Active Data Lakes: Regaining Physical Data Independence Without Losing Interoperability

Pascal Ginter
Technical University of Munich
pascal.ginter@tum.de

Viktor Leis
Technical University of Munich
leis@in.tum.de

ABSTRACT

Data lakes aim to avoid vendor lock-in and enable interoperability between different query engines on a single copy of data. While early data lakes were only collections of files in various formats, they have since evolved to incorporate some features traditionally associated with relational databases. Today, Apache Parquet is the de facto standard file format for relational data in data lakes. This standardization is fundamental to interoperability, but it comes at the cost of physical data independence because query engines integrate tightly with Parquet. As a result, adoption of novel approaches in the areas of file formats, access paths, and storage media has been limited. We propose the Active Data Lake architecture as a way to restore physical data independence and demonstrate its potential experimentally through three example optimizations.

PVLDB Reference Format:
Pascal Ginter and Viktor Leis. Active Data Lakes: Regaining Physical Data Independence Without Losing Interoperability. PVLDB, 19(6): XXX-XXX, 2026.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ActiveDataLake/vldb-26.

1 INTRODUCTION

Lakehouse architecture. The lakehouse architecture [82] has gained significant traction as a paradigm for data storage and analytics in the cloud. The lakehouse aims to unite the benefits of the *data lake*, namely interoperability between query engines through a single copy of data in open formats, with the performance and seamless user experience of a *data warehouse*. Traditional data lakes store data in open formats such as Apache Parquet [33] loosely organized in directories. Lakehouse systems extend this design with a layer of metadata and a transaction protocol. These enhancements, known as *Open Table Formats (OTFs)*, improve performance and restore essential database features such as ACID transactions and schema evolution to the data lake setting. The most widely adopted OTFs are Apache Iceberg [31], Delta Lake [4], and Apache Hudi [28]. Today, all major open-source systems and cloud data warehouses support at least one OTF [9, 18, 22, 25, 45, 50, 59, 63, 69, 80].

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 6 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Parquet and format ossification. The Parquet file format has become the de facto standard for interoperability and is deeply integrated into most query engines. While the lakehouse (and data lake) approach of giving query engines direct access to a standardized file format achieves the goal of interoperability, it has stifled innovation on the storage layer. Should any engine start writing data in a format other than Parquet, interoperability breaks, thereby limiting real-world adoption of alternative file formats. Not only have file formats with faster decoding schemes such as BtrBlocks [49] or FastLanes [1, 2] not seen meaningful real-world adoption, even Parquet encodings that were added a decade ago (Parquet v2) remain underutilized due to concerns that the consuming engine might lack support for the used encoding and thus be unable to read the produced file [48]. While recent efforts on self-describing encodings [39, 83] enable the evolution of encodings, the evolution of entire file formats remains unsolved.

Limitations of lakehouses. Beyond stagnation in file formats, the lakehouse architecture faces broader challenges. The responsibility of optimizing the data layout is placed on individual clients or background services. Yet no component observes the entire workload, making it difficult to identify the right optimizations. Indexing opportunities are limited: only zone maps and partitioning are used to prune data, leaving significant potential for further optimization. As a result, many vendors develop additional, proprietary acceleration layers, leading to duplicated efforts and undermining the open ecosystem. Additionally, fine-grained governance is nearly impossible, and many design decisions of current table formats are based on the properties of (disk-based) object stores.

Compromised physical data independence. A key principle of relational databases is the ability to modify the physical storage format without requiring changes to application logic. This principle, known as *physical data independence*, is one of Codd's twelve foundational rules for relational database systems [13], and has been a cornerstone of database system architecture. In the lakehouse ecosystem, however, the database has been decomposed into query engine and storage layer, with the query engine effectively acting as an application from the storage layer's perspective. Since the storage layer in modern lakehouses is only a passive repository of files, it lacks the ability to provide physical data independence. The authors of the original lakehouse paper acknowledged this limitation [82] but dismissed its impact, arguing that lakehouses still achieve state-of-the-art performance. While a system that directly exposes data files may be performant at any given moment, it is inherently difficult to evolve. Migrating to a new data file format is prohibitively expensive, and being the first to adopt a new standard comes at the cost of interoperability, creating strong incentives to maintain the status quo. This has led to the stagnation of used data encoding techniques that we observe today.

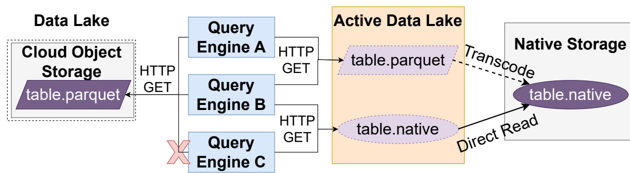

Figure 1: The Active Data Lake architecture decouples the physical data representation from the interface used for interoperability by serving virtualized files (e.g. Parquet).

