Analyzing and Comparing Lakehouse Storage Systems

Paras Jain*[1], Peter Kraft*[2], Conor Power*[1], Tathagata Das[3], Ion Stoica[1], Matei Zaharia[2]
[1]UC Berkeley, [2]Stanford University, [3]Databricks

ABSTRACT

Lakehouse storage systems that implement ACID transactions and other management features over data lake storage have rapidly grown in popularity. For example, over 70% of writes on the Databricks cloud platform are to transactional Delta Lake tables, and companies such as Uber and Netflix run their whole analytics stacks on lakehouse systems like Apache Hudi and Apache Iceberg. These open storage systems with rich management features promise to simplify management of large datasets, accelerate SQL workloads, and offer fast, direct file access for other workloads, such as machine learning. However, the research community has not explored the tradeoffs in designing lakehouse systems in detail. In this paper, we analyze the designs of the three most popular lakehouse storage systems—Delta Lake, Apache Hudi and Apache Iceberg—and compare their performance and features among varying axes based on these designs. We also release a simple benchmark, LHBench, that researchers can use to compare other designs. The benchmark is available at https://github.com/lhbench/lhbench.

1 INTRODUCTION

The past few years have seen the rise of a new type of analytical data management system, the *lakehouse*, that combines the benefit of low-cost, open-format data lakes and high-performance, transactional data warehouses [20]. These systems center around open storage formats such as Delta Lake [19], Apache Hudi [2] and Apache Iceberg [4] that implement transactions, indexing and other DBMS functionality on top of low-cost data lake storage (e.g., Amazon S3), while remaining directly readable from any processing engine. Lakehouse systems are quickly replacing traditional data lakes: for example, over 70% of bytes written on Databricks go into Delta Lake tables [19], many companies have standardized on various lakehouse formats for their data lakes [1, 25], and cloud data services such as Azure Synapse and Amazon EMR are adding support for them [6, 15]. There is also an increasing amount of research on improving lakehouse-like systems [18, 21, 24, 28].

Nonetheless, the tradeoffs in designing a lakehouse storage system are not well-studied. In this paper, we analyze and compare the three most popular open source lakehouse systems—Delta Lake, Hudi and Iceberg—to highlight some of these tradeoffs and identify areas for future work. We compare the systems quantitatively using a simple benchmark based on TPC-DS that exercises the areas where they make different design choices (LHBench), and qualitatively based on features such as transaction semantics. We plan to open source LHBench for use by other researchers.

Lakehouse systems are challenging to design for several reasons. First, they need to run over low-cost data lake storage systems, such as Amazon S3 or Azure Data Lake Storage (ADLS), that have relatively high latency compared to a traditional custom-built data warehousing cluster and offer weak transactional guarantees. Second, they aim to support a wide range of workload scales and objectives—from data lake "big data" workloads that involve loading and transforming hundreds of petabytes of data, to interactive data warehouse workloads on smaller tables, where users expect sub-second latency. Third, lakehouse systems aim to be accessible from multiple compute engines through open interfaces, unlike a traditional data warehouse storage system that is co-designed with one compute engine. The protocols that these client engines use to access data need to be designed carefully to support ACID transactions, scale, and high performance, and this is especially challenging when the lakehouse system runs over a high-latency object store with few built-in transactional features.

These challenges lead to several design tradeoffs that we see in the open source lakehouse systems used today. Some of the key design questions include:

How to coordinate transactions? Some systems do all their coordination through the object store when possible (for example, using atomic put-if-absent operations), in order to minimize the number of dependent required to use the lakehouse. For example, with Delta Lake on ADLS, a table remains accessible as long as ADLS is available. In contrast, other systems coordinate through an extenral service such as the Apache Hive MetaStore, which can offer lower latency than an object store, but adds more operational dependencies and risks limiting scalability. The three systems also offered different transaction isolation levels.

Where to store metadata? All of the systems we analyzed aim to store table zone maps (min-max statistics per file) and other metadata in a separate data structure for fast access during query planning, but they use different strategies, including placing the data in the object store as a separate table, placing it in the transaction log, or placing it in a separate service.

