Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores

Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał Świtakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz[1], Ali Ghodsi[2], Sameer Paranjpye, Pieter Senster, Reynold Xin, Matei Zaharia[3]
Databricks, [1]CWI, [2]UC Berkeley, [3]Stanford University
delta-paper-authors@databricks.com

ABSTRACT

Cloud object stores such as Amazon S3 are some of the largest and most cost-effective storage systems on the planet, making them an attractive target to store large data warehouses and data lakes. Unfortunately, their implementation as key-value stores makes it difficult to achieve ACID transactions and high performance: metadata operations such as listing objects are expensive, and consistency guarantees are limited. In this paper, we present Delta Lake, an open source ACID table storage layer over cloud object stores initially developed at Databricks. Delta Lake uses a transaction log that is compacted into Apache Parquet format to provide ACID properties, time travel, and significantly faster metadata operations for large tabular datasets (e.g., the ability to quickly search billions of table partitions for those relevant to a query). It also leverages this design to provide high-level features such as automatic data layout optimization, upserts, caching, and audit logs. Delta Lake tables can be accessed from Apache Spark, Hive, Presto, Redshift and other systems. Delta Lake is deployed at thousands of Databricks customers that process exabytes of data per day, with the largest instances managing exabyte-scale datasets and billions of objects.

PVLDB Reference Format:
Armbrust et al. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *PVLDB*, 13(12): 3411-3424, 2020.
DOI: https://doi.org/10.14778/3415478.3415560

1. INTRODUCTION

Cloud object stores such as Amazon S3 [4] and Azure Blob Storage [17] have become some of the largest and most widely used storage systems on the planet, holding exabytes of data for millions of customers [46]. Apart from the traditional advantages of clouds services, such as pay-as-you-go billing, economies of scale, and expert management [15], cloud object stores are especially attractive because they allow users to scale computing and storage resources separately: for example, a user can store a petabyte of data but only run a cluster to execute a query over it for a few hours.

As a result, many organizations now use cloud object stores to manage large structured datasets in data warehouses and data lakes. The major open source "big data" systems, including Apache Spark, Hive and Presto [45, 52, 42], support reading and writing to cloud object stores using file formats such as Apache Parquet and ORC [13, 12]. Commercial services including AWS Athena, Google BigQuery and Redshift Spectrum [1, 29, 39] can also query directly against these systems and these open file formats.

Unfortunately, although many systems support reading and writing to cloud object stores, achieving *performant* and *mutable* table storage over these systems is challenging, making it difficult to implement data warehousing capabilities over them. Unlike distributed filesystems such as HDFS [5], or custom storage engines in a DBMS, most cloud object stores are merely key-value stores, with no cross-key consistency guarantees. Their performance characteristics also differ greatly from distributed filesystems and require special care.

The most common way to store relational datasets in cloud object stores is using columnar file formats such as Parquet and ORC, where each table is stored as a set of objects (Parquet or ORC "files"), possibly clustered into "partitions" by some fields (e.g., a separate set of objects for each date) [45]. This approach can offer acceptable performance for scan workloads as long as the object files are moderately large. However, it creates both *correctness* and *performance* challenges for more complex workloads. First, because multi-object updates are not atomic, there is no isolation between queries: for example, if a query needs to update multiple objects in the table (e.g., remove the records about one user across all the table's Parquet files), readers will see partial updates as the query updates each object individually. Rolling back writes is also difficult: if an update query crashes, the table is in a corrupted state. Second, for large tables with millions of objects, metadata operations are expensive. For example, Parquet files include footers with min/max statistics that can be used to skip reading them in selective queries. Reading such a footer on HDFS might take a few milliseconds, but the latency of cloud object stores is so much higher that these data skipping checks can take longer than the actual query.

In our experience working with cloud customers, these consistency and performance issues create major challenges for enterprise data teams. Most enterprise datasets are continuously updated, so they require a solution for atomic writes; most datasets about users require table-wide updates to implement privacy policies such as GDPR compliance [27]; and even purely internal datasets may require updates to repair incorrect data, incorporate late records, etc. Anecdotally, in the first few years of Databricks' cloud service (2014–2016), around half the support escalations we received were due to data corruption, consistency or performance issues due to cloud storage strategies (e.g., undoing the effect of a crashed update job, or improving the performance of a query that reads tens of thousands of objects).

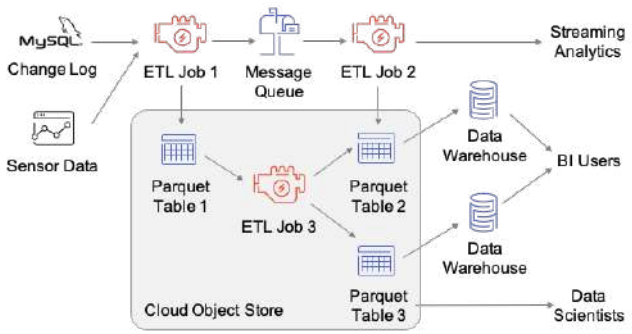
Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415560

To address these challenges, we designed Delta Lake, an ACID table storage layer over cloud object stores that we started providing to customers in 2017 and open sourced in 2019 [26]. The core idea of Delta Lake is simple: we maintain information about *which objects* are part of a Delta table in an ACID manner, using a write-ahead log that is *itself* stored in the cloud object store. The objects themselves are encoded in Parquet, making it easy to write connectors from engines that can already process Parquet. This design allows clients to update multiple objects at once, replace a subset of the objects with another, etc., in a serializable manner while still achieving high parallel read and write performance from the objects themselves (similar to raw Parquet). The log also contains metadata such as min/max statistics for each data file, enabling order of magnitude faster metadata searches than the "files in object store" approach. Crucially, we designed Delta Lake so that all the metadata is in the underlying object store, and transactions are achieved using optimistic concurrency protocols against the object store (with some details varying by cloud provider). This means that no servers need to be running to maintain state for a Delta table; users only need to launch servers when running queries, and enjoy the benefits of separately scaling compute and storage.

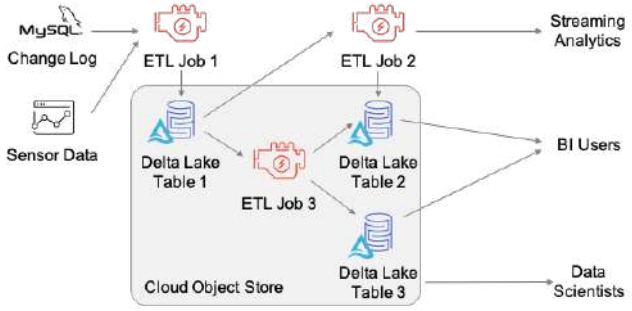
Based on this transactional design, we were also able add multiple other features in Delta Lake that are not available in traditional cloud data lakes to address common customer pain points, including:

- **Time travel** to let users query point-in-time snapshots or roll back erroneous updates to their data.
- **UPSERT, DELETE and MERGE operations,** which efficiently rewrite the relevant objects to implement updates to archived data and compliance workflows (e.g., for GDPR [27]).
- **Efficient streaming I/O,** by letting streaming jobs write small objects into the table at low latency, then transactionally coalescing them into larger objects later for performance. Fast "tailing" reads of the new data added to a table are also supported, so that jobs can treat a Delta table as a message bus.
- **Caching:** Because the objects in a Delta table and its log are immutable, cluster nodes can safely cache them on local storage. We leverage this in the Databricks cloud service to implement a transparent SSD cache for Delta tables.
- **Data layout optimization:** Our cloud service includes a feature that automatically optimizes the size of objects in a table and the clustering of data records (e.g., storing records in Z-order to achieve locality along multiple dimensions) without impacting running queries.
- **Schema evolution,** allowing Delta to continue reading old Parquet files without rewriting them if a table's schema changes.
- **Audit logging** based on the transaction log.

Together, these feature improve both the manageability and performance of working with data in cloud object stores, and enable a *"lakehouse"* paradigm that combines the key features of data warehouses and data lakes: standard DBMS management functions usable directly against low-cost object stores. In fact, we found that many Databricks customers could simplify their overall data architectures with Delta Lake, by replacing previously separate data lake, data warehouse and streaming storage systems with Delta tables that provide appropriate features for all these use cases. Figure 1 shows an extreme example, where a data pipeline that includes object storage, a message queue and two data warehouses for different business intelligence teams (each running their own computing resources)



(a) Pipeline using separate storage systems.



(b) Using Delta Lake for both stream and table storage.

Figure 1: A data pipeline implemented using three storage systems (a message queue, object store and data warehouse), or using Delta Lake for both stream and table storage. The Delta Lake version removes the need to manage multiple copies of the data and uses only low-cost object storage.

is replaced with just Delta tables on object storage, using Delta's streaming I/O and performance features to run ETL and BI. The new pipeline uses only low-cost object storage and creates fewer copies of the data, reducing both storage cost and maintenance overheads.