Active Data Lakes. In this paper, we propose the first architecture that provides both interoperability and physical data independence. Rather than exposing physical data representation through direct access, we introduce a layer of abstraction that decouples the on-wire representation used for interoperability from the physical storage format. Traditional, passive data lakes require all query engines to understand the same file format – in practice Parquet. As the left-hand side of Figure 1 illustrates, this can lead to interoperability issues if a query engine is unable to read the stored Parquet data. In contrast, the Active Data Lake is a server sitting between query engine and storage, which allows access to the same data through multiple interfaces and virtualization. For instance, it allows data written in a new, native data format to remain accessible as a virtualized Parquet file, ensuring backwards compatibility with engines that do not support the new format. This enables interoperability between query engines, even if there is no file format that both engines support. While the Active Data Lake architecture introduces an additional network hop, usually between instances in the same availability zone, the latency of this hop is negligible compared to the latency of cloud object stores.
Contributions. (i) In Section 3, we propose an architecture that enables both interoperability and physical data independence. (ii) We demonstrate that file virtualization can achieve near-native performance in Section 4. This allows offering multiple access methods to a single copy of data. (iii) We show in Section 5 that the Active Data Lake can transparently improve performance through prefetching. (iv) In Section 6, we show that transaction rates can be significantly improved while maintaining existing data lake interfaces.

2 RELATED WORK AND BACKGROUND

2.1 Related Work

Data lake history: From HDFS to lakehouse. Data lakes emerged in the late 2000s as a response to the high cost and vendor lock-in associated with traditional data warehouses. Early implementations ran on Hadoop [27] clusters, using HDFS [66] for storage and MapReduce [20] programs for compute. Hive [79] introduced declarative queries to the data lake ecosystem in 2010 through a SQL interface. In 2013, Parquet [33] and ORC [32] were introduced to standardize data layouts. Gradually, cloud object stores like Amazon S3 replaced HDFS, enabling independent scaling of storage and compute. To address reliability and consistency challenges, OTFs such as Hudi [28], Delta Lake [4], and Iceberg [31] emerged between 2016 and 2018. In 2020, Databricks introduced the term lakehouse [82] to describe OTF-based architectures combining the openness of data lakes with the performance of data warehouses.

DuckLake. In 2025, DuckLake [62] challenged the lakehouse architecture by opting to track metadata in a relational database instead of using files stored on cloud object storage. DuckLake makes the observation that despite OTFs' efforts to rely exclusively on object storage, a database is still required for (multi-table) transactions. Based on this observation, DuckLake elevates the role of the database from catalog to handling and storing all metadata in normalized relational tables. This approach resembles the architecture of systems like Snowflake and BigQuery, which use FoundationDB and Spanner respectively for their metadata [15, 53]. The DuckLake specification does not restrict the transactional DBMS choice as long as the system supports basic SQL. While DuckLake rethinks the metadata layer of the data lake, the data layer remains unchanged and clients continue to directly access data in the form of Parquet files on cloud object storage. Thus, DuckLake does not achieve physical data independence.
Composable data systems. Our proposed architecture continues the trend of composable data systems [61] which breaks the traditional database monolith into modular components that can be mixed and matched to facilitate the development of specialized data systems. While current efforts decompose the database into language frontend (e.g. Ibis [43]), optimizer (e.g. Calcite [8], Orca [71]) and execution engine (e.g. DataFusion [51], Velox [60]), they standardize the data storage layer as OTFs on cloud object storage. Our architecture additionally allows for the modularization and specialization of the data storage layer.
From unstructured files to managed tables. The evolution of data lakes reveals the desire for and the trend towards warehouse-like performance and functionality. This trend is far from complete as ongoing efforts in the areas of multi-table transactions [42, 44, 46], lower commit latency [74], table format agnostic data pruning [14] and fine-granular access control [67], and cross-engine caching [36] demonstrate. However, the current architecture of lakehouses can only partially provide such functionality due to the lack of physical data independence: Concurrent transactions on the same tables are always conflicting, commit latency is bound by the (high) latency of object stores, catalog based data pruning is table but not file format agnostic, fine-grained access control requires trusting all participating clients, and caching needs to be integrated into every single client. In contrast, by providing physical data independence, the Active Data Lake makes these features straightforward.

2.2 Advantages of Lakehouses over Data Lakes

Single table ACID transactions. Through their transaction protocols, OTFs promise ACID compliant transactions. This promise, however, currently holds only for single-statement and single-table transactions (and at very low speeds). All major OTFs are currently working on removing this limitation [42, 44, 46].
Schema evolution. OTFs offer the ability to modify a table's schema without having to rewrite existing data files. To achieve this, OTFs restrict possible schema changes, ensuring that existing files remain interpretable and define clear semantics for reconciling possible differences between the table schema and the schema defined in individual data files.
Learning 1. The demand for database-like functionality led to the rise of lakehouse architectures.