How to query metadata? Delta Lake and Hudi can query their metadata storage in a parallel job (e.g., using Spark), speeding up query planning for very large tables but potentially adding latency for smaller tables. On the other hand, Iceberg currently does metadata processing on a single node in its client libraries.

How to efficiently handle updates? Like data lakes and warehouses, lakehouse systems experience a high volume of data loading queries, including both appends and upserts (SQL MERGE). Supporting both fast random updates and fast read queries is challenging on object stores with high latency that perform best with large reads and writes. The different systems optimize for these in different ways, e.g., supporting both "copy-on-write" and "merge-on-read" strategies for when to update existing tables with new data.

We evaluated these tradeoffs for all three systems using Apache Spark on Amazon Elastic MapReduce (running the same engine to avoid vendor-specific engines that optimize for one format, and focus on the differences that stem from the formats themselves and

*These authors contributed equally to this paper.

their client libraries). We created a simple benchmark suite for this purpose, LHBench, that runs TPC-DS as well as microbenchmarks to test metadata management and updates on large tables. We found that there are significant differences in performance and in transactional guarantees across these systems; for TPC-DS performance varies by 1.6× between systems on average and ranges up to 10× for individual queries. LHBench also identifies areas where we believe it is possible to improve on all three formats, including accelerating metadata operations for both small and large tables, and strategies that do better than copy-on-write and merge-on-read for frequently updated tables. We plan to open source LHBench, and hope that this paper and LHBench help the research community study more challenges in real-world lakehouse systems.

2 LAKEHOUSE SYSTEMS

Lakehouses are data management systems based on open formats that run over low-cost storage and provide traditional analytical DBMS features such as ACID transactions, data versioning, auditing, and indexing [20]. As described in the Introduction, these systems are becoming widely used, supplanting raw data lake file formats such as Parquet and ORC at many organizations. For example, over 70% of bytes written across Databricks' 7000 customers are to Delta Lake, and organizations like Netflix and Uber run the majority of their data platforms on lakehouse formats [1, 25].

Because of their ability to mix data lake and warehouse functionality, lakehouse systems are used for a wide range of workloads, often larger than those of lakes or warehouses alone. On the one hand, organizations use lakehouse systems to ingest and organize very large datasets—for example, the largest Delta Lake tables at Databricks customers span well into the hundreds of petabytes, consist of billions of files, and are updated with hundreds of terabytes of ingested data per day [19]. On the other hand, the data warehouse capabilities of lakehouse systems are encouraging organizations to load and manage smaller datasets in them too, in order to combine these with the latest data from their ingest pipelines and build a single management system for all data. For example, most of the CPU-hours on Databricks are used to process tables smaller than 1 TB, and query durations across all tables range from sub-second to hours. New latency-optimized SQL engines for lakehouse formats, such as Photon [22] and Presto [13], are further expanding the range of data sizes and performance regimes that users want.

The main difference between lakehouse systems and conventional analytical RDBMSs is that lakehouse systems aim to provide an *open* interface that allows multiple engines to directly query the same data, so that they can be used efficiently both by SQL workloads and other workloads such as machine learning and graph processing. Lakehouse systems break down the traditional monolithic system design of data warehouses: they provide a storage manager, table metadata, and transaction manager through client libraries that can be embedded in multiple engines, but allow the engines to plan and execute queries.

We next discuss some of the key design questions for lakehouse systems and how Delta Lake, Hudi and Iceberg implement them.

	Table Metadata	Transaction Atomicity	Isolation Levels
Delta Lake	Transaction Log + Metadata Checkpoints	Atomic Log Appends	Serializability, Strict Serializability
Hudi	Transaction Log + Metadata Table	Table-Level Lock	Snapshot Isolation
Iceberg	Hierarchical Files	Table-Level Lock	Snapshot Isolation, Serializability

Table 1: Lakehouse system design features: How they store table metadata, how they provide atomicity for transactions, and what transaction isolation levels they provide.

2.1 Transaction Coordination

One of the most important features of lakehouse systems are transactions. Each lakehouse system we examine claims to offer ACID transaction guarantees for reads and writes. All three systems provide transactions across all partitions of a single table, but not across tables. However, each system implements transactions differently and provides different guarantees, as we show in Table 1.