Delta Lake is now used by most of Databricks' large customers, where it processes exabytes of data per day (around half our overall workload). It is also supported by Google Cloud, Alibaba, Tencent, Fivetran, Informatica, Qlik, Talend, and other products [50, 26, 33]. Among Databricks customers, Delta Lake's use cases are highly diverse, ranging from traditional ETL and data warehousing workloads to bioinformatics, real time network security analysis (on hundreds of TB of streaming event data per day), GDPR compliance, and data management for machine learning (managing millions of images as records in a Delta table rather than S3 objects to get ACID and improved performance). We detail these use cases in Section 5.

Anecdotally, Delta Lake reduced the fraction of support issues about cloud storage at Databricks from a half to nearly none. It also improved workload performance for most customers, with speedups as high as 100× in extreme cases where its data layout optimizations and fast access to statistics are used to query very high-dimensional datasets (e.g., the network security and bioinformatics use cases). The open source Delta Lake project [26] includes connectors to Apache Spark (batch or streaming), Hive, Presto, AWS Athena, Redshift and Snowflake, and can run over multiple cloud object stores or over HDFS. In the rest of this paper, we present the motivation and design of Delta Lake, along with customer use cases and performance experiments that motivated our design.

2. MOTIVATION: CHARACTERISTICS AND CHALLENGES OF OBJECT STORES

In this section, we describe the API and performance characteristics of cloud object stores to explain why efficient table storage on these systems can be challenging, and sketch existing approaches to manage tabular datasets on them.

2.1 Object Store APIs

Cloud object stores, such as Amazon S3 [4] and Azure Blob Storage [17], Google Cloud Storage [30], and OpenStack Swift [38], offer a simple but easy-to-scale *key-value store* interface. These systems allow users to create *buckets* that each store multiple *objects*, each of which is a binary blob ranging in size up to a few TB (for example, on S3, the limit on object sizes is 5 TB [4]). Each object is identified by a string *key*. It is common to model keys after file system paths (e.g., `warehouse/table1/part1.parquet`), but unlike file systems, cloud object stores do *not* provide cheap renames of objects or of "directories". Cloud object stores also provide metadata APIs, such as S3's LIST operation [41], that can generally list the available objects in a bucket by lexicographic order of key, given a start key. This makes it possible to efficiently list the objects in a "directory" if using file-system-style paths, by starting a LIST request at the key that represents that directory prefix (e.g., `warehouse/table1/`). Unfortunately, these metadata APIs are generally expensive: for example, S3's LIST only returns up to 1000 keys per call, and each call takes tens to hundreds of milliseconds, so it can take minutes to list a dataset with millions of objects using a sequential implementation.

When reading an object, cloud object stores usually support byte-range requests, so it is efficient to read just a range within a large object (e.g., bytes 10,000 to 20,000). This makes it possible to leverage storage formats that cluster commonly accessed values.

Updating objects usually requires rewriting the whole object at once. These updates can be made atomic, so that readers will either see the new object version or the old one. Some systems also support appends to an object [48].

Some cloud vendors have also implemented distributed filesystem interfaces over blob storage, such as Azure's ADLS Gen2 [18], which over similar semantics to Hadoop's HDFS (e.g., directories and atomic renames). Nonetheless, many of the problems that Delta Lake tackles, such as small files [36] and atomic updates across multiple directories, are still present even when using a distributed filesystem—indeed, multiple users run Delta Lake over HDFS.

2.2 Consistency Properties

The most popular cloud object stores provide eventual consistency for each key and no consistency guarantees across keys, which creates challenges when managing a dataset that consists of multiple objects, as described in the Introduction. In particular, after a client uploads a new object, other clients are not necessarily guaranteed to see the object in LIST or read operations right away. Likewise, updates to an existing object may not immediately be visible to other clients. Moreover, depending on the object store, even the client doing a write may not immediately see the new objects.

The exact consistency model differs by cloud provider, and can be fairly complex. As a concrete example, Amazon S3 provides read-after-write consistency for clients that write a new object, meaning that read operations such as S3's GET will return the object contents after a PUT. However, there is one exception: if the client writing the object issued a GET to the (nonexistent) key before its PUT, then subsequent GETs might *not* read the object for a period of time, most likely because S3 employs negative caching. Moreover, S3's LIST operations are always eventually consistent, meaning that a LIST after a PUT might not return the new object [40]. Other cloud object stores offer stronger guarantees [31], but still lack atomic operations across multiple keys.

2.3 Performance Characteristics

In our experience, achieving high throughput with object stores requires a careful balance of *large sequential I/Os* and *parallelism*.

For reads, the most granular operation available is reading a sequential byte range, as described earlier. Each read operation usually incurs at least 5–10 ms of base latency, and can then read data at roughly 50–100 MB/s, so an operation needs to read at least several hundred kilobytes to achieve at least half the peak throughput for sequential reads, and multiple megabytes to approach the peak throughput. Moreover, on typical VM configurations, applications need to run multiple reads in parallel to maximize throughput. For example, the VM types most frequently used for analytics on AWS have at least 10 Gbps network bandwidth, so they need to run 8–10 reads in parallel to fully utilize this bandwidth.

LIST operations also require significant parallelism to quickly list large sets of objects. For example, S3's LIST operations can only return up to 1000 objects per requests, and take tens to hundreds of milliseconds, so clients need to issue hundreds of LISTs in parallel to list large buckets or "directories". In our optimized runtime for Apache Spark in the cloud, we sometimes parallelize LIST operations over the *worker nodes* in the Spark cluster in addition to threads in the driver node to have them run faster. In Delta Lake, the metadata about available objects (including their names and data statistics) is stored in the Delta log instead, but we also parallelize reads from this log over the cluster.

Write operations generally have to replace a whole object (or append to it), as discussed in Section 2.1. This implies that if a table is expected to receive point updates, then the objects in it should be kept small, which is at odds with supporting large reads. Alternatively, one can use a log-structured storage format.

Implications for Table Storage. The performance characteristics of object stores lead to three considerations for analytical workloads:

1. Keep frequently accessed data close-by sequentially, which generally leads to choosing columnar formats.

2. Make objects large, but not too large. Large objects increase the cost of updating data (e.g., deleting all data about one user) because they must be fully rewritten.

3. Avoid LIST operations, and make these operations request lexicographic key ranges when possible.

2.4 Existing Approaches for Table Storage

Based on the characteristics of object stores, three major approces are used to manage tabular datasets on them today. We briefly sketch these approaches and their challenges.

1. Directories of Files. The most common approach, supported by the open source big data stack as well as many cloud services, is to store the table as a collection of objects, typically in a columnar format such as Parquet. As a refinement, the records may be "partitioned" into directories based on one or more attributes. For example, for a table with a `date` field, we might create a separate directory of objects for each date, e.g., `mytable/date=2020-01-01/obj1` and `mytable/date=2020-01-01/obj2` for data from Jan 1st, then `mytable/date=2020-01-02/obj1` for Jan 2nd, etc, and split incoming data into multiple objects based on this field. Such partitioning reduces the cost of LIST operations and reads for queries that only access a few partitions.

This approach is attractive because the table is "just a bunch of objects" that can be accessed from many tools without running any additional data stores or systems. It originated in Apache Hive on HDFS [45] and matches working with Parquet, Hive and other big data software on filesystems.

Challenges with this Approach. As described in the Introduction, the "just a bunch of files" approach suffers from both performance and consistency problems on cloud object stores. The most common challenges customers encountered are:

- No atomicity across multiple objects: Any transaction that needs to write or update multiple objects risks having partial writes visible to other clients. Moreover, if such a transaction fails, data is left in a corrupt state.
- Eventual consistency: Even with successful transactions, clients may see some of the updated objects but not others.
- Poor performance: Listing objects to find the ones relevant for a query is expensive, even if they are partitioned into directories by a key. Moreover, accessing per-object statistics stored in Parquet or ORC files is expensive because it requires additional high-latency reads for each feature.
- No management functionality: The object store does not implement standard utilities such as table versioning or audit logs that are familiar from data warehouses.

2. Custom Storage Engines. "Closed-world" storage engines built for the cloud, such as the Snowflake data warehouse [23], can bypass many of the consistency challenges with cloud object stores by managing metadata themselves in a separate, strongly consistent service, which holds the "source of truth" about what objects comprise a table. In these engines, the cloud object store can be treated as a dumb block device and standard techniques can be used to implement efficient metadata storage, search, updates, etc. over the cloud objects. However, this approach requires running a highly available service to manage the metadata, which can be expensive, can add overhead when querying the data with an external computing engine, and can lock users into one provider.

Challenges with this Approach. Despite the benefits of a clean-slate "closed-world" design, some specific challenges we encountered with this approach are:

- All I/O operations to a table need contact the metadata service, which can increase its resource cost and reduce performance and availability. For example, when accessing a Snowflake dataset in Spark, the reads from Snowflake's Spark connector stream data through Snowflake's services, reducing performance compared to direct reads from cloud object stores.
- Connectors to existing computing engines require more engineering work to implement than an approach that reuses existing open formats such as Parquet. In our experience, data teams wish to use a wide range of computing engines on their data (e.g. Spark, TensorFlow, PyTorch and others), so making connectors easy to implement is important.
- The proprietary metadata service ties users to a specific service provider, whereas an approach based on directly accessing objects in cloud storage enables users to always access their data using different technologies.

