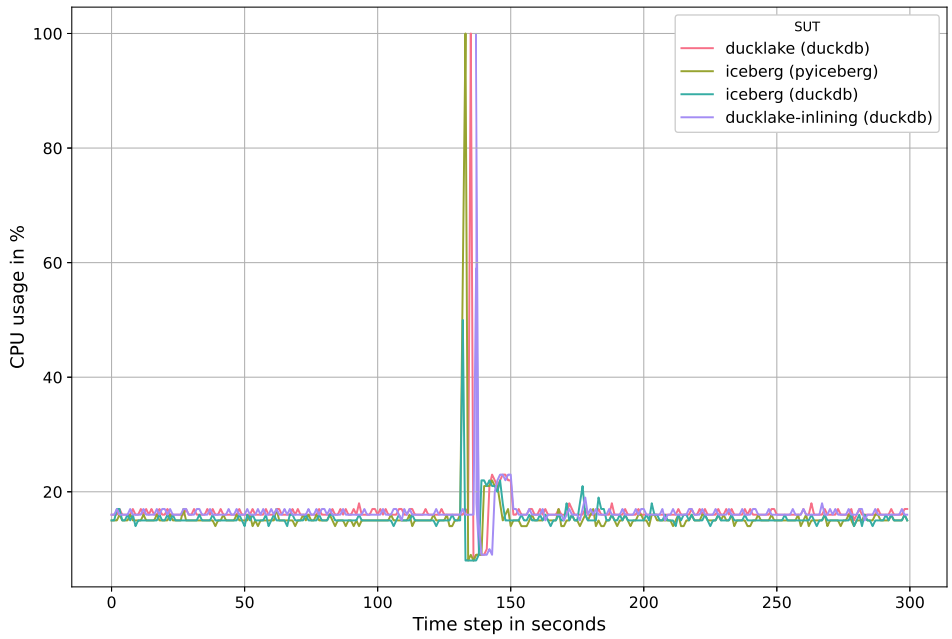

Figure A.12: CPU usage percentage measured at every second throughout the second hot run of writing individual events to each SUT.

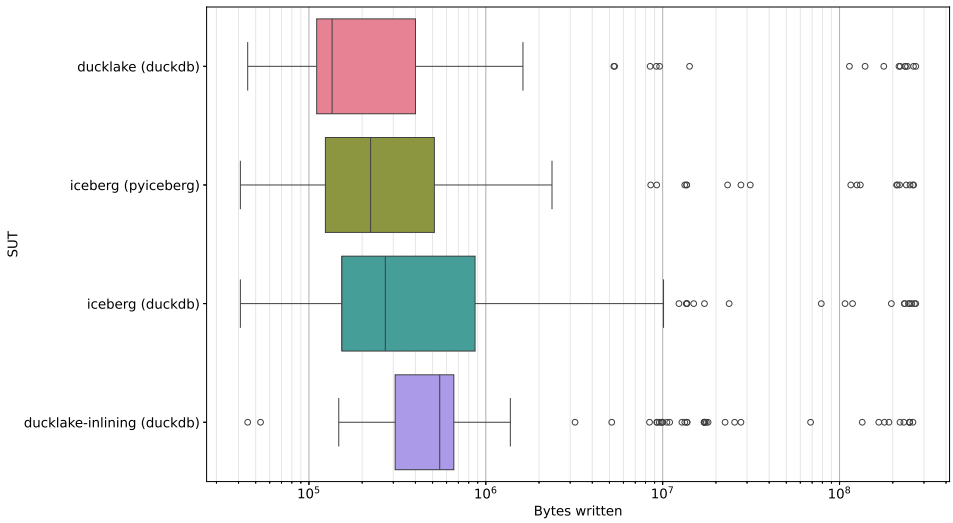

Figure A.13: Boxplot of bytes written to disk at every second when inserting individual events to each SUT (cold run). The measurements were taken at intervals of 1 second.