Delta Lake, Hudi, and Iceberg all implement transactions using multi-version concurrency control [3, 12, 19]. A metadata structure defines which file versions belong to the table. When a transaction begins, it reads this metadata structure to obtain a *snapshot* of the table, then performs all reads from this snapshot. Transactions commit by atomically updating this metadata structure. Delta Lake relies on the underlying storage service to provide atomicity through operations such as put-if-absent (coordinating through DynamoDB for storage layers that do not support suitable operations) [19], while Hudi and Iceberg use table-level locks implemented in ZooKeeper, Hive MetaStore, or DynamoDB [3]. It is worth noting that the most popular lock-based implementations, using Hive MetaStore, do not provide strong guarantees; for example in Iceberg it is possible for a transaction to commit a dirty write if the Hive MetaStore lock times out between a lock heartbeat and the metadata write [9].

To provide isolation between transactions, Delta Lake, Hudi, and Iceberg all use optimistic concurrency control. They validate transactions before committing to check for conflicts with concurrent committed transactions. They provide different isolation levels depending on how they implement validation. By default, Hudi and Iceberg verify that a transaction does not write to any files also written to by committed transactions not in the transaction snapshot [3, 5]. Thus, they provide what Adya [17] terms snapshot isolation: transactions always read data from a snapshot of committed data valid as of the time they started and can only commit if, at commit time, no committed transaction has written data they intend to write. Delta by default (and Iceberg optionally) additionally verifies no transaction read conditions (e.g., in `UPDATE/MERGE/DELETE` operations) could be matched by rows in files committed by transactions not in the snapshot [5, 8]. Thus, they provide serializability: the result of a sequence of transactions is equivalent to that of some serial order of those transactions, but not necessarily the order in the transaction log. Delta can optionally perform this check for read-only operations (e.g., `SELECT` statements), decreasing read-write concurrency but providing what Herlihy and Wing [26] term strict serializability: all transactions (including reads) serialize in the order they appear in the transaction log.

2.2 Metadata Management

For distributed processing engines such as Apache Spark or Presto to plan queries over a table of data stored in a lakehouse format, they need metadata such as the names and sizes of all files in the table. How fast the metadata of these objects can be retrieved puts a limit on how fast queries can complete. For example, S3's LIST only returns 1000 keys per call and therefore can take minutes to retrieve a list of millions of files [19]. Despite their processing scalability, slow metadata processing limits data lakes from providing the same query performance as databases. Thus, efficient metadata management is a crucial aspect of the lakehouse architecture.

To overcome the metadata API rate limits of cloud object stores, lakehouse systems leverage the stores' higher data read rates. All three formats we examine store metadata in files kept alongside the actual data files. Listing the metadata files (much fewer in number than data files) and reading the metadata from them leads to faster query planning times than listing the data files directly from S3. Two metadata organization formats are used: tabular and hierarchical. In the tabular format, adopted by Delta Lake and Hudi, the metadata for a lakehouse table is stored in another, special table: a metadata table in Hudi and a transaction log checkpoint (in a combination of Parquet and JSON formats) in Delta Lake [10, 19]. Transactions do not write to this table directly, but instead write log records that are periodically compacted into the table using a merge-on-read strategy. In the hierarchical format, adopted by Iceberg, metadata is stored in a hierarchy of manifest files [12]. Each file in the bottom level stores metadata for a set of data files, which each file in the upper level contains aggregate metadata for a set of manifest files in the layer below. This is analogous to the tabular format, but with the upper level acting as a table index.

In order to gather the data needed to plan queries, lakehouse systems adopt two different metadata access schemes with different scalability tradeoffs. Queries over Delta Lake and Hudi are typically planned in a distributed fashion, as a batch job must scan the metadata table to find all files involved in a query so they can be used in query planning. By contrast, queries over Iceberg can be planned by a single node which uses the upper level of the manifest hierarchy as an index to minimize the number of reads it must make to the lower level [11]. As we show in Section 3.4, single-node planning improves performance for small queries where the overhead of distributed query planning is high, but may not scale as well as distributed query planning to full scans on large tables with many files. It is an interesting research question whether query planners can be improved to use a cost model to intelligently choose between query planning strategies.

