Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics

Michael Armbrust[1], Ali Ghodsi[1,2], Reynold Xin[1], Matei Zaharia[1,3]
[1]Databricks, [2]UC Berkeley, [3]Stanford University

Abstract

This paper argues that the data warehouse architecture as we know it today will wither in the coming years and be replaced by a new architectural pattern, the Lakehouse, which will (i) be based on open direct-access data formats, such as Apache Parquet, (ii) have first-class support for machine learning and data science, and (iii) offer state-of-the-art performance. Lakehouses can help address several major challenges with data warehouses, including data staleness, reliability, total cost of ownership, data lock-in, and limited use-case support. We discuss how the industry is already moving toward Lakehouses and how this shift may affect work in data management. We also report results from a Lakehouse system using Parquet that is competitive with popular cloud data warehouses on TPC-DS.

1 Introduction

This paper argues that the data warehouse architecture as we know it today will wane in the coming years and be replaced by a new architectural pattern, which we refer to as the Lakehouse, characterized by (i) open direct-access data formats, such as Apache Parquet and ORC, (ii) first-class support for machine learning and data science workloads, and (iii) state-of-the-art performance.

The history of data warehousing started with helping business leaders get analytical insights by collecting data from operational databases into centralized warehouses, which then could be used for decision support and business intelligence (BI). Data in these warehouses would be written with schema-on-write, which ensured that the data model was optimized for downstream BI consumption. We refer to this as the first generation data analytics platforms.

A decade ago, the first generation systems started to face several challenges. First, they typically coupled compute and storage into an on-premises appliance. This forced enterprises to provision and pay for the peak of user load and data under management, which became very costly as datasets grew. Second, not only were datasets growing rapidly, but more and more datasets were completely unstructured, e.g., video, audio, and text documents, which data warehouses could not store and query at all.

To solve these problems, the second generation data analytics platforms started offloading all the raw data into *data lakes*: low-cost storage systems with a file API that hold data in generic and usually open file formats, such as Apache Parquet and ORC [8, 9]. This approach started with the Apache Hadoop movement [5], using the Hadoop File System (HDFS) for cheap storage. The data lake was a schema-on-read architecture that enabled the agility of storing any data at low cost, but on the other hand, punted the problem of data quality and governance downstream. In this architecture, a small subset of data in the lake would later be ETLed to a downstream data warehouse (such as Teradata) for the most important decision support and BI applications. The use of open formats also made data lake data directly accessible to a wide range of other analytics engines, such as machine learning systems [30, 37, 42].

From 2015 onwards, cloud data lakes, such as S3, ADLS and GCS, started replacing HDFS. They have superior durability (often >10 nines), geo-replication, and most importantly, extremely low cost with the possibility of automatic, even cheaper, archival storage, e.g., AWS Glacier. The rest of the architecture is largely the same in the cloud as in the second generation systems, with a downstream data warehouse such as Redshift or Snowflake. This two-tier data lake + warehouse architecture is now dominant in the industry in our experience (used at virtually all Fortune 500 enterprises).

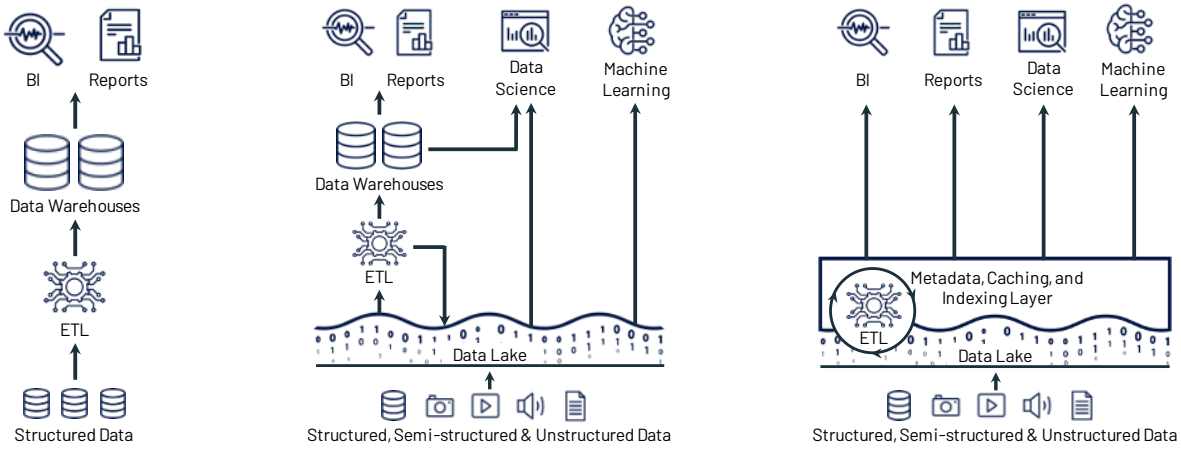
This brings us to the challenges with current data architectures. While the cloud data lake and warehouse architecture is ostensibly cheap due to separate storage (e.g., S3) and compute (e.g., Redshift), a two-tier architecture is highly complex for users. In the first generation platforms, all data was ETLed from operational data systems directly into a warehouse. In today's architectures, data is first ETLed into lakes, and then again ELTed into warehouses, creating complexity, delays, and new failure modes. Moreover, enterprise use cases now include advanced analytics such as machine learning, for which *neither* data lakes nor warehouses are ideal. Specifically, today's data architectures commonly suffer from four problems:

Reliability. Keeping the data lake and warehouse consistent is difficult and costly. Continuous engineering is required to ETL data between the two systems and make it available to high-performance decision support and BI. Each ETL step also risks incurring failures or introducing bugs that reduce data quality, e.g., due to subtle differences between the data lake and warehouse engines.