Apache Hive ACID [32] implements a similar approach over HDFS or object stores by using the Hive Metastore (a transactional RDBMS such as MySQL) to keep track of multiple files that hold updates for a table stored in ORC format. However, this approach is limited by the performance of the metastore, which can become a bottleneck for tables with millions of objects in our experience.

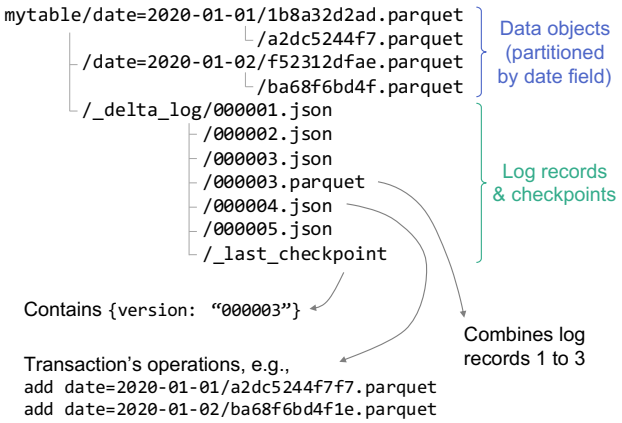


Figure 2: Objects stored in a sample Delta table.

3. Metadata in Object Stores. Delta Lake's approach is to store a transaction log and metadata directly within the cloud object store, and use a set of protocols over object store operations to achieve serializability. The data within a table is then stored in Parquet format, making it easy to access from any software that already supports Parquet as long as a minimal connector is available to discover the set of objects to read.[1] Although we believe that Delta Lake was the first system to use this design (starting in 2016), two other software packages also support it now — Apache Hudi [8] and Apache Iceberg [10]. Delta Lake offers a number of unique features not supported by these systems, such as Z-order clustering, caching, and background optimization. We discuss the similarities and differences between these systems in more detail in Section 8.

3. DELTA LAKE STORAGE FORMAT AND ACCESS PROTOCOLS

A Delta Lake table is a directory on a cloud object store or file system that holds *data objects* with the table contents and a *log* of transaction operations (with occasional checkpoints). Clients update these data structures using optimistic concurrency control protocols that we tailored for the characteristics of cloud object stores. In this section, we describe Delta Lake's storage format and these access protocols. We also describe Delta Lake's transaction isolation levels, which include serializable and snapshot isolation within a table.

3.1 Storage Format

Figure 2 shows the storage format for a Delta table. Each table is stored within a file system directory (`mytable` here) or as objects starting with the same "directory" key prefix in an object store.

3.1.1 Data Objects

The table contents are stored in Apache Parquet objects, possibly organized into directories using Hive's partition naming convention.

[1] As we discuss in Section 4.8, most Hadoop ecosystem projects already supported a simple way to read only a subset of files in a directory called "manifest files," which were first added to support symbolic links in Hive. Delta Lake can maintain a manifest file for each table to enable consistent reads from these systems.

For example, in Figure 2, the table is partitioned by the `date` column, so the data objects are in separate directories for each date. We chose Parquet as our underlying data format because it was column-oriented, offered diverse compression updates, supported nested data types for semi-structured data, and already had performant implementations in many engines. Building on an existing, open file format also ensured that Delta Lake can continue to take advantage of newly released updates to Parquet libraries and simplified developing connectors to other engines (Section 4.8). Other open source formats, such as ORC [12], would likely have worked similarly, but Parquet had the most mature support in Spark.

Each data object in Delta has a unique name, typically chosen by the writer by generating a GUID. However, *which* objects are part of each version of the table is determined by the transaction log.

3.1.2 Log

The log is stored in the `_delta_log` subdirectory within the table. It contains a sequence of JSON objects with increasing, zero-padded numerical IDs to store the log records, together with occasional *checkpoints* for specific log objects that summarize the log up to that point in Parquet format.[2] As we discuss in Section 3.2, some simple access protocols (depending on the atomic operations available in each object store) are used to create new log entries or checkpoints and have clients agree on an order of transactions.

Each log record object (e.g., `000003.json`) contains an array of *actions* to apply to the previous version of the table in order to generate the next one. The available actions are:

Change Metadata. The `metaData` action changes the current metadata of the table. The first version of a table must contain a `metaData` action. Subsequent `metaData` actions completely overwrite the current metadata of the table. The metadata is a data structure containing the schema, partition column names (i.e., `date` in our example) if the column is partitioned, the storage format of data files (typically Parquet, but this provides extensibility), and other configuration options, such as marking a table as append-only.

Add or Remove Files. The `add` and `remove` actions are used to modify the data in a table by adding or removing individual data objects respectively. Clients can thus search the log to find all `added` objects that have not been `removed` to determine the set of objects that make up the table.

The `add` record for a data object can also include data statistics, such as the total record count and per-column min/max values and null counts. When an `add` action is encountered for a path that is already present in the table, statistics from the latest version replace that from any previous version. This can be used to "upgrade" old tables with more types of statistics in new versions of Delta Lake.

The `remove` action includes a timestamp that indicates when the removal occurred. Physical deletion of the data object can happen lazily after a user-specified retention time threshold. This delay allows concurrent readers to continue to execute against stale snapshots of the data. A `remove` action should remain in the log and any log checkpoints as a tombstone until the underlying data object has been deleted.

The `dataChange` flag on either add or remove actions can be set to `false` to indicate that this action, when combined with other actions in the same log record object, only rearranges existing data or adds statistics. For example, streaming queries that are tailing the transaction log can use this flag to skip actions that would not affect their results, such as changing the sort order in earlier data files.

[2]Zero-padding the IDs of log records makes it efficient for clients to find all the new records after a checkpoint using the lexicographic LIST operations available on object stores.

Protocol Evolution. The `protocol` action is used to increase the version of the Delta protocol that is required to read or write a given table. We use this action to add new features to the format while indicating which clients are still compatible.

Add Provenance Information. Each log record object can also include provenance information in a `commitInfo` action, e.g., to log which user did the operation.

Update Application Transaction IDs. Delta Lake also provides a means for application to include their own data inside log records, which can be useful for implementing end-to-end transactional applications. For example, stream processing systems that write to a Delta table need to know which of their writes have previously been committed in order to achieve "exactly-once" semantics: if the streaming job crashes, it needs to know which of its writes have previously made it into the table, so that it can replay subsequent writes starting at the correct offset in its input streams. To support this use case, Delta Lake allows applications to write a custom `txn` action with `appId` and `version` fields in their log record objects that can track application-specific information, such as the corresponding offset in the input stream in our example. By placing this information in the same log record as the corresponding Delta `add` and `remove` operations, which is inserted into the log atomically, the application can ensure that Delta Lake adds the new data and stores its version field atomically. Each application can simply generate its `appId` randomly to receive a unique ID. We use this facility in the Delta Lake connector for Spark Structured Streaming [14].

3.1.3 Log Checkpoints

For performance, it is necessary to compress the log periodically into checkpoints. Checkpoints store all the *non-redundant* actions in the table's log up to a certain log record ID, in Parquet format. Some sets of actions are redundant and can be removed. These include:

- `add` actions followed by a `remove` action for the same data object. The `add`s can be removed because the data object is no longer part of the table. The `remove` actions should be kept as a tombstone according to the table's data retention configuration. Specifically, clients use the timestamp in `remove` actions to decide when to delete an object from storage.
- Multiple `add`s for the same object can be replaced by the last one, because new ones can only add statistics.
- Multiple `txn` actions from the same `appId` can be replaced by the latest one, which contains its latest `version` field.
- The `changeMetadata` and `protocol` actions can also be coalesced to keep only the latest metadata.

The end result of the checkpointing process is therefore a Parquet file that contains an `add` record for each object still in the table, `remove` records for objects that were deleted but need to be retained until the retention period has expired, and a small number of other records such as `txn`, `protocol` and `changeMetadata`. This column-oriented file is in an ideal format for querying metadata about the table, and for finding which objects may contain data relevant for a selective query based on their data statistics. In our experience, finding the set of objects to read for a query is nearly always faster using a Delta Lake checkpoint than using LIST operations and reading Parquet file footers on an object store.

Any client may attempt to create a checkpoint up to a given log record ID, and should write it as a `.parquet` file for the corresponding ID if successful. For example, `000003.parquet` would represent a checkpoint of the records up to and including `000003.json`. By default, our clients write checkpoints every 10 transactions.

Lastly, clients accessing the Delta Lake table need to efficiently find the last checkpoint (and the tail of the log) without LISTing all the objects in the `_delta_log` directory. Checkpoint writers write their new checkpoint ID in the `_delta_log/_last_checkpoint` file if it is newer than the current ID in that file. Note that it is fine for the `_last_checkpoint` file to be out of date due to eventual consistency issues with the cloud object store, because clients will still search for new checkpoints after the ID in this file.

3.2 Access Protocols

Delta Lake's access protocols are designed to let clients achieve serializable transactions using only operations on the object store, despite object stores' eventual consistency guarantees. The key choice that makes this possible is that a log record object, such as `000003.json`, is the "root" data structure that a client needs to know to read a specific version of the table. Given this object's content, the client can then query for other objects from the object store, possibly waiting if they are not yet visible due to eventual consistency delays, and read the table data. For transactions that perform writes, clients need a way to ensure that only a single writer can create the next log record (e.g., `000003.json`), and can then use this to implement optimistic concurrency control.