2.3 Data Update Strategies

Lakehouse storage systems adopt two strategies for updating data, with differing tradeoffs between read and write performance:

The **Copy-On-Write (CoW)** strategy identifies the files containing records that need to be updated and eagerly rewrites them to new files with the updated data, thus incurring a high write amplification but no read amplification.

The **Merge-On-Read (MoR)** strategy does not rewrite files. It instead writes out information about record-level changes in additional files and defers the reconciliation until query time, thus producing lower write amplification (i.e., faster writes than CoW) but higher read amplification (i.e., slower reads than CoW). Lower write latency at the cost of read latency is sometime desired in workloads that frequently apply record-level changes (for example, continuously replicating changes from a one database to another).

All three systems support the CoW strategy, as most lakehouse workloads favor high read performance. Iceberg and Hudi currently support the MoR strategy, and Delta is planning to support it as well [7]. Iceberg (and in the future, Delta) MoR implementations use auxiliary "tombstone" files that mark records in Parquet/ORC data files. At query time, these tombstoned records are filtered out. Record updates are implemented by tombstoning the existing record and writing the updated record into Parquet/ORC files. By contrast, the Hudi implementation of MoR stores all the record-level inserts, deletes and updates in row-based Avro files. On query, Hudi reconciles these changes while reading data from the Parquet files. It is worth nothing that Hudi by default deduplicates and sorts the ingested data by keys, thus incurring additional write latency even when using MoR. We examine the performance of the CoW and MoR update strategies in detail in Section 3.3.

3 BENCHMARKING LAKEHOUSE SYSTEMS

In this section, we compare the three lakehouse storage systems, focusing on three main areas: end-to-end performance, performance of different data ingestion strategies, and performance of different metadata access strategies during query planning. We plan to release our benchmark suite, **LHBench**, as an open source project with implementations for all three formats.

3.1 Experimental Setup

We run all experiments using Apache Spark on AWS EMR 6.9.0 storing data in AWS S3 using Delta Lake 2.1.1, Apache Hudi 0.12.0, and Apache Iceberg 1.1.0. This version of EMR is based on Apache Spark 3.3.0. We choose EMR because it lets us compare the three lakehouse formats fairly as it supports all of them but is not specially optimized for any of them. We use the default out-of-the-box configuration for all three lakehouse systems and for EMR, without any manual tuning except increasing the Spark driver JVM memory size to 4 GB to eliminate out-of-memory errors. We perform all experiments with 16 workers on AWS i3.2xlarge instances with 8 vCPUs and 61 GiB of RAM each.

3.2 End-to-end Performance

We first evaluate the effects of different data lakehouse storage formats on load times and query performance using the TPC-DS benchmark suite. We load 3 TB of TPC-DS data into Delta Lake, Hudi, and Iceberg then run all TPC-DS queries three times and report the median runtime, showing results in Figure 1.

Looking at loading times, we find that Delta and Iceberg load in approximately the same time, but Hudi is almost ten times slower than either of them. This is because Hudi is optimized for keyed upserts, not bulk data ingestion, and does an expensive key uniqueness check on each record before inserting it.

Looking at query times, we find that, overall, TPC-DS runs 1.4× faster on Delta Lake than on Hudi and 1.7× faster on Delta Lake than on Iceberg. To investigate the reasons for this performance difference, we examine individual queries. We look specifically at Q90, a query where Delta Lake outperforms Hudi by an especially large margin, as well as Q67, the most expensive query overall (accounting for 8% of total runtime). We find that Spark generates the same query plan for these queries for all lakehouse formats.

The performance difference between the three lakehouse storage systems is explained almost entirely by data reading time; this is unsurprising as the executor and query plans are the same. For example, in Q90, reading the TPC-DS Web Sales table (the largest of the six tables accessed) took 6.5 minutes in Delta Lake across all executors, but 18.8 minutes in Hudi and 18.6 minutes in Iceberg. Delta Lake outperforms Hudi because Hudi targets a smaller file size. For example, a single partition of the Store Sales table is stored in one 128 MB file in Delta Lake but twenty-two 8.3 MB files in Hudi. This reduces the efficiency of columnar compression and increases overhead for the large table scans common in TPC-DS; for example, to read the Web Sales Table in Q90, the executor must read 2128 files (138 GB) in Delta Lake but 18443 files (186 GB) in Hudi. Comparing Delta Lake and Iceberg, we find that both read the same amount of bytes, but that Iceberg uses a custom-built Parquet reader in Spark that is significantly slower than the default Spark reader used by Delta Lake and Hudi. Iceberg's support for column drops and renames required custom functionality not present in Apache Spark's built-in Parquet reader.