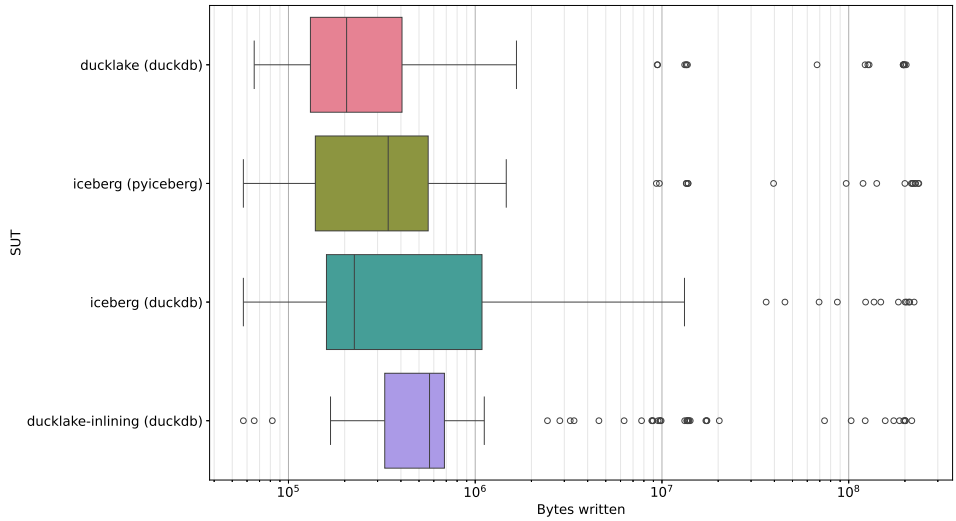

Figure A.14: Boxplot of bytes written to disk at every second when inserting individual events to each SUT (second hot run). The measurements were taken at intervals of 1 second.

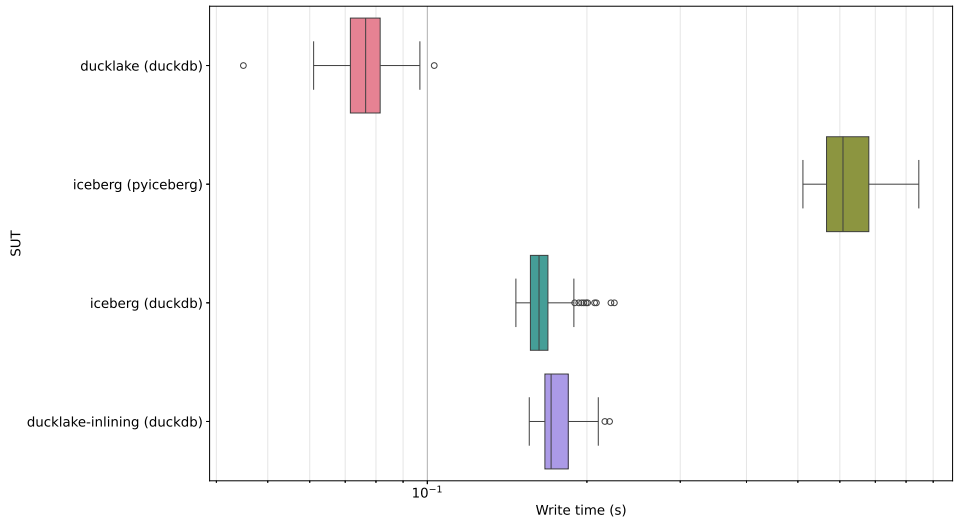

Figure A.15: Box plots of the time spent writing each individual event to the lakehouse tables during the cold run, in seconds.