3.2.1 Reading from Tables

We first describe how to run read-only transactions against a Delta table. These transactions will safely read some version of the table. Read-only transactions have five steps:

1. Read the `_last_checkpoint` object in the table's log directory, if it exists, to obtain a recent checkpoint ID.
2. Use a LIST operation whose start key is the last checkpoint ID if present, or 0 otherwise, to find any newer `.json` and `.parquet` files in the table's log directory. This provides a list files that can be used to reconstruct the table's state starting from a recent checkpoint. (Note that, due to eventual consistency of the cloud object store, this LIST operation may return a non-contiguous set of objects, such has `000004.json` and `000006.json` but not `000005.json`. Nonetheless, the client can use the largest ID returned as a target table version to read from, and wait for missing objects to become visible.)
3. Use the checkpoint (if present) and subsequent log records identified in the previous step to reconstruct the state of the table—namely, the set of data objects that have `add` records but no corresponding `remove` records, and their associated data statistics. Our format is designed so that this task can run in parallel: for example, in our Spark connector, we read the checkpoint Parquet file and log objects using Spark jobs.
4. Use the statistics to identify the set of data object files relevant for the read query.
5. Query the object store to read the relevant data objects, possibly in parallel across a cluster. Note that due to eventual consistency of the cloud object stores, some worker nodes may not be able to query objects that the query planner found in the log; these can simply retry after a short amount of time.

We note that this protocol is designed to tolerate eventual consistency at each step. For example, if a client reads a stale version of the `_last_checkpoint` file, it can still discover newer log files in the subsequent LIST operation and reconstruct a recent snapshot of the table. The `_last_checkpoint` file only helps to reduce the cost of the LIST operation by providing a recent checkpoint ID. Likewise, the client can tolerate inconsistency in listing the recent records (e.g., gaps in the log record IDs) or in reading the data objects referenced in the log that may not yet be visible to it in the object store.

3.2.2 Write Transactions

Transactions that write data generally proceed in up to five steps, depending on the operations in the transaction:

1. Identify a recent log record ID, say r, using steps 1–2 of the read protocol (i.e., looking forward from the last checkpoint ID). The transaction will then read the data at table version r (if needed) and attempt to write log record $r + 1$.
2. Read data at table version r, if required, using the same steps as the read protocol (i.e. combining the previous checkpoint and any further log records, then reading the data objects referenced in those).
3. Write any new data objects that the transaction aims to add to the table into new files in the correct data directories, generating the object names using GUIDs. This step can happen in parallel. At the end, these objects are ready to reference in a new log record.
4. Attempt to write the transaction's log record into the $r + 1$ `.json` log object, if no other client has written this object. This step needs to be atomic, and we discuss how to achieve that in various object stores shortly. If the step fails, the transaction can be retried; depending on the query's semantics, the client can also reuse the new data objects it wrote in step 3 and simply try to add them to the table in a new log record.
5. Optionally, write a new `.parquet` checkpoint for log record $r + 1$. (In practice, our implementations do this every 10 records by default.) Then, after this write is complete, update the `_last_checkpoint` file to point to checkpoint $r + 1$.

Note that the fifth step, of writing a checkpoint and then updating the `_last_checkpoint` object, only affects performance, and a client failure anywhere during this step will not corrupt the data. For example, if a client fails to write a checkpoint, or writes a checkpoint Parquet object but does not update `_last_checkpoint`, then other clients can still read the table using earlier checkpoints. The transaction commits atomically if step 4 is successful.

Adding Log Records Atomically. As is apparent in the write protocol, step 4, i.e., creating the $r + 1$ `.json` log record object, needs to be atomic: only one client should succeed in creating the object with that name. Unfortunately, not all large-scale storage systems have an atomic put-if-absent operation, but we were able to implement this step in different ways for different storage systems:

- Google Cloud Storage and Azure Blob Store support atomic put-if-absent operations, so we use those.
- On distributed filesystems such as HDFS, we use atomic renames to rename a temporary file to the target name (e.g., `000004.json`) or fail if it already exists. Azure Data Lake Storage [18] also offers a filesystem API with atomic renames, so we use the same approach there.
- Amazon S3 does not have atomic "put if absent" or rename operations. In Databricks service deployments, we use a separate lightweight coordination service to ensure that only one client can add a record with each log ID. This service is only needed for log writes (not reads and not data operations), so

its load is low. In our open source Delta Lake connector for Apache Spark, we ensure that writes going through the same Spark driver program (`SparkContext` object) get different log record IDs using in-memory state, which means that users can still make concurrent operations on a Delta table in a single Spark cluster. We also provide an API to plug in a custom `LogStore` class that can use other coordination mechanisms if the user wants to run a separate, strongly consistent store.

3.3 Available Isolation Levels

Given Delta Lake's concurrency control protocols, all transactions that perform writes are serializable, leading to a serial schedule in increasing order of log record IDs. This follows from the commit protocol for write transactions, where only one transaction can write the record with each record ID. Read transactions can achieve either snapshot isolation or serializability. The read protocol we described in Section 3.2.1 only reads a snapshot of the table, so clients that leverage this protocol will achieve snapshot isolation, but a client that wishes to run a serializable read (perhaps between other serializable transactions) could execute a read-write transaction that performs a dummy write to achieve this. In practice, Delta Lake connector implementations also cache the latest log record IDs they have accessed for each table in memory, so clients will "read their own writes" even if they use snapshot isolation for reads, and read a monotonic sequence of table versions when doing multiple reads.

Importantly, Delta Lake currently only supports transactions *within one table*. The object store log design could also be extended to manage multiple tables in the same log in the future.

3.4 Transaction Rates

Delta Lake's write transaction rate is limited by the latency of the put-if-absent operations to write new log records, described in Section 3.2.2. As in any optimistic concurrency control protocol, a high rate of write transactions will result in commit failures. In practice, the latency of writes to object stores can be tens to hundreds of milliseconds, limiting the write transaction rate to several transactions per second. However, we have found this rate sufficient for virtually all current Delta Lake applications: even applications that ingest streaming data into cloud storage typically have a few highly parallel jobs (e.g., Spark Streaming jobs) doing writes that can batch together many new data objects in a transaction. If higher rates are required in the future, we believe that a custom LogStore that coordinates access to the log, similar to our S3 commit service, could provide significantly faster commit times (e.g. by persisting the end of the log in a low-latency DBMS and asynchronously writing it to the object store). Of course, read transactions at the snapshot isolation level create no contention, as they only read objects in the object store, so any number of these can run concurrently.

4. HIGHER-LEVEL FEATURES IN DELTA

Delta Lake's transactional design enables a wide range of higher-level data management features, similar to many of the facilities in a traditional analytical DBMS. In this section, we discuss some of the most widely used features and the customer use cases or pain points that motivated them.

4.1 Time Travel and Rollbacks

Data engineering pipelines often go awry, especially when ingesting "dirty" data from external systems, but in a traditional data lake design, it is hard to undo updates that added objects into a table. In addition, some workloads, such as machine learning training, require faithfully reproducing an old version of the data (e.g., to compare a new and old training algorithm on the same data). Both of these issues created significant challenges for Databricks users before Delta Lake, requiring them to design complex remediations to data pipeline errors or to duplicate datasets.

Because Delta Lake's data objects and log are immutable, Delta Lake makes it straightforward to query a past snapshot of the data, as in typical MVCC implementations. A client simply needs to read the table state based on an older log record ID. To facilitate time travel, Delta Lake allows users to configure a per-table data retention interval, and supports SQL `AS OF timestamp` and `VERSION AS OF commit_id` syntax for reading past snapshots. Clients can also discover which commit ID they just read or wrote in an operation through Delta Lake's API. For example, we use this API in the MLflow open source project [51] to automatically record the table versions read during an ML training workload.

Users have found time travel especially helpful for fixing errors in data pipelines. For example, to efficiently undo an update that overwrote some users' data, an analyst could use a SQL `MERGE` statement of the table against its previous version as follows:

```
MERGE INTO mytable target
USING mytable TIMESTAMP AS OF <old_date> source
ON source.userId = target.userId
WHEN MATCHED THEN UPDATE SET *
```

We are also developing a `CLONE` command that creates a copy-on-write new version of a table starting at one of its existing snapshots.

4.2 Efficient UPSERT, DELETE and MERGE

Many analytical datasets in enterprises need to be modified over time. For example, to comply with data privacy regulations such as GDPR [27], enterprises need to be able to delete data about a specific user on demand. Even with internal datasets that are not about individuals, old records may need to be updated due to errors in upstream data collection or late-arriving data. Finally, applications that compute an aggregate dataset (e.g., a table summary queried by business analysts) will need to update it over time.