Data staleness. The data in the warehouse is stale compared to that of the data lake, with new data frequently taking days to load. This is a step back compared to the first generation of analytics systems, where new operational data was immediately available for queries. According to a survey by Dimensional Research and Fivetran, 86% of analysts use out-of-date data and 62% report waiting on engineering resources numerous times per month [47].

Limited support for advanced analytics. Businesses want to ask predictive questions using their warehousing data, e.g., "which customers should I offer discounts to?" Despite much research on the confluence of ML and data management, none of the leading machine learning systems, such as TensorFlow, PyTorch and XGBoost, work well on top of warehouses. Unlike BI queries, which extract a small amount of data, these systems need to process large datasets using complex non-SQL code. Reading this data via ODBC/JDBC

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2021. *11th Annual Conference on Innovative Data Systems Research (CIDR '21)*, January 11–15, 2021, Online.



(a) First-generation platforms. **(b) Current two-tier architectures.** **(c) Lakehouse platforms.**

Figure 1: Evolution of data platform architectures to today's two-tier model (a-b) and the new Lakehouse model (c).

is inefficient, and there is no way to directly access the internal warehouse proprietary formats. For these use cases, warehouse vendors recommend exporting data to files, which further increases complexity and staleness (adding a third ETL step!). Alternatively, users can run these systems against data lake data in open formats. However, they then lose rich management features from data warehouses, such as ACID transactions, data versioning and indexing.

Total cost of ownership. Apart from paying for continuous ETL, users pay double the storage cost for data copied to a warehouse, and commercial warehouses lock data into proprietary formats that increase the cost of migrating data or workloads to other systems.

A straw-man solution that has had limited adoption is to eliminate the data lake altogether and store all the data in a warehouse that has built-in separation of compute and storage. We will argue that this has limited viability, as evidenced by lack of adoption, because it still doesn't support managing video/audio/text data easily or fast direct access from ML and data science workloads.

In this paper, we discuss the following technical question: is it possible to turn data lakes based on standard open data formats, such as Parquet and ORC, into high-performance systems that can provide *both* the performance and management features of data warehouses *and* fast, direct I/O from advanced analytics workloads? We argue that this type of system design, which we refer to as a *Lakehouse* (Fig. 1), is both feasible and is already showing evidence of success, in various forms, in the industry. As more business applications start relying on operational data and on advanced analytics, we believe the Lakehouse is a compelling design point that can eliminate some of the top challenges with data warehousing.

In particular, we believe that the time for the Lakehouse has come due to recent solutions that address the following key problems:

1. Reliable data management on data lakes: A Lakehouse needs to be able to store raw data, similar to today's data lakes, while simultaneously supporting ETL/ELT processes that curate this data to improve its quality for analysis. Traditionally, data lakes have managed data as "just a bunch of files" in semi-structured formats, making it hard to offer some of the key management features that simplify ETL/ELT in data warehouses, such as transactions, rollbacks to old table versions, and zero-copy cloning. However, a recent family of systems such as Delta Lake [10] and Apache Iceberg [7] provide transactional views of a data lake, and enable these management features. Of course, organizations still have to do the hard work of writing ETL/ELT logic to create curated datasets with a Lakehouse, but there are fewer ETL steps overall, and analysts can also easily and performantly query the raw data tables if they wish to, much like in first-generation analytics platforms.

2. Support for machine learning and data science: ML systems' support for direct reads from data lake formats already places them in a good position to efficiently access a Lakehouse. In addition, many ML systems have adopted DataFrames as the abstraction for manipulating data, and recent systems have designed *declarative* DataFrame APIs [11] that enable performing query optimizations for data accesses in ML workloads. These APIs enable ML workloads to directly benefit from many optimizations in Lakehouses.

3. SQL performance: Lakehouses will need to provide state-of-the-art SQL performance on top of the massive Parquet/ORC datasets that have been amassed over the last decade (or in the long term, some other standard format that is exposed for *direct* access to applications). In contrast, classic data warehouses accept SQL and are free to optimize everything under the hood, including proprietary storage formats. Nonetheless, we show that a variety of techniques can be used to maintain *auxiliary* data about Parquet/ORC datasets and to optimize *data layout* within these existing formats to achieve competitive performance. We present results from a SQL engine over Parquet (the Databricks Delta Engine [19]) that outperforms leading cloud data warehouses on TPC-DS.

In the rest of the paper, we detail the motivation, potential technical designs, and research implications of Lakehouse platforms.

2 Motivation: Data Warehousing Challenges

Data warehouses are critical for many business processes, but they still regularly frustrate users with incorrect data, staleness, and high costs. We argue that at least part of each of these challenges is

"accidental complexity" [18] from the way enterprise data platforms are designed, which could be eliminated with a Lakehouse.