While the performance of most queries follows the patterns just described, explaining the overall performance difference, there are some noteworthy exceptions. For extremely small queries such as Q68, the performance bottleneck is often metadata operations used in query planning. As our reported results are the medians of three consecutive runs, these queries are fastest in Hudi because it caches query plans. We examine metadata access strategies in more detail, including from cold starts, in Section 3.4. Additionally, because Iceberg uses the Spark Data Source v2 (DSv2) API instead of the Data Source v1 API (DSv1) used by Delta Lake and Hudi, Spark occasionally generates different query plans over data stored in Iceberg than over data stored in Delta Lake or Hudi. The DSv2 API is less mature and does not report some metrics useful in query planning, so this often results in less performant query plans over Iceberg. For example, in Q9, Spark optimizes a complex aggregation with a cross-join in Delta Lake and Hudi but not in Iceberg, leading to the largest performance difference observed in all of TPC-DS.

3.3 When is Merge-on-Read Faster than Copy-on-Write?

To evaluate the performance of MoR-based formats to CoW-based formats, we evaluate an end-to-end benchmark based on the TPC-DS data refresh maintenance benchmark as well as a synthetic microbenchmark with varying merge source sizes.

3.3.1 TPC-DS Refresh Benchmark. The TPC-DS benchmark specification provides a set of data refresh operations which simulate maintenance of a data warehouse [16, 29]. We evaluate Delta, Hudi, and Iceberg against the 100GB TPC-DS benchmark. CoW was tested in all systems while MoR was tested in Hudi and Iceberg but not Delta which does not have a released MoR implementation. In the benchmark, the 100GB base dataset was loaded followed by the evaluation of a query sample (Q3, Q9, Q34, Q42, and Q59). A total of

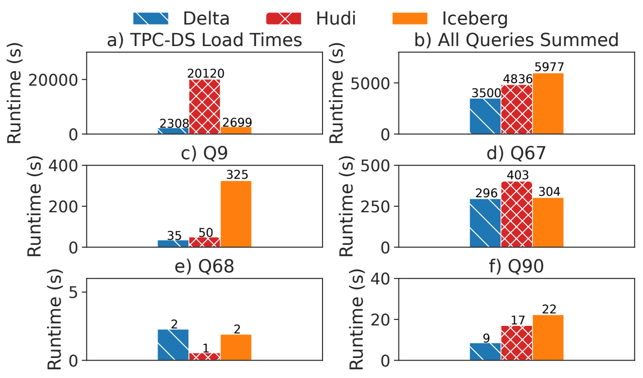

Figure 1: Comparison of Delta Lake, Hudi and Iceberg TPC-DS data ingestion and query times.

10 refreshes (each representing 3% of the original dataset) were performed using the `MERGE INTO` operation. Finally, the query sample was re-evaluated on on tables with all incremental refreshes.

Figure 2 visualizes the resulting latency of each stage of the 100GB TPC-DS data refresh benchmark. We report performance for both Iceberg MoR versions 0.14.0 (on Spark 3.2) and 1.1.0 (on Spark 3.3) due to performance variation.

Both Hudi CoW and MoR have poor write performance during the initial load due to additional pre-processing to rebalance write file sizes and to distribute the data by key. Hudi MoR demonstrates faster merge latencies due to reduced write amplification at the cost of read latency for the additional merge stage. Similarly, merges in Iceberg 0.14.0 in MoR are 1.73× than Iceberg CoW. Iceberg 1.1.0 MoR performance is slower than 0.14.0 due to repeated S3 request retries. We increased the S3 connection pool size for all formats following EMR's documentation [14] but did not tune further parameters as we otherwise use EMR's default settings.

3.3.2 Merge Microbenchmark. Generated TPC-DS refresh data does not have a configurable scale parameter. To better understand the impact of the size of a refresh on merge and query performance, we benchmark Iceberg CoW 1.1.0 and Iceberg MoR 1.1.0 with an isolated microbenchmark. We load a synthetic table with four columns and then apply a single merge from a randomly sampled table with a configurable fraction of the base table size. In addition to the latency of the merge operation, we compare the slowdown for a query after the merge. For each merge scale evaluated, 50% of the rows are inserts while 50% are updates.