In traditional data lake storage formats, such as a directory of Parquet files on S3, it is hard to perform these updates without stopping concurrent readers. Even then, update jobs must be executed carefully because a failure during the job will leave the table in a partially-updated state. With Delta Lake, all of these operations can be executed transactionally, replacing any updated objects through new `add` and `remove` records in the Delta log. Delta Lake supports standard SQL `UPSERT`, `DELETE` and `MERGE` syntax.

4.3 Streaming Ingest and Consumption

Many data teams wish to deploy streaming pipelines to ETL or aggregate data in real time, but traditional cloud data lakes are difficult to use for this purpose. These teams thus deploy a separate streaming message bus, such as Apache Kafka [11] or Kinesis [2], which often duplicates data and adds management complexity.

We designed Delta Lake so that a table's log can help both data producers and consumers treat it as a message queue, removing the need for separate message buses in many scenarios. This support comes from three main features:

Write Compaction. A simple data lake organized as a collection of objects makes it easy to insert data (just write a new object), but creates an unpleasant tradeoff between write latency and query performance. If writers wish to add new records into a table quickly by writing small objects, readers will ultimately be slowed down due to smaller sequential reads and more metadata operations. In contrast, Delta Lake allows users to run a background process that compacts small data objects transactionally, without affecting readers. Setting `dataChange` flag to `false` on log records that compact

files, described in Section 3.1.2, also allows streaming consumers to ignore these compaction operations altogether if they have already read the small objects. Thus, streaming applications can quickly transfer data to one another by writing small objects, while queries on old data stay fast.

Exactly-Once Streaming Writes. Writers can use the `txn` action type in log records, described in Section 3.1.2, to keep track of which data they wrote into a Delta Lake table and implement "exactly-once" writes. In general, stream processing systems that aim to update data in an external store need some mechanism to make their writes idempotent in order to avoid duplicate writes after a failure. This could be done by ensuring that each record has a unique key in the case of overwrites, or more generally, by atomically updating a "last version written" record together with each write, which can then be used to only write newer changes. Delta Lake facilitates this latter pattern by allowing applications to update an `(appId, version)` pair with each transaction. We use this feature in our Structured Streaming [14] connector to support exactly-once writes for any kind of streaming computation (append, aggregation, upsert, etc).

Efficient Log Tailing. The final tool needed to use Delta Lake tables as message queues is a mechanism for consumers to efficiently find new writes. Fortunately, the storage format for the log, in a series of `.json` objects with lexicographically increasing IDs, makes this easy: a consumer can simply run object store LIST operations starting at the last log record ID it has seen to discover new ones. The `dataChange` flag in log records allows streaming consumers to skip log records that only compact or rearrange existing data, and just read new data objects. It is also easy for a streaming application to stop and restart at the same log record in a Delta Lake table by remembering the last record ID it finished processing.

Combining these three features, we found that many users could avoid running a separate message bus system altogether and use a low-cost cloud object store with Delta to implement streaming pipelines with latency on the order of seconds.

4.4 Data Layout Optimization

Data layout has a large effect on query performance in analytical systems, especially because many analytical queries are highly selective. Because Delta Lake can update the data structures that represent a table transactionally, it can support a variety of layout optimizations without affecting concurrent operations. For example, a background process could compact data objects, change the record order within these objects, or even update auxiliary data structures such as data statistics and indexes without impacting other clients. We take advantage of this property to implement a number of data layout optimization features:

OPTIMIZE Command. Users can manually run an `OPTIMIZE` command on a table that compacts small objects without affecting ongoing transactions, and computes any missing statistics. By default, this operation aims to make each data object 1 GB in size, a value that we found suitable for many workloads, but users can customize this value.

Z-Ordering by Multiple Attributes. Many datasets receive highly selective queries along multiple attributes. For example, one network security dataset that we worked with stored information about data sent on the network in as (sourceIp, destIp, time) tuples, with highly selective queries along each of these dimensions. A simple directory partitioning scheme, as in Apache Hive [45], can help to partition the data by a few attributes once it is written, but the number of partitions becomes prohibitively large when using multiple attributes. Delta Lake supports reorganizing the records

```
1 DESCRIBE HISTORY flightdelays
```

▸ (1) Spark Jobs

version	timestamp	userId	userName	operation	notebook
7	2019-10-08T16:47:22	101543	...@databricks.com	MERGE	▸{"notebookId":"25"}
6	2019-10-08T16:44:16	101543	...@databricks.com	MERGE	▸{"notebookId":"25"}
5	2019-10-06T19:26:53	101543	...@databricks.com	UPDATE	▸{"notebookId":"25"}

Figure 3: `DESCRIBE HISTORY` output for a Delta Lake table on Databricks, showing where each update came from.

in a table in Z-order [35] along a given set of attributes to achieve high locality along multiple dimensions. The Z-order curve is an easy-to-compute space-filling curve that creates locality in all of the specified dimensions. It can lead to significantly better performance for query workloads that combine these dimensions in practice, as we show in Section 6. Users can set a Z-order specification on a table and then run `OPTIMIZE` to move a desired subset of the data (e.g., just the newest records) into Z-ordered objects along the selected attributes. Users can also change the order later.

Z-ordering works hand-in-hand with data statistics to let queries read less data. In particular, Z-ordering will tend to make each data object contain a small range of the possible values in each of the chosen attributes, so that more data objects can be skipped when running a selective query.

AUTO OPTIMIZE. On Databricks's cloud service, users can set the `AUTO OPTIMIZE` property on a table to have the service automatically compact newly written data Objects.

More generally, Delta Lake's design also allows maintaining indexes or expensive-to-compute statistics when updating a table. We are exploring several new features in this area.

4.5 Caching

Many cloud users run relatively long-lived clusters for ad-hoc query workloads, possibly scaling the clusters up and down automatically based on their workload. In these clusters, there is an opportunity to accelerate queries on frequently accessed data by caching object store data on local devices. For example, AWS `i3` instances offer 237 GB of NVMe SSD storage per core at roughly 50% higher cost than the corresponding `m5` (general-purpose) instances.

At Databricks, we built a feature to cache Delta Lake data on clusters transparently, which accelerates both data and metadata queries on these tables by caching data and log objects. Caching is safe because data, log and checkpoint objects in Delta Lake tables are immutable. As we show in Section 6, reading from the cache can significantly increase query performance.

4.6 Audit Logging

Delta Lake's transaction log can also be used for audit logging based on `commitInfo` records. On Databricks, we offer a locked-down execution mode for Spark clusters where user-defined functions cannot access cloud storage directly (or call private APIs in Apache Spark), which allows us to ensure that only the runtime engine can write `commitInfo` records, and ensures an immutable audit log. Users can view the history of a Delta Lake table using the `DESCRIBE HISTORY` command, as shown in Figure 3. Commit information logging is also available in the open source version of

Delta Lake. Audit logging is a data security best practice that is increasingly mandatory for many enterprises due to regulation.

4.7 Schema Evolution and Enforcement

Datasets maintained over a long time often require schema updates, but storing these datasets as "just a bunch of objects" means that older objects (e.g., old Parquet files) might have the "wrong" schema. Delta Lake can perform schema changes transactionally and update the underlying objects along with the schema change if needed (e.g., delete a column that the user no longer wishes to retain). Keeping a history of schema updates in the transaction log can also allow using older Parquet objects without rewriting them for certain schema changes (e.g., adding columns). Equally importantly, Delta clients ensure that newly written data follows the table's schema. These simple checks have caught many user errors appending data with the wrong schema that had been challenging to trace down when individual jobs were simply writing Parquet files to the same directory before the use of Delta Lake.

4.8 Connectors to Query and ETL Engines

Delta Lake provides full-fledged connectors to Spark SQL and Structured Streaming using Apache Spark's data source API [16]. In addition, it currently provides read-only integrations with several other systems: Apache Hive, Presto, AWS Athena, AWS Redshift, and Snowflake, enabling users of these systems to query Delta tables using familiar tools and join them with data in these systems. Finally, ETL and Change Data Capture (CDC) tools including Fivetran, Informatica, Qlik and Talend can write to Delta Lake [33, 26].

Several of the query engine integrations use a special mechanism that was initially used for symbolic links in Hive, called *symlink manifest files*. A symlink manifest file is a text file in the object store or file system that contains a lists of paths that should be visible in a directory. Various Hive-compatible systems can look for such manifest files, usually named `_symlink_format_manifest`, when they read a directory, and then treat the paths specified in the manifest file as the contents of the directory. In the context of Delta Lake, manifest files allow us to expose as static snapshot of the Parquet data objects that make up a table to readers that support this input format, by simply creating a manifest file that lists those objects. This file can be written atomically for each directory, which means that systems that read from a non-partitioned Delta table see a fully consistent read-only snapshot of the table, while systems that read from a partitioned table see a consistent snapshot of each partition directory. To generate manifest files for a table, users run a simple SQL command. They can then load the data as an external table in Presto, Athena, Redshift or Snowflake.

In other cases, such as Apache Hive, the open source community has designed a Delta Lake connector using available plugin APIs.