First, the top problem reported by enterprise data users today is usually data quality and reliability [47, 48]. Implementing correct data pipelines is intrinsically difficult, but today's two-tier data architectures with a separate lake and warehouse add extra complexity that exacerbates this problem. For example, the data lake and warehouse systems might have different semantics in their supported data types, SQL dialects, etc; data may be stored with different schemas in the lake and the warehouse (e.g., denormalized in one); and the increased number of ETL/ELT jobs, spanning multiple systems, increases the probability of failures and bugs.

Second, more and more business applications require up-to-date data, but today's architectures increase data staleness by having a separate staging area for incoming data before the warehouse and using periodic ETL/ELT jobs to load it. Theoretically, organizations could implement more streaming pipelines to update the data warehouse faster, but these are still harder to operate than batch jobs. In contrast, in the first-generation platforms, warehouse users had immediate access to raw data loaded from operational systems in the same environment as derived datasets. Business applications such as customer support systems and recommendation engines are simply ineffective with stale data, and even human analysts querying warehouses report stale data as a major problem [47].

Third, a large fraction of data is now unstructured in many industries [22] as organizations collect images, sensor data, documents, etc. Organizations need easy-to-use systems to manage this data, but SQL data warehouses and their API do not easily support it.

Finally, most organizations are now deploying machine learning and data science applications, but these are not well served by data warehouses and lakes. As discussed before, these applications need to process large amounts of data with non-SQL code, so they cannot run efficiently over ODBC/JDBC. As advanced analytics systems continue to develop, we believe that giving them direct access to data in an open format will be the most effective way to support them. In addition, ML and data science applications suffer from the same data management problems that classical applications do, such as data quality, consistency, and isolation [17, 27, 31], so there is immense value in bringing DBMS features to their data.

Existing steps towards Lakehouses. Several current industry trends give further evidence that customers are unsatisfied with the two-tier lake + warehouse model. First, in recent years, virtually all the major data warehouses have added support for external tables in Parquet and ORC format [12, 14, 43, 46]. This allows warehouse users to also query the data lake from the same SQL engine, but it does not make data lake tables easier to manage and it does not remove the ETL complexity, staleness, and advanced analytics challenges for data in the warehouse. In practice, these connectors also often perform poorly because the SQL engine is mostly optimized for its internal data format. Second, there is also broad investment in SQL engines that run directly against data lake storage, such as Spark SQL, Presto, Hive, and AWS Athena [3, 11, 45, 50]. However, these engines alone cannot solve all the problems with data lakes and replace warehouses: data lakes still lack basic management features such as ACID transactions and efficient access methods such as indexes to match data warehouse performance.

3 The Lakehouse Architecture

We define a Lakehouse as a data management system based on low-cost and *directly-accessible* storage that also provides traditional analytical DBMS management and performance features such as ACID transactions, data versioning, auditing, indexing, caching, and query optimization. Lakehouses thus combine the key benefits of data lakes and data warehouses: low-cost storage in an open format accessible by a variety of systems from the former, and powerful management and optimization features from the latter. The key question is whether one can combine these benefits in an effective way: in particular, Lakehouses' support for direct access means that they give up some aspects of data independence, which has been a cornerstone of relational DBMS design.

We note that Lakehouses are an especially good fit for cloud environments with separate compute and storage: different computing applications can run on-demand on completely separate computing nodes (e.g., a GPU cluster for ML) while directly accessing the same storage data. However, one could also implement a Lakehouse over an on-premise storage system such as HDFS.

In this section, we sketch one possible design for Lakehouse systems, based on three recent technical ideas that have appeared in various forms throughout the industry. We have been building towards a Lakehouse platform based on this design at Databricks through the Delta Lake, Delta Engine and Databricks ML Runtime projects [10, 19, 38]. Other designs may also be viable, however, as are other concrete technical choices in our high-level design (e.g., our stack at Databricks currently builds on the Parquet storage format, but it is possible to design a better format). We discuss several alternatives and future directions for research.

3.1 Implementing a Lakehouse System

The first key idea we propose for implementing a Lakehouse is to have the system store data in a low-cost object store (e.g., Amazon S3) using a standard file format such as Apache Parquet, but implement a transactional *metadata layer* on top of the object store that defines *which* objects are part of a table version. This allows the system to implement management features such as ACID transactions or versioning within the metadata layer, while keeping the bulk of the data in the low-cost object store and allowing clients to directly read objects from this store using a standard file format in most cases. Several recent systems, including Delta Lake [10] and Apache Iceberg [7] have successfully added management features to data lakes in this fashion; for example, Delta Lake is now used in about half of Databricks' workload, by thousands of customers.

Although a metadata layer adds management capabilities, it is not sufficient to achieve good SQL performance. Data warehouses use several techniques to get state-of-the-art performance, such as storing hot data on fast devices such as SSDs, maintaining statistics, building efficient access methods such as indexes, and co-optimizing the data format and compute engine. In a Lakehouse based on existing storage formats, it is not possible to change the format, but we show that it is possible to implement other optimizations that leave the data files unchanged, including *caching*, *auxiliary data structures* such as indexes and statistics, and *data layout optimizations*.