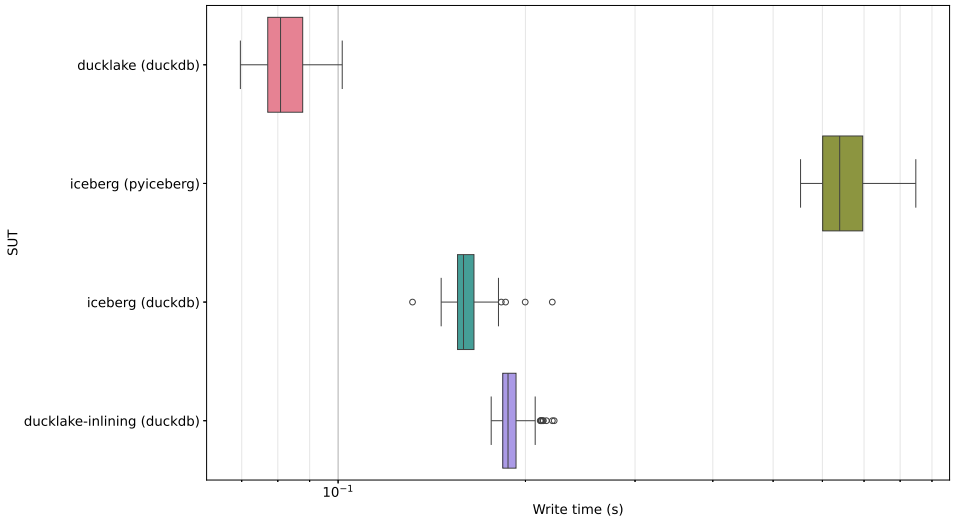

Figure A.16: Box plots of the time spent writing each individual event to the lakehouse tables during the second hot run, in seconds.

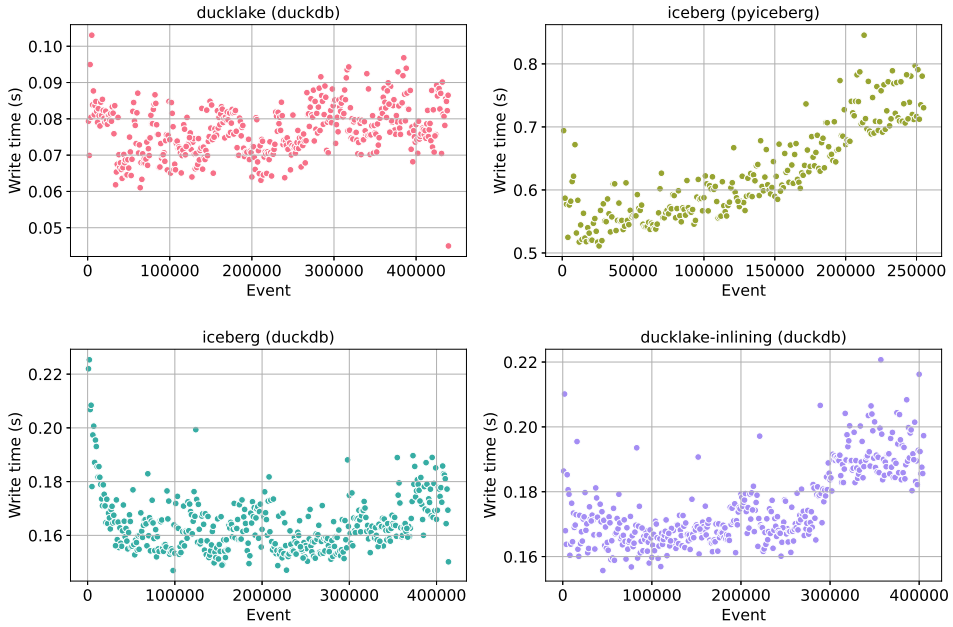

Figure A.17: Scatter plots of time spent writing individual event to each SUT during the cold run, in seconds.