5. DELTA LAKE USE CASES

Delta Lake is currently in active use at thousands of Databricks customers, where it processes exabytes of data per day, as well as at other organizations in the open source community [26]. These use cases span a variety of data sources and applications. The data types stored in Delta Lake include Change Data Capture (CDC) logs from enterprise OLTP systems, application logs, time series data, graphs, aggregate tables for reporting, and image or feature data for machine learning (ML). The applications running over this data include SQL workloads (the most common application type), business intelligence, streaming, data science, machine learning and graph analytics. Delta Lake is a good fit for most data lake applications that would have used structured storage formats such as Parquet or ORC, and many traditional data warehousing workloads.

Across these use cases, we found that customers often use Delta Lake to simplify their enterprise data architectures, by running more workloads directly against cloud object stores and creating a "lakehouse" system with both data lake and transactional features. For example, consider a typical data pipeline that loads records from multiple sources—say, CDC logs from an OLTP database and sensor data from a facility—and then passes it through ETL steps to make derived tables available for data warehousing and data science workloads (as in Figure 1). A traditional implementation would need to combine message queues such as Apache Kafka [11] for any results that need to be computed in real time, a data lake for long-term storage, and a data warehouse such as Redshift [3] for users that need fast analytical queries by leveraging indexes and fast node-attached storage device (e.g., SSDs). This requires multiple copies of the data and constantly running ingest jobs into each system. With Delta Lake, several of these storage systems can be replaced with object store tables depending on the workloads, taking advantage of features such as ACID transactions, streaming I/O and SSD caching to regain some of the performance optimizations in each specialized system. Although Delta Lake clearly cannot replace *all* the functionality in the systems we listed, we found that in many cases it can replace at least some of them. Delta's connectors (§4.8) also enable querying it from many existing engines.

In the rest of this section, we detail several common use cases.

5.1 Data Engineering and ETL

Many organizations are migrating ETL/ELT and data warehousing workloads to the cloud to simplify their management, while others are augmenting traditional enterprise data sources (e.g., point-of-sale events in OLTP systems) with much larger data streams from other sources (e.g., web visits or inventory tracking systems) for downstream data and machine learning applications. These applications all require a reliable and easy-to-maintain data engineering / ETL process to feed them with data. When organizations deploy their workloads to the cloud, we found that many of them prefer using cloud object stores as a landing area (data lake) to minimize storage costs, and then compute derived datasets that they load into more optimized data warehouse systems (perhaps with node-attached storage). Delta Lake's ACID transactions, UPSERT/MERGE support and time travel features allow these organizations to reuse existing SQL queries to perform their ETL process directly on the object store, and to leverage familiar maintenance features such as rollbacks, time travel and audit logs. Moreover, using a single storage system (Delta Lake) instead of a separate data lake and warehouse reduces the latency to make new data queryable by removing the need for a separate ingest process. Finally, Delta Lake's support of both SQL and programmatic APIs (via Apache Spark) makes it easy to write data engineering pipelines using a variety of tools.

This data engineering use case is common in virtually all the data and ML workloads we encountered, spanning industries such as financial services, healthcare and media. In many cases, once their basic ETL pipeline is complete, organizations also expose part of their data to new workloads, which can simply run on separate clusters accessing the same object store with Delta Lake (e.g., a data science workload using PySpark). Other organizations convert parts of the pipeline to streaming queries using tools as Spark's Structured Streaming (streaming SQL) [14]. These other workloads can easily run on new cloud VMs and access the same tables.

5.2 Data Warehousing and BI

Traditional data warehouse systems combine ETL/ELT functionality with efficient tools to query the tables produced to enable interactive query workloads such as business intelligence (BI). The key

technical features to support these workloads are usually efficient storage formats (e.g. columnar formats), data access optimizations such as clustering and indexing, fast storage hardware, and a suitably optimized query engine [43]. Delta Lake can support all these features directly for tables in a cloud object store, through its combination of columnar formats, data layout optimization, max-min statistics, and SSD caching, all of which can be implemented reliably due to its transactional design. Thus, we have found that most Delta Lake users also run ad-hoc query and BI workloads against their lakehouse datasets, either through SQL directly or through BI software such as Tableau. This use case is common enough that Databricks has developed a new vectorized execution engine for BI workloads [21], as well as optimizations to its Spark runtime. Like in the case of ETL workloads, one advantage of running BI directly on Delta Lake is that it is easier to give analysts fresh data to work on, since the data does not need to be loaded into a separate system.

5.3 Compliance and Reproducibility

Traditional data lake storage formats were designed mostly for immutable data, but new data privacy regulation such as the EU's GDPR [27], together with industry best practices, require organizations to have an efficient way to delete or correct data about individual users. We have seen organizations multiple industries convert existing cloud datasets to Delta Lake in order to use its efficient UPSERT, MERGE and DELETE features. Users also leverage the audit logging feature (Section 4.6) for data governance.

Delta Lake's time travel support is also useful for reproducible data science and machine learning. We have integrated Delta Lake with MLflow [51], an open source model management platform developed at Databricks, to automatically record which version of a dataset was used to train an ML model and let developers reload it.

5.4 Specialized Use Cases

5.4.1 Computer System Event Data

One of the largest single use cases we have seen deploys Delta Lake as a Security Information and Event Management (SIEM) platform at a large technology company. This organization logs a wide range of computer system events throughout the company, such as TCP and UDP flows on the network, authentication requests, SSH logins, etc., into a centralized set of Delta Lake tables that span well into the petabytes. Multiple programmatic ETL, SQL, graph analytics and machine learning jobs then run against these tables to search for known patterns that indicate an intrusion (e.g., suspicious login events from a user, or a set of servers exporting a large amount of data). Many of these are streaming jobs to minimize the time to detect issues. In addition, over 100 analysts query the source and derived Delta Lake tables directly to investigate suspicious alerts or to design new automated monitoring jobs.

This information security use case is interesting because it is easy to collect vast amounts of data automatically (hundreds of terabytes per day in this deployment), because the data has to be kept for a long time to allow forensic analysis for newly discovered intrusions (sometimes months after the fact), and because the data needs to be queried along multiple dimensions. For example, if an analyst discovers that a particular server was once compromised, she may wish to query network flow data by source IP address (to see what other servers the attacker reached from there), by destination IP address (to see how the attacker logged into the original server), by time, and by any number of other dimensions (e.g., an employee access token that this attacker obtained). Maintaining heavyweight index structures for these multi-petabyte datasets would be highly expensive, so this organization uses Delta Lake's `ZORDER BY` feature to rearrange the records within Parquet objects to provide clustering across many dimensions. Because forensic queries along these dimensions are highly selective (e.g., looking for one IP address out of millions), Z-ordering combines well with Delta Lake min/max statistics-based skipping to significantly reduce the number of objects that each query has to read. Delta Lake's `AUTO OPTIMIZE` feature, time travel and ACID transactions have also played a large role in keeping these datasets correct and fast to access despite hundreds of developers collaborating on the data pipeline.

5.4.2 Bioinformatics

Bioinformatics is another domain where we have seen Delta Lake used extensively to manage machine-generated data. Numerous data sources, including DNA sequencing, RNA sequencing, electronic medical records, and time series from medical devices, have enabled biomedical companies to collect detailed information about patients and diseases. These data sources are often joined against public datasets, such as the UK Biobank [44], which holds sequencing information and medical records for 500,000 individuals.

Although traditional bioinformatics tools have used custom data formats such as SAM, BAM and VCF [34, 24], many organizations are now storing this data in data lake formats such as Parquet. The Big Data Genomics project [37] pioneered this approach. Delta Lake further enhances bioinformatics workloads by enabling fast multi-dimensional queries (through Z-ordering), ACID transactions, and efficient UPSERTs and MERGEs. In several cases, these features have led to over $100\times$ speedups over previous Parquet implementations. In 2019, Databricks and Regeneron released Glow [28], an open source toolkit for genomics data that uses Delta for storage.

5.4.3 Media Datasets for Machine Learning

One of the more surprising applications we have seen is using Delta Lake to manage multimedia datasets, such as a set of images uploaded to a website that needs to be used for machine learning. Although images and other media files are already encoded in efficient binary formats, managing these datasets as collections of millions of objects in a cloud object store is challenging because each object is only a few kilobytes in size. Object store LIST operations can take minutes to run, and it is also difficult to read enough objects in parallel to feed a machine learning inference job running on GPUs. We have seen multiple organizations store these media files as `BINARY` records in a Delta table instead, and leverage Delta for faster inference queries, stream processing, and ACID transactions. For example, leading e-commerce and travel companies are using this approach to manage the millions of user-uploaded images.

6. PERFORMANCE EXPERIMENTS

In this section, we motivate some of Delta Lake's features through performance experiments. We study (1) the impact of tables with a large number of objects or partitions on open source big data systems, which motivates Delta Lake's decision to centralize metadata and statistics in checkpoints, and (2) the impact of Z-ordering on a selective query workload from a large Delta Lake use case. We also show that Delta improves query performance vs. Parquet on TPC-DS and does not add significant overhead for write workloads.

6.1 Impact of Many Objects or Partitions

Many of the design decisions in Delta Lake stem from the high latency of listing and reading objects in cloud object stores. This latency can make patterns like loading a stream as thousands of small objects or creating Hive-style partitioned tables with thousands of partitions expensive. Small files are also often a problem in HDFS [36], but the performance impact is worse with cloud storage.