Finally, Lakehouses can both speed up advanced analytics workloads and give them better data management features thanks to

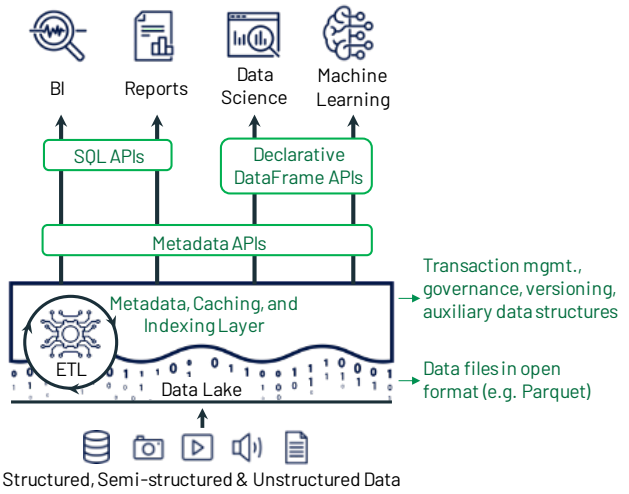


Figure 2: Example Lakehouse system design, with key components shown in green. The system centers around a metadata layer such as Delta Lake that adds transactions, versioning, and auxiliary data structures over files in an open format, and can be queried with diverse APIs and engines.

the development of *declarative DataFrame APIs* [11, 37]. Many ML libraries, such as TensorFlow and Spark MLlib, can already read data lake file formats such as Parquet [30, 37, 42]. Thus, the simplest way to integrate them with a Lakehouse would be to query the metadata layer to figure out which Parquet files are currently part of a table, and simply pass those to the ML library. However, most of these systems support a DataFrame API for data preparation that creates more optimization opportunities. DataFrames were popularized by R and Pandas [40] and simply give users a table abstraction with various transformation operators, most of which map to relational algebra. Systems such as Spark SQL have made this API declarative by lazily evaluating the transformations and passing the resulting operator plan to an optimizer [11]. These APIs can thus leverage the new optimization features in a Lakehouse, such as caches and auxiliary data, to further accelerate ML.

Figure 2 shows how these ideas fit together into a Lakehouse system design. In the next three sections, we expand on these technical ideas in more detail and discuss related research questions.

3.2 Metadata Layers for Data Management

The first component that we believe will enable Lakehouses is metadata layers over data lake storage that can raise its abstraction level to implement ACID transactions and other management features. Data lake storage systems such as S3 or HDFS only provide a low-level object store or filesystem interface where even simple operations, such as updating a table that spans multiple files, are not atomic. Organizations soon began designing richer data management layers over these systems, starting with Apache Hive ACID [33], which tracks which data files are part of a Hive table at a given table version using an OLTP DBMS and allows operations to update this set transactionally. In recent years, new systems have provided even more capabilities and improved scalability. In 2016, Databricks began developing Delta Lake [10], which stores the information about which objects are part of a table *in the data lake itself* as a transaction log in Parquet format, enabling it to scale to billions of objects per table. Apache Iceberg [7], which started at Netflix, uses a similar design and supports both Parquet and ORC storage. Apache Hudi [6], which started at Uber, is another system in this area focused on simplifying streaming ingest into data lakes, although it does not support concurrent writers.

Experience with these systems has shown that they generally provide similar or better performance to raw Parquet/ORC data lakes, while adding highly useful management features such as transactions, zero-copy cloning and time travel to past versions of a table [10]. In addition, they are easy to adopt for organizations that already have a data lake: for example, Delta Lake can convert an existing directory of Parquet files into a Delta Lake table with zero copies just by adding a transaction log that starts with an entry that references all the existing files. As a result, organizations are rapidly adopting these metadata layers: for example, Delta Lake grew to cover half the compute-hours on Databricks in three years.

In addition, metadata layers are a natural place to implement *data quality enforcement* features. For example, Delta Lake implements schema enforcement to ensure that the data uploaded to a table matches its schema, and constraints API [24] that allows table owners to set constraints on the ingested data (e.g., `country` can only be one of a list of values). Delta's client libraries will automatically reject records that violate these expectations or quarantine them in a special location. Customers have found these simple features very useful to improve the quality of data lake based pipelines.

Finally, metadata layers are a natural place to implement *governance* features such as access control and audit logging. For example, a metadata layer can check whether a client is allowed to access a table before granting it credentials to read the raw data in the table from a cloud object store, and can reliably log all accesses.

Future Directions and Alternative Designs. Because metadata layers for data lakes are a fairly new development, there are many open questions and alternative designs. For example, we designed Delta Lake to store its transaction log in the same object store that it runs over (e.g., S3) in order to simplify management (removing the need to run a separate storage system) and offer high availability and high read bandwidth to the log (the same as the object store). However, this limits the rate of transactions/second it can support due to object stores' high latency. A design using a faster storage system for the metadata may be preferable in some cases. Likewise, Delta Lake, Iceberg and Hudi only support transactions on one table at a time, but it should be possible to extend them to support cross-table transactions. Optimizing the format of transaction logs and the size of objects managed are also open questions.