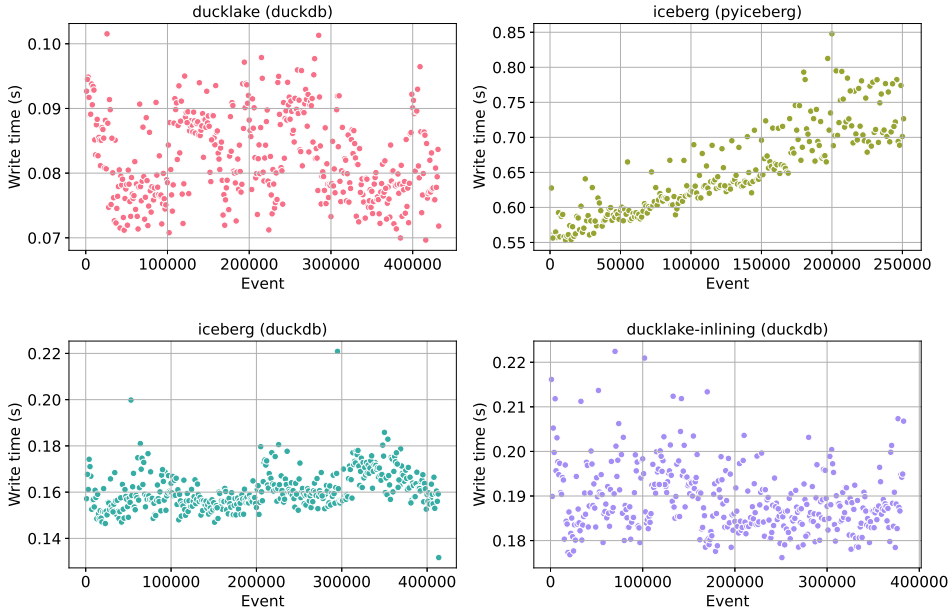

Figure A.18: Scatter plots of time spent writing individual event to each SUT during the second hot run, in seconds.

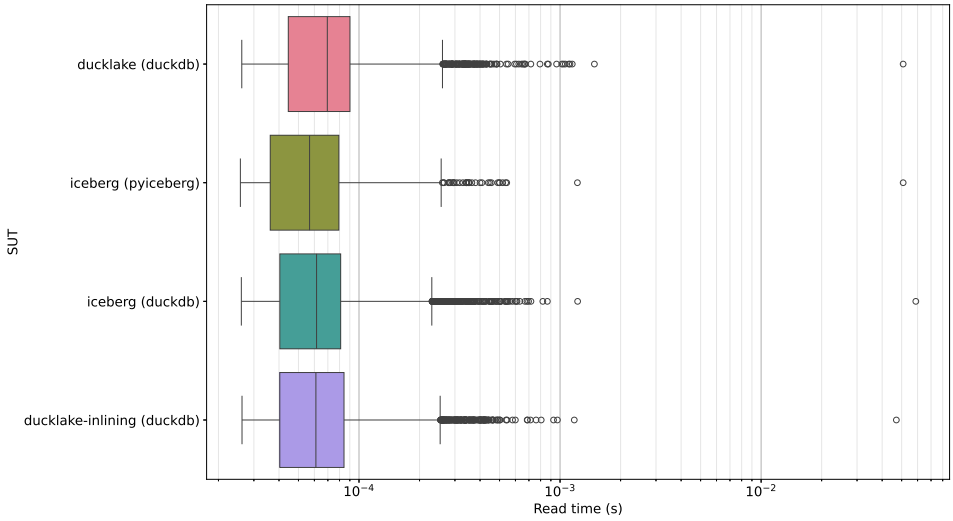

Figure A.19: Box plots of the time spent reading individual events from Kafka during the cold run of each SUT, in seconds.

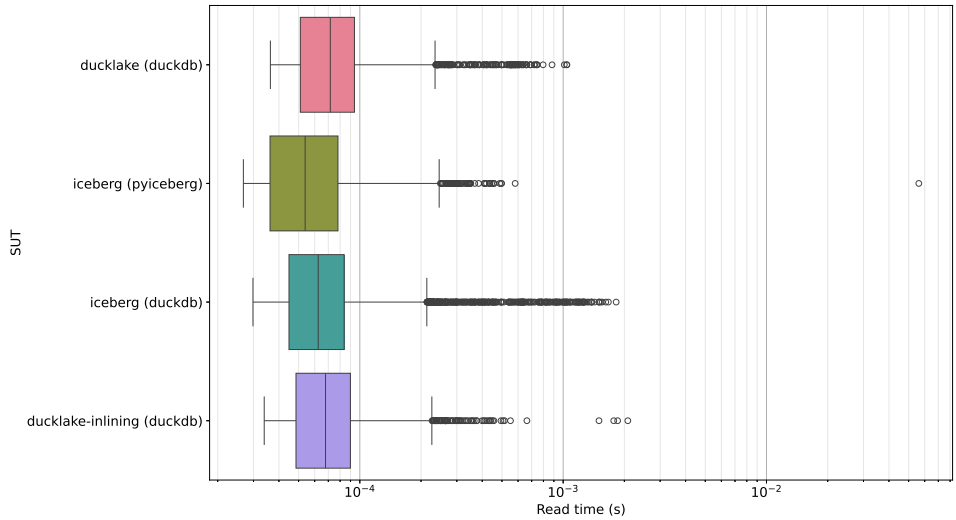

Figure A.20: Box plots of the time spent reading individual events from Kafka during the second hot run of each SUT, in seconds.