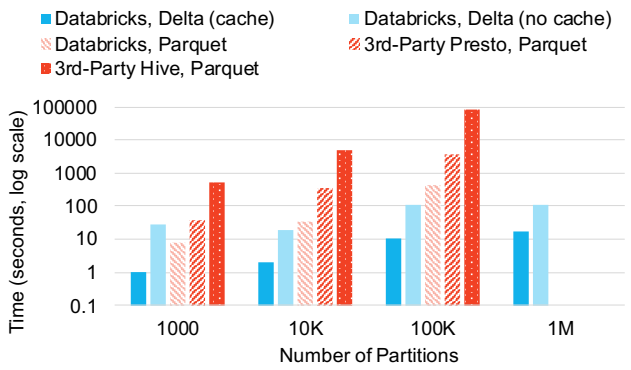


Figure 4: Performance querying a small table with a large number of partitions in various systems. The non-Delta systems took over an hour for 1 million partitions so we do not include their results there.

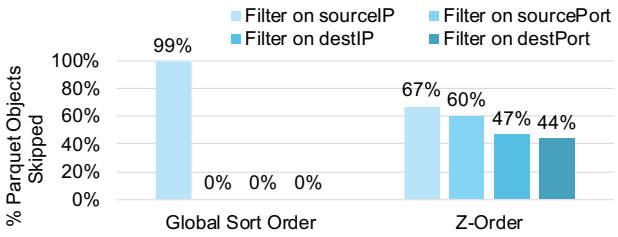


Figure 5: Percent of Parquet objects in a 100-object table that could be skipped using min/max statistics for either a global sort order on the four fields, or Z-order.

To evaluate the impact of a high number of objects, we created 16-node AWS clusters of `i3.2xlarge` VMs (where each VM has 8 vCPUs, 61 GB RAM and 1.9 TB SSD storage) using Databricks and a popular cloud vendor that offers hosted Apache Hive and Presto. We then created small tables with 33,000,000 rows but between 1000 and 1,000,000 partitions in S3, to measure just the metadata overhead of a large number of partitions, and ran a simple query that sums all the records. We executed this query on Apache Spark as provided by the Databricks Runtime [25] (which contains optimizations over open source Spark) and Hive and Presto as offered by the other vendor, on both Parquet and Delta Lake tables. As shown in Figure 4, Databricks Runtime with Delta Lake significantly outperforms the other systems, even without the SSD cache. Hive takes more than an hour to find the objects in a table with only 10,000 partitions, which is a reasonable number of to expect when partitioning a table by date and one other attribute, and Presto takes more than an hour for 100,000 partitions. Databricks Runtime listing Parquet files completes in 450 seconds with 100,000 partitions, largely because we have optimized it to run LIST requests in parallel across the cluster. However, Delta Lake takes 108 seconds even with 1 million partitions, and only 17 seconds if the log is cached on SSDs.

While millions of Hive partitions may seem unrealistic, real-world petabyte-scale tables using Delta Lake do contain hundreds of millions of objects, and listing these large objects is as expensive as listing the small objects in our experiment.

6.2 Impact of Z-Ordering

To motivate Z-ordering, we evaluate the percent of data objects in a table skipped using Z-ordering compared to partitioning or sorting the table by a single column. We generate a dataset inspired by the information security use case in Section 5.4.1, with four fields: `sourceIP`, `sourcePort`, `destIP` and `destPort`, where each record represents a network flow. We generate records by selecting 32-bit IP addresses and 16-bit port numbers uniformly at random, and we store the table as 100 Parquet objects. We then evaluate the number of objects that can be skipped in queries that search for records matching a specific value in each of the dimensions (e.g., `SELECT SUM(col) WHERE sourceIP = "127.0.0.1"`).

Figure 5 shows the results using either (1) a global sort order (specifically, `sourceIP`, `sourcePort`, `destIP` and `destPort` in that order) and (2) Z-ordering by these four fields. With the global order, searching by source IP results in effective data skipping using the min/max column statistics for the Parquet objects (most queries only need to read one of the 100 Parquet objects), but searching by any other field is ineffective, because each file contains many records and its min and max values for those columns are close to the min and max for the whole dataset. In contrast, Z-ordering by all four columns allows skipping at least 43% of the Parquet objects for queries in each dimension, and 54% on average if we assume that queries in each dimension are equally likely (compared to 25% for the single sort order). These improvements are higher for tables with even more Parquet objects because each object contains a smaller range of the Z-order curve, and hence, a smaller range of values in each dimension. For example, multi-attribute queries on a 500 TB network traffic dataset at the organization described in Section 5.4.1, Z-ordered using multiple fields similar to this experiment, were able to skip 93% of the data in the table.

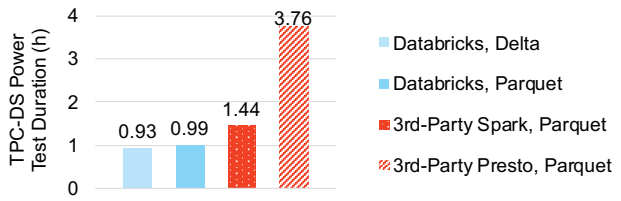


Figure 6: TPC-DS power test duration for Spark on Databricks and Spark and Presto on a third-party cloud service.

6.3 TPC-DS Performance

To evaluate end-to-end performance of Delta Lake on a standard DBMS benchmark, we ran the TPC-DS power test [47] on Databricks Runtime (our implementation of Apache Spark) with Delta Lake and Parquet file formats, and on the Spark and Presto implementations in a popular cloud service. Eacy system ran one master and 8 workers on `i3.2xlarge` AWS VMs, which have 8 vCPUs each. We used 1 TB of total TPC-DS data in S3, with fact tables partitioned on the surrogate key date column. Figure 6 shows the average duration across three runs of the test in each configuration. We see that Databricks Runtime with Delta Lake outperforms all the other configurations. In this experiment, some of Delta Lake's advantages handling large numbers of partitions (Section 6.1) do not manifest because many tables are small, but Delta Lake does provide a speedup over Parquet, primarily due to speeding up the longer queries in the benchmark. The execution and query planning optimizations in Databricks Runtime account for the difference over the third party Spark service (both are based on Apache Spark 2.4).

6.4 Write Performance

We also evaluated the performance of loading a large dataset into Delta Lake as opposed to Parquet to test whether Delta's statistics collection adds significant overhead. Figure 7 shows the time to load

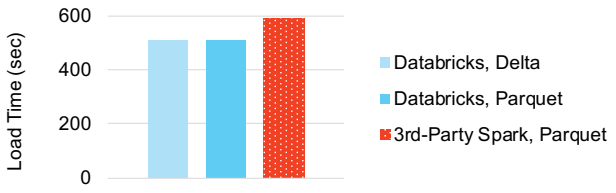


Figure 7: Time to load 400 GB of TPC-DS `store_sales` data into Delta or Parquet format.

a 400 GB TPC-DS `store_sales` table, initially formatted as CSV, on a cluster with one `i3.2xlarge` master and eight `i3.2xlarge` workers (with results averaged over 3 runs). Spark's performance writing to Delta Lake is similar to writing to Parquet, showing that statistics collection does not add a significant overhead over the other data loading work.

7. DISCUSSION AND LIMITATIONS

Our experience with Delta Lake shows that ACID transactions can be implemented over cloud object stores for many enterprise data processing workloads, and that they can support large-scale streaming, batch and interactive workloads. Delta Lake's design is especially attractive because it does not require any other heavy-weight system to mediate access to cloud storage, making it trivial to deploy and directly accessible from a wide range of query engines that support Parquet. Delta Lake's support for ACID then enables other powerful performance and management features.

Nonetheless, Delta Lake's design and the current implementation have some limits that are interesting avenues for future work. First, Delta Lake currently only provides serializable transactions within a single table, because each table has its own transaction log. Sharing the transaction log across multiple tables would remove this limitation, but might increase contention to append log records via optimistic concurrency. For very high transaction volumes, a coordinator could also mediate write access to the log without being part of the read and write path for data objects. Second, for streaming workloads, Delta Lake is limited by the latency of the underlying cloud object store. For example, it is difficult to achieve millisecond-scale streaming latency using object store operations. However, we found that for the large-scale enterprise workloads where users wish to run parallel jobs, latency on the order of a few seconds using Delta Lake tables was acceptable. Third, Delta Lake does not currently support secondary indexes (other than the min-max statistics for each data object), but we have started prototyping a Bloom filter based index. Delta's ACID transactions allow us to update such indexes transactionally with changes to the base data.

8. RELATED WORK

Multiple research and industry projects have sought to adapt data management systems to a cloud environment. For example, Brantner et al. explored building an OLTP database system over S3 [20]; bolt-on consistency [19] implements causal consistency on top of eventually consistent key-value stores; AWS Aurora [49] is a commercial OLTP DBMS with separately scaling compute and storage layers; and Google BigQuery [29], AWS Redshift Spectrum [39] and Snowflake [23] are OLAP DBMSes that can scale computing clusters separately from storage and can read data from cloud object stores. Other work, such as the Relational Cloud project [22], considers how to automatically adapt DBMS engines to elastic, multi-tenant workloads.