3.3 SQL Performance in a Lakehouse

Perhaps the largest technical question with the Lakehouse approach is how to provide state-of-the-art SQL performance while giving up a significant portion of the data independence in a traditional DBMS design. The answer clearly depends on a number of factors, such as what hardware resources we have available (e.g., can we implement a caching layer on top of the object store) and whether we can change the data object storage format instead of using existing standards such as Parquet and ORC (new designs that improve over

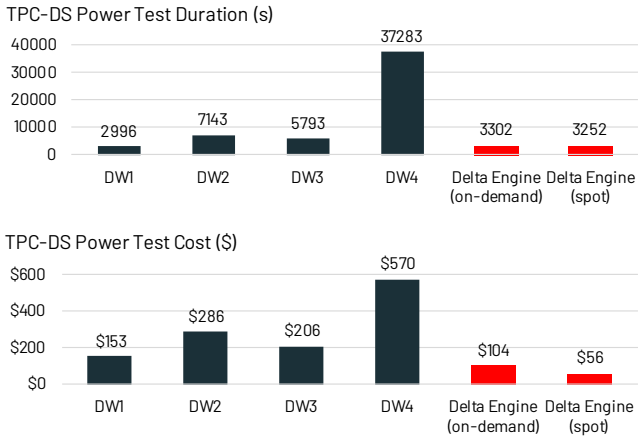


Figure 3: TPC-DS power score (time to run all queries) and cost at scale factor 30K using Delta Engine vs. popular cloud data warehouses on AWS, Azure and Google Cloud.

these formats continue to emerge [15, 28]). Regardless of the exact design, however, the core challenge is that the data storage format becomes part of the system's public API to allow fast direct access, unlike in a traditional DBMS.

We propose several techniques to implement SQL performance optimizations in a Lakehouse *independent of the chosen data format*, which can therefore be applied either with existing or future formats. We have also implemented these techniques within the Databricks Delta Engine [19] and show that they yield competitive performance with popular cloud data warehouses, though there is plenty of room for further performance optimizations. These format-independent optimizations are:

Caching: When using a transactional metadata layer such as Delta Lake, it is safe for a Lakehouse system to cache files from the cloud object store on faster storage devices such as SSDs and RAM on the processing nodes. Running transactions can easily determine when cached files are still valid to read. Moreover, the cache can be in a transcoded format that is more efficient for the query engine to run on, matching any optimizations that would be used in a traditional "closed-world" data warehouse engine. For example, our cache at Databricks partially decompresses the Parquet data it loads.

Auxiliary data: Even though a Lakehouse needs to expose the base table storage format for direct I/O, it can maintain other data that helps optimize queries in auxiliary files that it has full control over. In Delta Lake and Delta Engine, we maintain column min-max statistics for each data file in the table within the same Parquet file used to store the transaction log, which enables data skipping optimizations when the base data is clustered by particular columns. We are also implementing a Bloom filter based index. One can imagine implementing a wide range of auxiliary data structures here, similar to proposals for indexing "raw" data [1, 2, 34].

Data layout: Data layout plays a large role in access performance. Even when we fix a storage format such as Parquet, there are multiple layout decisions that can be optimized by the Lakehouse system. The most obvious is record ordering: which records are clustered together and hence easiest to read together. In Delta Lake, we support ordering records using individual dimensions or space-filling curves such as Z-order [39] and Hilbert curves to provide locality across multiple dimensions. One can also imagine new formats that support placing columns in different orders within each data file, choosing compression strategies differently for various groups of records, or other strategies [28].

These three optimizations work especially well together for the typical access patterns in analytical systems. In typical workloads, most queries tend to be concentrated against a "hot" subset of the data, which the Lakehouse can cache using the same optimized data structures as a closed-world data warehouse to provide competitive performance. For "cold" data the a cloud object store, the main determinant of performance is likely to be the amount of data read per query. In that case, the combination of data layout optimizations (which cluster co-accessed data) and auxiliary data structures such as zone maps (which let the engine rapidly figure out what ranges of the data files to read) can allow a Lakehouse system to minimize I/O the same way a closed-world proprietary data warehouse would, despite running against a standard open file format.

Performance Results. At Databricks, we combined these three Lakehouse optimizations with a new C++ execution engine for Apache Spark called Delta Engine [19]. To evaluate the feasibility of the Lakehouse architecture, Figure 3 compares Delta Engine on TPC-DS at scale factor 30,000 with four widely used cloud data warehouses (from cloud providers as well as third-party companies that run over public clouds), using comparable clusters on AWS, Azure and Google Cloud with 960 vCPUs each and local SSD storage.[1] We report the time to run all 99 queries as well as the total cost for customers in each service's pricing model (Databricks lets users choose spot and on-demand instances, so we show both). Delta Engine provides comparable or better performance than these systems at a lower price point.

Future Directions and Alternative Designs. Designing performant yet directly-accessible Lakehouse systems is a rich area for future work. One clear direction that we have not explored yet is designing new data lake storage formats that will work better in this use case, e.g., formats that provide more flexibility for the Lakehouse system to implement data layout optimizations or indexes over or are simply better suited to modern hardware. Of course, such new formats may take a while for processing engines to adopt, limiting the number of clients that can read from them, but designing a high quality directly-accessible open format for next generation workloads is an important research problem.

Even without changing the data format, there are many types of caching strategies, auxiliary data structures and data layout strategies to explore for Lakehouses [4, 49, 53]. Determining which ones are likely to be most effective for massive datasets in cloud object stores is an open question.

Finally, another exciting research direction is determining when and how to use serverless computing systems to answer queries [41] and optimizing the storage, metadata layer, and query engine designs to minimize latency in this case.