Figure 3 shows results for a 100GB benchmark. Iceberg MoR merge latency is consistently lower than that of Iceberg CoW except for the 0.0001% merge configuration (due to fixed overheads in the Iceberg MoR implementation). Iceberg MoR is 1.48× faster at the largest merge configuration (0.1%). We expect Iceberg CoW and MoR merge latency to converge for large merges approaching the size of a full file. For reads, MoR experiences large slowdowns after the merge is performed due to additional read amplification. Iceberg must combine the incremental merge files with the larger columnar files from the initial load. At about 10,000 rows, Iceberg MoR query latency exceeds that of Iceberg CoW.

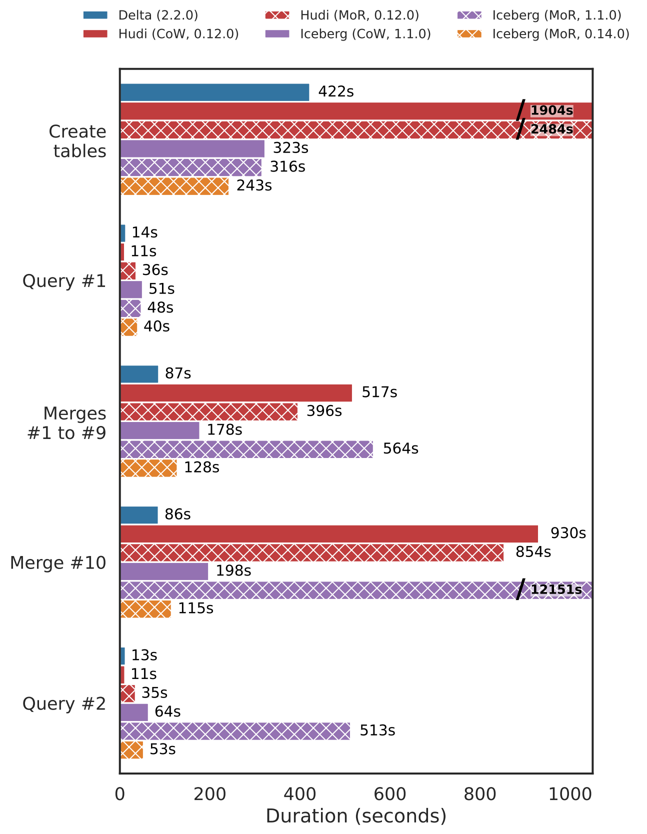

Figure 2: Performance of the 100GB TPC-DS incremental refresh benchmark for queries Q3, Q9, Q34, Q42, and Q59. Hudi performed compaction in merge iteration 10, which is reported separately from iterations 1 to 9.

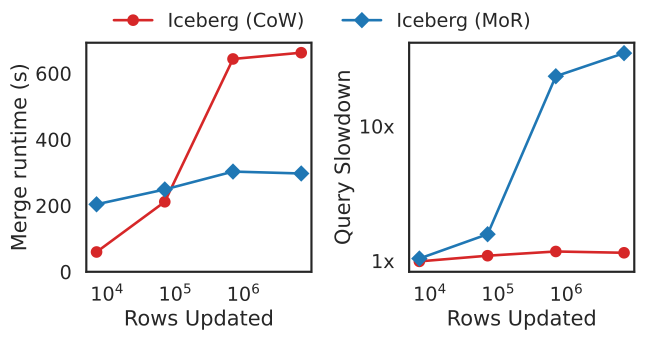

Figure 3: Latency of a merge of varying sizes (left) into a synthetic table with associated slowdown in query latency after the merge (right). The merge rows are 50% inserts and 50% updates.