Delta Lake shares these works' vision of leveraging widely available cloud infrastructure, but targets a different set of requirements. Specifically, most previous DBMS-on-cloud-storage systems require the DBMS to mediate interactions between clients and storage (e.g., by having clients connect to an Aurora or Redshift frontend server). This creates an additional operational burden (frontend nodes have to always be running), as well as possible scalability, availability or cost issues when streaming large amounts of data through the frontend nodes. In contrast, we designed Delta Lake so that many, independently running clients could coordinate access to a table directly through cloud object store operations, without a separately running service in most cases (except for a lightweight coordinator for log record IDs on S3, as described in §3.2.2). This design makes Delta Lake operationally simple for users and ensures highly scalable reads and writes at the same cost as the underlying object store. Moreover, the system is as highly available as the underlying cloud object store: no other components need to be hardened or restarted for disaster recovery. Of course, this design is feasible here due to the nature of the workload that Delta Lake targets: an OLAP workload with relatively few write transactions per second, but large transaction sizes, which works well with our optimistic concurrency approach.

The closest systems to Delta Lake's design and goals are Apache Hudi [8] and Apache Iceberg [10], both of which define data formats and access protocols to implement transactional operations on cloud object stores. These systems were developed concurrently with Delta Lake and do not provide all its features. For example, neither system provides data layout optimizations such as Delta Lake's `ZORDER BY` (§4.4), a streaming input source that applications can use to efficiently scan new records added to a table (§4.3), or support for local caching as in the Databricks service (§4.5). In addition, Apache Hudi only supports one writer at a time (but multiple readers) [9]. Both projects offer connectors to open source engines including Spark and Presto, but lack connectors to commercial data warehouses such as Redshift and Snowflake, which we implemented using manifest files (§4.8), and to commercial ETL tools.

Apache Hive ACID [32] also implements transactions over object stores or distributed file systems, but it relies on the Hive metastore (running in an OLTP DBMS) to track the state of each table. This can create a bottleneck in tables with millions of partitions, and increases users' operational burden. Hive ACID also lacks support for time travel (§4.1). Low-latency stores over HDFS, such as HBase [7] and Kudu [6], can also combine small writes before writing to HDFS, but require running a separate distributed system.

There is a long line of work to combine high-performance transactional and analytical processing, exemplified by C-Store [43] and HTAP systems. These systems usually have a separate writable store optimized for OLTP and a long-term store optimized for analytics. In our work, we sought instead to support a modest transaction rate without running a separate highly available write store by designing the concurrency protocol to go directly against object stores.

9. CONCLUSION

We have presented Delta Lake, an ACID table storage layer over cloud object stores that enables a wide range of DBMS-like performance and management features for data in low-cost cloud storage. Delta Lake is implemented solely as a storage format and a set of access protocols for clients, making it simple to operate and highly available, and giving clients direct, high-bandwidth access to the object store. Delta Lake is used at thousands of organizations to processes exabytes of data per day, oftentimes replacing more complex architectures that involved multiple data management systems. It is open source under an Apache 2 license at `https://delta.io`.

10. REFERENCES

[1] Amazon Athena. https://aws.amazon.com/athena/.

[2] Amazon Kinesis. https://aws.amazon.com/kinesis/.

[3] Amazon Redshift. https://aws.amazon.com/redshift/.

[4] Amazon S3. https://aws.amazon.com/s3/.

[5] Apache Hadoop. https://hadoop.apache.org.

[6] Apache Kudu. https://kudu.apache.org.

[7] Apache HBase. https://hbase.apache.org.

[8] Apache Hudi. https://hudi.apache.org.

[9] Apache Hudi GitHub issue: Future support for multi-client concurrent write? https://github.com/apache/incubator-hudi/issues/1240.

[10] Apache Iceberg. https://iceberg.apache.org.

[11] Apache Kafka. https://kafka.apache.org.

[12] Apache ORC. https://orc.apache.org.

[13] Apache Parquet. https://parquet.apache.org.

[14] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative API for real-time applications in Apache Spark. In *SIGMOD*, page 601–613, New York, NY, USA, 2018. Association for Computing Machinery.

[15] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53:50–58, 04 2010.

[16] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. In *SIGMOD*, 2015.

[17] Azure Blob Storage. https://https://azure.microsoft.com/en-us/services/storage/blobs/.

[18] Azure Data Lake Storage. https://azure.microsoft.com/en-us/services/storage/data-lake-storage/.

[19] P. Bailis, A. Ghodsi, J. Hellerstein, and I. Stoica. Bolt-on causal consistency. pages 761–772, 06 2013.

[20] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. pages 251–264, 01 2008.

[21] A. Conway and J. Minnick. Introducing Delta Engine. https://databricks.com/blog/2020/06/24/introducing-delta-engine.html.

[22] C. Curino, E. Jones, R. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational cloud: A database-as-a-service for the cloud. In *CIDR*, pages 235–240, 04 2011.

[23] B. Dageville, J. Huang, A. Lee, A. Motivala, A. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, P. Unterbrunner, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, and M. Hentschel. The Snowflake elastic data warehouse. pages 215–226, 06 2016.

[24] P. Danecek, A. Auton, G. Abecasis, C. A. Albers, E. Banks, M. A. DePristo, R. E. Handsaker, G. Lunter, G. T. Marth, S. T. Sherry, G. McVean, R. Durbin, and . G. P. A. Group. The variant call format and VCFtools. *Bioinformatics*, 27(15):2156–2158, 06 2011.

[25] Databricks runtime. https://databricks.com/product/databricks-runtime.

[26] Delta Lake website. https://delta.io.

[27] General Data Protection Regulation. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46. *Official Journal of the European Union*, 59:1–88, 2016.

[28] Glow: An open-source toolkit for large-scale genomic analysis. https://projectglow.io.

[29] Google BigQuery. https://cloud.google.com/bigquery.

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

[31] Google Cloud Storage consistency documentation. https://cloud.google.com/storage/docs/consistency.

[32] Hive 3 ACID documentation from Cloudera. https://docs.cloudera.com/HDPDocuments/HDP3/HDP-3.1.5/using-hiveql/content/hive_3_internals.html.

[33] H. Jaani. New data ingestion network for Databricks: The partner ecosystem for applications, database, and big data integrations into Delta Lake. https://databricks.com/blog/2020/02/24/new-databricks-data-ingestion-network-for-applications-database-and-big-data-integrations-into-delta-lake.html, 2020.

[34] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and 1000 Genome Project Data Processing Subgroup. The sequence alignment/map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, Aug. 2009.

[35] G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. IBM Technical Report, 1966.

[36] S. Naik and B. Gummalla. Small files, big foils: Addressing the associated metadata and application challenges. https://blog.cloudera.com/small-files-big-foils-addressing-the-associated-metadata-and-application-challenges/, 2019.

[37] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, and et al. Rethinking data-intensive science using scalable analytics systems. In *SIGMOD*, page 631–646, New York, NY, USA, 2015. ACM.

[38] OpenStack Swift. https://www.openstack.org/software/releases/train/components/swift.

[39] Querying external data using Amazon Redshift Spectrum. https://docs.aws.amazon.com/redshift/latest/dg/c-using-spectrum.html.

[40] S3 consistency documentation. https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html#ConsistencyModel.

[41] S3 ListObjectsV2 API. https://docs.aws.amazon.com/AmazonS3/latest/API/API_ListObjectsV2.html.

[42] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner. Presto: SQL on everything. In *ICDE*, pages 1802–1813, April 2019.

[43] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB

'05, page 553–564. VLDB Endowment, 2005.

[44] C. Sudlow, J. Gallacher, N. Allen, V. Beral, P. Burton, J. Danesh, P. Downey, P. Elliott, J. Green, M. Landray, B. Liu, P. Matthews, G. Ong, J. Pell, A. Silman, A. Young, T. Sprosen, T. Peakman, and R. Collins. UK Biobank: An open access resource for identifying the causes of a wide range of complex diseases of middle and old age. *PLOS Medicine*, 12(3):1–10, 03 2015.

[45] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005. IEEE, 2010.

[46] M.-L. Tomsen Bukovec. AWS re:Invent 2018. Building for durability in Amazon S3 and Glacier. `https://www.youtube.com/watch?v=nLyppihvhpQ`, 2018.

[47] Transaction Processing Performance Council. TPC benchmark DS standard specification version 2.11.0, 2019.

[48] Understanding block blobs, append blobs, and page blobs. `https://docs.microsoft.com/en-us/rest/api/storageservices/understanding-block-blobs--append-blobs--and-page-blobs`.

[49] A. Verbitski, X. Bao, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, and T. Kharatishvili. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD*, pages 1041–1052, 05 2017.

[50] R. Yao and C. Crosbie. Getting started with new table formats on Dataproc. `https://cloud.google.com/blog/products/data-analytics/getting-started-with-new-table-formats-on-dataproc`.

[51] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, F. Xie, and C. Zumar. Accelerating the machine learning lifecycle with MLflow. *IEEE Data Eng. Bull.*, 41:39–45, 2018.

[52] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *NSDI*, pages 15–28, 2012.