[1] We started all systems with data cached on SSDs when applicable, because some of the warehouses we compared with only supported node-attached storage. However, Delta Engine was only 18% slower when starting with a cold cache.

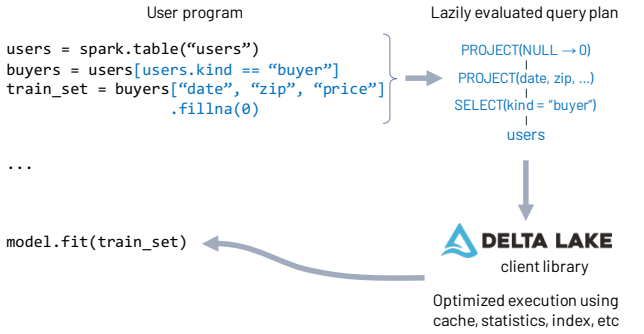


Figure 4: Execution of the declarative DataFrame API used in Spark MLlib. The DataFrame operations in user code execute lazily, allowing the Spark engine to capture a query plan for the data loading computation and pass it to the Delta Lake client library. This library queries the metadata layer to determine which partitions to read, use caches, etc.

3.4 Efficient Access for Advanced Analytics

As we discussed earlier in the paper, advanced analytics libraries are usually written using imperative code that cannot run as SQL, yet they need to access large amounts of data. There is an interesting research question in how to design the data access layers in these libraries to maximize flexibility for the code running on top but still benefit from optimization opportunities in a Lakehouse.

One approach that we've had success with is offering a declarative version of the DataFrame APIs used in these libraries, which maps data preparation computations into Spark SQL query plans and can benefit from the optimizations in Delta Lake and Delta Engine. We used this approach in both Spark DataFrames [11] and in Koalas [35], a new DataFrame API for Spark that offers improved compatibility with Pandas. DataFrames are the main data type used to pass input into the ecosystem of advanced analytics libraries for Apache Spark, including MLlib [37], GraphFrames [21], SparkR [51] and many community libraries, so all of these workloads can enjoy accelerated I/O if we can optimize the DataFrame computation. Spark's query planner pushes selections and projections in the user's DataFrame computation directly into the "data source" plugin class for each data source read. Thus, in our implementation of the Delta Lake data source, we leverage the caching, data skipping and data layout optimizations described in Section 3.3 to accelerate these reads from Delta Lake and thus accelerate ML and data science workloads, as illustrated in Figure 4.

Machine learning APIs are quickly evolving, however, and there are also other data access APIs, such as TensorFlow's `tf.data`, that do not attempt to push query semantics into the underlying storage system. Many of these APIs also focus on overlapping data loading on the CPU with CPU-to-GPU transfers and GPU computation, which has not received much attention in data warehouses. Recent systems work has shown that keeping modern accelerators well-utilized, especially for ML inference, can be a difficult problem [44], so Lakehouse access libraries will need to tackle this challenge.

Future Directions and Alternative Designs. Apart from the questions about existing APIs and efficiency that we have just discussed, we can explore radically different designs for data access interfaces from ML. For example, recent work has proposed "factorized ML" frameworks that push ML logic into SQL joins, and other query optimizations that can be applied for ML algorithms implemented in SQL [36]. Finally, we still need standard interfaces to let data scientists take full advantage of the powerful data management capabilities in Lakehouses (or even data warehouses). For example, at Databricks, we have integrated Delta Lake with the ML experiment tracking service in MLflow [52] to let data scientists easily track the table versions used in an experiment and reproduce that version of the data later. There is also an emerging abstraction of *feature stores* in the industry as a data management layer to store and update the features used in an ML application [26, 27, 31], which would benefit from using the standard DBMS functions in a Lakehouse design, such as transactions and data versioning.

4 Research Questions and Implications

Beyond the research challenges that we raised as future directions in Sections 3.2–3.4, Lakehouses raise several other research questions. In addition, the industry trend toward increasingly feature-rich data lakes has implications for other areas of data systems research.

Are there other ways to achieve the Lakehouse goals? One can imagine other means to achieve the primary goals of the Lakehouse, such as building a massively parallel serving layer for a data warehouse that can support parallel reads from advanced analytics workloads. However, we believe that such infrastructure will be significantly more expensive to run, harder to manage, and likely less performant than giving workloads direct access to the object store. We have not seen broad deployment of systems that add this type of serving layer, such as Hive LLAP [32]. Moreover, this approach punts the problem of selecting an efficient data format for reads to the serving layer, and this format still needs to be easy to transcode from the warehouse's internal format. The main draws of cloud object stores are their low cost, high bandwidth access from elastic workloads, and extremely high availability; all three get worse with a separate serving layer in front of the object store.

Beyond the performance, availability, cost and lock-in challenges with these alternate approaches, there are also important governance reasons why enterprises may prefer to keep their data in an open format. With increasing regulatory requirements about data management, organizations may need to search through old datasets, delete various data, or change their data processing infrastructure on short notice, and standardizing on an open format means that they will always have direct access to the data without blocking on a vendor. The long-term trend in the software industry has been towards open data formats, and we believe that this trend will continue for enterprise data.