3.4 When Does Distributed Query Planning Help?

We now examine the performance impact of lakehouse metadata access strategies. We generate TPC-DS data (from the `store_sales` table) and store it in a varying number of 10 MB files (1K to 200K files, or 10 GB to 2 TB of data) in Delta Lake and Iceberg. We choose these two lakehouse storage formats to contrast their different metadata access strategies: Delta Lake distributes query planning, while Iceberg runs it on a single node. To isolate the impact of metadata operations, we use three queries with high selectivity: one accessing a single row, one accessing a single partition, and one accessing only the rows containing a specific value. We measure both the query startup time, defined as the time elapsed between when a query is submitted and when the first job in its query plan begins executing, and the query execution time, defined as the time the query plan takes to execute. All reported measurements are taken from a warm start as the median of three consecutive runs. We show results in Figure 4.

We find that for these queries, metadata access strategies have a significant effect on query performance and are the performance bottleneck for large tables. Moreover, while Iceberg's single-node query planning performed better for smaller tables, Delta Lake's distributed query planning scales better and is dramatically faster for larger tables.

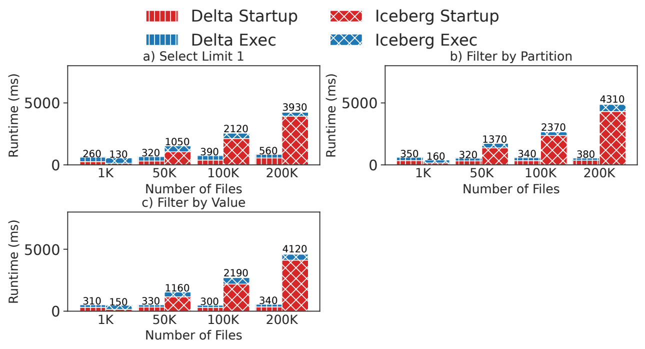

Figure 4: Startup (including query planning) and query execution times of basic table operations on a varying amount of data stored in 10 MB files in Delta Lake and Iceberg.

4 RELATED WORK

To provide low-cost, directly-accessible storage, lakehouse systems build on cloud object stores and data lakes such as HDFS, AWS S3, and Google Cloud Storage. However, these systems provide a minimal interface based on basic primitives such as PUT, GET, and LIST. Lakehouse systems augment data lakes with advanced data management features such as ACID transactions and metadata management for efficient query optimization. There has been much prior work on building ACID transactions on weakly consistent stores. Percolator [30] bolts ACID transactions onto BigTable using multi-version concurrency control, much like lakehouse systems, but otherwise uses BigTable's query executor unmodified. Brantner et al. [23] showed how to build a transactional database on top of S3, although their focus was on supporting highly concurrent small operations (OLTP) not large analytical queries.

Cloud data warehouses, such as AWS Redshift and Snowflake, provide scalable management of structured data, supporting DBMS features such as transactions and query optimization but with a focus on analytics applications. These systems conventionally use disk-based columnar architectures inspired by C-Store [31] and

MonetDB/X100 [27]. They often ingest data into a specialized format that must later be reconciled with the steady-state storage format (for example, by C-Store's tuple mover), similar to the merge-on-read strategy used by some lakehouse systems. To improve scalability, cloud data warehouses are increasingly adopting a disaggregated architecture where data resides in a cloud object store but is locally cached on database servers during query execution [32]. Hence, warehouse and lakehouse architectures are converging, as both rely on low-cost storage. However, unlike lakehouse storage formats, data warehouses do not provide directly accessible storage.

5 CONCLUSION AND OPEN QUESTIONS

The design of lakehouse systems involves important tradeoffs around transaction coordination, metadata storage, and data ingestion strategies that significantly effect performance. Intelligently navigating these tradeoffs is critical to efficiently executing diverse real-world workloads. In this paper, we discussed these tradeoffs in detail and proposed and evaluated an open-source benchmark suite, LHBench, for future researchers to use to study lakehouse systems. We close with several suggestions for future research:

- Can we use a cost model to intelligently choose between query planning strategies, using an indexed search to plan small or highly selective queries on a single node but a distributed metadata scan to plan larger queries?
- Can lakehouse systems efficiently support high write QPS under concurrency? Currently, this is challenging because lakehouse systems must write to the underlying object store to update metadata on every write, and these systems have high latency (often >50 ms), which limits possible write QPS.
- How can lakehouse systems best balance ingest latency and query latency? Can write amplification be pushed off of the critical path and done asynchronously without impacting query latency? Can we automatically find the optimal compaction strategy for a given workload?