What are the right storage formats and access APIs? The access interface to a Lakehouse includes the raw storage format, client libraries to directly read this format (e.g., when reading into TensorFlow), and a high-level SQL interface. There are many different ways to place rich functionality across these layers, such as storage schemes that provide more flexibility to the system by asking readers to perform more sophisticated, "programmable" decoding logic [28]. It remains to be seen which combination of storage formats, metadata layer designs, and access APIs works best.

How does the Lakehouse affect other data management research and trends? The prevalence of data lakes and the increasing use of rich management interfaces over them, whether they be metadata layers or the full Lakehouse design, has implications for several other areas of data management research.

Polystores were designed to solve the difficult problem of querying data across disparate storage engines [25]. This problem will persist in enterprises, but the increasing fraction of data that is available in an open format in a cloud data lake means that many polystore queries could be answered by running directly against the cloud object store, even if the underlying data files are part of logically separate Lakehouse deployments.

Data integration and cleaning tools can also be designed to run in place over a Lakehouse with fast parallel access to all the data, which may enable new algorithms such as running large joins and clustering algorithms over many of the datasets in an organization.

HTAP systems could perhaps be built as "bolt-on" layers in front of a Lakehouse by archiving data directly into a Lakehouse system using its transaction management APIs. The Lakehouse would be able to query consistent snapshots of the data.

Data management for ML may also become simpler and more powerful if implemented over a Lakehouse. Today, organizations are building a wide range of ML-specific data versioning and "feature store" systems [26, 27, 31] that reimplement standard DBMS functionality. It might be simpler to just use a data lake abstraction with DBMS management functions built-in to implement feature store functionality. At the same time, declarative ML systems such as factorized ML [36] could likely run well against a Lakehouse.

Cloud-native DBMS designs such as serverless engines [41] will need to integrate with richer metadata management layers such as Delta Lake instead of just scanning over raw files in a data lake, but may be able to achieve increased performance.

Finally, there is ongoing discussion in the industry about *how to organize data engineering processes and teams*, with concepts such as the "data mesh" [23], where separate teams own different data products end-to-end, gaining popularity over the traditional "central data team" approach. Lakehouse designs lend themselves easily to distributed collaboration structures because all datasets are directly accessible from an object store without having to onboard users on the same compute resources, making it straightforward to share data regardless of which teams produce and consume it.

5 Related Work

The Lakehouse approach builds on many research efforts to design data management systems for the cloud, starting with early work to use S3 as a block store in a DBMS [16] and to "bolt-on" consistency over cloud object stores [13]. It also builds heavily on research to accelerate query processing by building auxiliary data structures around a fixed data format [1, 2, 34, 53].

The most closely related systems are "cloud-native" data warehouses backed by separate storage [20, 29] and data lake systems like Apache Hive [50]. Cloud-native warehouses such as Snowflake and BigQuery [20, 29] have seen good commercial success, but they are still not the primary data store in most large organizations: the majority of data continues to be in data lakes, which can easily store the time-series, text, image, audio and semi-structured formats that high-volume enterprise data arrives in. As a result, cloud warehouse systems have all added support to read external tables in data lake formats [12, 14, 43, 46]. However, these systems cannot provide any management features over the data in data lakes (e.g., implement ACID transactions over it) the same way they do for their internal data, so using them with data lakes continues to be difficult and error-prone. Data warehouses are also not a good fit for large-scale ML and data science workloads due to the inefficiency in streaming data out of them compared to direct object store access.

On the other hand, while early data lake systems purposefully cut down the feature set of a relational DBMS for ease of implementation, the trend in all these systems has been to add ACID support [33] and increasingly rich management and performance features [6, 7, 10]. In this paper, we extrapolate this trend to discuss what technical designs may allow Lakehouse systems to completely replace data warehouses, show quantitative results from a new query engine optimized for a Lakehouse, and sketch some significant research questions and design alternatives in this domain.

6 Conclusion

We have argued that a unified data platform architecture that implements data warehousing functionality over open data lake file formats can provide competitive performance with today's data warehouse systems and help address many of the challenges facing data warehouse users. Although constraining a data warehouses's storage layer to open, directly-accessible files in a standard format appears like a significant limitation at first, optimizations such as caching for hot data and data layout optimization for cold data can allow Lakehouse systems to achieve competitive performance. We believe that the industry is likely to converge towards Lakehouse designs given the vast amounts of data already in data lakes and the opportunity to greatly simplify enterprise data architectures.

Acknowledgements

We thank the Delta Engine, Delta Lake, and Benchmarking teams at Databricks for their contributions to the results we discuss in this work. Awez Syed, Alex Behm, Greg Rahn, Mostafa Mokhtar, Peter Boncz, Bharath Gowda, Joel Minnick and Bart Samwel provided valuable feedback on the ideas in this paper. We also thank the CIDR reviewers for their feedback.

References

[1] I. Alagiannis, R. Borovica-Gajic, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient query execution on raw data files. *CACM*, 58(12):112–121, Nov. 2015.
[2] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: a hands-free adaptive store. In *SIGMOD*, 2014.
[3] Amazon Athena. https://aws.amazon.com/athena/.
[4] G. Ananthanarayanan, A. Ghodsi, A. Warfield, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated memory caching for parallel jobs. In *NSDI*, pages 267–280, 2012.
[5] Apache Hadoop. https://hadoop.apache.org.
[6] Apache Hudi. https://hudi.apache.org.
[7] Apache Iceberg. https://iceberg.apache.org.
[8] Apache ORC. https://orc.apache.org.
[9] Apache Parquet. https://parquet.apache.org.
[10] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. undefineduszczak, M. undefinedwitakowski, M. Szafrański, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, and M. Zaharia. Delta Lake: High-performance ACID table storage over cloud object stores. In *VLDB*, 2020.
[11] 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.