ACKNOWLEDGEMENTS

This work was supported by NSF CISE Expeditions Award CCF-1730628, NSF CAREER grant CNS-1651570, and gifts from Amazon, Ant Financial, Astronomer, Cisco, Google, IBM, Intel, Lacework, Microsoft, Meta, Nexla, Samsung SDS, SAP, Uber, and VMWare. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] 2020. Building a Large-scale Transactional Data Lake at Uber Using Apache Hudi. https://www.uber.com/blog/apache-hudi-graduation/
[2] 2022. Apache Hudi. https://hudi.apache.org
[3] 2022. Apache Hudi Concurrency Control. https://hudi.apache.org/docs/next/concurrency_control/
[4] 2022. Apache Iceberg. https://iceberg.apache.org
[5] 2022. Apache Iceberg Isolation Levels. https://iceberg.apache.org/javadoc/0.11.0/org/apache/iceberg/IsolationLevel.html
[6] 2022. AWS EMR Hudi. https://docs.aws.3.com/emr/latest/ReleaseGuide/emr-hudi.html
[7] 2022. Deletion Vectors to speed up DML operations. https://github.com/delta-io/delta/issues/1367
[8] 2022. Delta Lake Concurency Control. https://docs.databricks.com/delta/concurrency-control.html
[9] 2022. Hive: Fix concurrent transactions overwriting commits by adding hive lock heartbeats. https://github.com/apache/iceberg/pull/5036
[10] 2022. Hudi Metadata Table. https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=147427331
[11] 2022. Iceberg Query Planning Performance. https://iceberg.apache.org/docs/latest/performance/
[12] 2022. Iceberg Table Spec. https://iceberg.apache.org/spec/
[13] 2022. Presto. https://prestodb.io/
[14] 2022. Resolve "Timeout waiting for connection from pool" error in Amazon EMR. https://aws.amazon.com/premiumsupport/knowledge-center/emr-timeout-connection-wait/
[15] 2022. Synapse Delta Lake. https://docs.microsoft.com/en-us/azure/synapse-analytics/spark/apache-spark-what-is-delta-lake
[16] 2022. TPC-DS Specification Version 2.1.0. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.1.0.pdf
[17] Atul Adya. 1999. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions.* Ph. D. Dissertation. MIT.
[18] Azim Afroozeh. 2020. Towards a New File Format for Big Data: SIMD-Friendly Composable Compression. https://homepages.cwi.nl/~boncz/msc/2020-AzimAfroozeh.pdf
[19] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3411–3424.
[20] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR.*
[21] Edmon Begoli, Ian Goethert, and Kathryn Knight. 2021. A Lakehouse Architecture for the Management and Analysis of Heterogeneous Data for Biomedical Research and Mega-biobanks. In *2021 IEEE Big Data.* 4643–4651.
[22] Alexander et al. Behm. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *SIGMOD 2022.* 2326–2339.
[23] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. 2008. Building a Database on S3. In *SIGMOD 2008.* 251–264.
[24] Bogdan Vladimir Ghita, Diego G. Tomé, and Peter A. Boncz. 2020. White-box Compression: Learning and Exploiting Compact Table Representations. In *CIDR.*
[25] Ted Gooch. 2020. Why and How Netflix Created and Migrated to a New Table Format: Iceberg. https://www.dremio.com/subsurface/why-and-how-netflix-created-and-migrated-to-a-new-table-format-iceberg/
[26] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (jul 1990), 463–492. https://doi.org/10.1145/78969.78972
[27] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. 2012. Monetdb: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin* 35, 1 (2012), 40–45.
[28] Samuel Madden, Jialin Ding, Tim Kraska, Sivaprasad Sudhir, David Cohen, Timothy Mattson, and Nesmie Tatbul. 2022. Self-Organizing Data Containers. In *CIDR.*
[29] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS.. In *VLDB*, Vol. 6. Citeseer, 1049–1058.
[30] Daniel Peng and Frank Dabek. 2010. Large-scale incremental processing using distributed transactions and notifications. In *OSDI 2010.*
[31] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-Store: A Column-Oriented DBMS. In *VLDB 2005.* 553–564.
[32] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *NSDI 2020.* 449–462.