[12] Azure Synapse: Create external file format. https://docs.microsoft.com/en-us/sql/t-sql/statements/create-external-file-format-transact-sql.

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

[14] BigQuery: Creating a table definition file for an external data source. https://cloud.google.com/bigquery/external-table-definition, 2020.

[15] P. Boncz, T. Neumann, and V. Leis. FSST: Fast random access string compression. In *VLDB*, 2020.

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

[17] E. Breck, M. Zinkevich, N. Polyzotis, S. Whang, and S. Roy. Data validation for machine learning. In *SysML*, 2019.

[18] F. Brooks, Jr. No silver bullet – essence and accidents of software engineering. *IEEE Computer*, 20:10–19, April 1987.

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

[20] 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.

[21] A. Dave, A. Jindal, L. E. Li, R. Xin, J. Gonzalez, and M. Zaharia. GraphFrames: An integrated API for mixing graph and relational queries. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, New York, NY, USA, 2016. Association for Computing Machinery.

[22] D. Davis. AI unleashes the power of unstructured data. https://www.cio.com/article/3406806/, 2019.

[23] Z. Dehghani. How to move beyond a monolithic data lake to a distributed data mesh. https://martinfowler.com/articles/data-monolith-to-mesh.html, 2019.

[24] Delta Lake constraints. https://docs.databricks.com/delta/delta-constraints.html, 2020.

[25] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik. The BigDAWG polystore system. *SIGMOD Rec.*, 44(2):11–16, Aug. 2015.

[26] Data Version Control (DVC). https://dvc.org.

[27] Feast: Feature store for machine learning. https://feast.dev, 2020.

[28] B. Ghita, D. G. Tomé, and P. A. Boncz. White-box compression: Learning and exploiting compact table representations. In *CIDR*. www.cidrdb.org, 2020.

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

[30] Getting data into your H2O cluster. https://docs.h2o.ai/h2o/latest-stable/h2o-docs/getting-data-into-h2o.html, 2020.

[31] K. Hammar and J. Dowling. Feature store: The missing data layer in ML pipelines? https://www.logicalclocks.com/blog/feature-store-the-missing-data-layer-in-ml-pipelines, 2018.

[32] Hive LLAP. https://cwiki.apache.org/confluence/display/Hive/LLAP, 2020.

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

[34] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my data files. here are my queries. where are my results? In *CIDR*, 2011.

[35] koalas library. https://github.com/databricks/koalas, 2020.

[36] S. Li, L. Chen, and A. Kumar. Enabling and optimizing non-linear feature interactions in factorized linear algebra. In *SIGMOD*, page 1571–1588, 2019.

[37] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine learning in Apache Spark. *J. Mach. Learn. Res.*, 17(1):1235–1241, Jan. 2016.

[38] Databricks ML runtime. https://databricks.com/product/machine-learning-runtime.

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

[40] pandas Python data analysis library. https://pandas.pydata.org, 2017.

[41] M. Perron, R. Castro Fernandez, D. DeWitt, and S. Madden. Starling: A scalable query engine on cloud functions. In *SIGMOD*, page 131–141, 2020.

[42] Petastorm. https://github.com/uber/petastorm.

[43] Redshift CREATE EXTERNAL TABLE. https://docs.aws.amazon.com/redshift/latest/dg/r_CREATE_EXTERNAL_TABLE.html, 2020.

[44] D. Richins, D. Doshi, M. Blackmore, A. Thulaseedharan Nair, N. Pathapati, A. Patel, B. Daguman, D. Dobrijalowski, R. Illikkal, K. Long, D. Zimmerman, and V. Janapa Reddi. Missing the forest for the trees: End-to-end ai application performance in edge data centers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 515–528, 2020.

[45] 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.

[46] Snowflake CREATE EXTERNAL TABLE. https://docs.snowflake.com/en/sql-reference/sql/create-external-table.html, 2020.

[47] Fivetran data analyst survey. https://fivetran.com/blog/analyst-survey, 2020.

[48] M. Stonebraker. Why the 'data lake' is really a 'data swamp'. BLOG@CACM, 2014.

[49] L. Sun, M. J. Franklin, J. Wang, and E. Wu. Skipping-oriented partitioning for columnar layouts. *Proc. VLDB Endow.*, 10(4):421–432, Nov. 2016.

[50] A. Thusoo et al. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE*, pages 996–1005. IEEE, 2010.

[51] S. Venkataraman, Z. Yang, D. Liu, E. Liang, H. Falaki, X. Meng, R. Xin, A. Ghodsi, M. Franklin, I. Stoica, and M. Zaharia. SparkR: Scaling R programs with Spark. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1099–1104, New York, NY, USA, 2016. Association for Computing Machinery.

[52] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. 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.

[53] M. Ziauddin, A. Witkowski, Y. J. Kim, D. Potapov, J. Lahorani, and M. Krishna. Dimensions based data clustering and zone maps. *Proc. VLDB Endow.*, 10(12):1622–1633, Aug. 2017.