No reliance on listing operations. While data lakes need listing operations to discover all files in the directory associated with a table, OTFs avoid listing operations by explicitly storing the path to all relevant data files in the metadata files. Listing operations are problematic due to their considerable latency and the need for sequential API calls as a single list API call returns at most 1,000 objects on AWS S3 [64] and 5,000 on Azure Blob Storage [6].

File-level pruning abilities. When processing very selective queries over many data files, having to access each data file's footer only to determine that it does not contain qualifying data can significantly degrade performance. OTFs store min/max statistics for each column at file granularity to allow pruning entire files.

Learning 2. Raw data files alone can not match the performance of a data warehouse. Additional, up-to-date metadata is needed for competitive performance.

2.3 The Role of the Catalog

In relational databases, the catalog acts as the central repository of metadata that defines the structure, organization, and access control of data stored within the database. While a core idea of the data lake and lakehouse architectures was to avoid always-on services to eliminate potential sources of failure and improve availability, all OTFs have since introduced such an always-on component in the form of a catalog. Notable catalogs include Apache Polaris (incubating) [37] and Project Nessie [58] for Apache Iceberg, and Unity Catalog [12, 16] for both Iceberg and Delta Lake. Iceberg has introduced a language-agnostic REST API [29] to unify the interfaces of different catalog implementations.

Discoverability. The catalog exposes APIs to list managed tables and maps logical table names to physical locations. Without a catalog, each engine must be manually configured with the physical paths of individual tables.

Governance. Catalogs enable role-based access control by hiding unauthorized tables from listings or using credential vending to grant clients temporary access to the underlying storage after authentication.

Transaction handling. In Apache Iceberg, the catalog is responsible for atomically swapping the pointer to the current top-level metadata file, which is usually implemented using a relational database. In Apache Hudi and Delta Lake, the catalog is currently not involved with transactions. However, current proposals [42, 46] would introduce a dependency on the catalog for multi-table transactions despite the existence of catalog-avoiding approaches [40].

Scan planning. A recent addition to the Iceberg catalog REST specification is a catalog-driven scan planning API [14]. Currently, each client has to implement parsing of Iceberg's three metadata file types and file-level pruning based on partitions and zone maps. The new API allows clients to delegate determining which data files need to be accessed for a query to the catalog by communicating their table-level filters. This development introduces new opportunities: By offloading metadata parsing and data pruning to the catalog, clients can abstract away table format details, extending read interoperability to any engine capable of reading Parquet files. Additionally, catalogs can implement optimizations that were previously infeasible, such as leveraging secondary index structures not defined by the Iceberg specification for more efficient file pruning.

Learning 3. While a key design paradigm for data lakes was to rely solely on files stored on cloud object storage, an active component (the catalog) had to be reintroduced. The trend of delegating increasing responsibility to the catalog clearly shows that a significant share of users can accept active components in their architecture.

3 ACTIVE DATA LAKES

Data lakes and lakehouses. Data lakes achieve interoperability by standardizing the storage format, and reduce vendor lock-in by storing data in customer-owned buckets on cloud object storage. However, in doing so they require tight integration with the chosen standard file format and storage device by systems accessing the data lake. As such, they lose the concept of physical data independence, a concept that allows database systems to evolve their internal data format and access paths without requiring every application to be updated to support the new data format.

Active Data Lakes. We propose the Active Data Lake architecture as the first architecture to achieve both interoperability and physical data independence. The Active Data Lake architecture distinguishes itself from the traditional data lake architecture by treating the storage layer as a first-class citizen, making it a standalone, active component instead of a collection of passive (meta-)data files. This decision advances the current trend in lakehouses that rely more and more on an active component in the form of the catalog in order to offer data warehouse-like functionality and performance. A clear distinction from current architectures is the decision not to allow direct reads from storage. All read and write operations pass through the Active Data Lake, making it a central component that is capable of making workload-driven decisions. A dedicated service for storage allows offering query engines multiple access methods for the same physical copy of data through the means of virtualization. This approach decouples the on-disk and on-wire representations, restoring physical data independence.

Virtualized access. The Active Data Lake offers clients the ability to request data as virtualized files in a format that may be distinct from the native representation. These virtual files do not need to be materialized, but can be transcoded efficiently (see Section 4) into the target format when requested. Internally, the Active Data Lake can use a native format that offers better performance than Parquet. Depending on the query engine's capabilities, it can either request the data as a Parquet file, prompting on-the-fly transcoding of the requested data for each HTTP range request or ask the Active Data Lake for the data in the native format, in which case no transformation is necessary and the engine can profit from the improved performance of the native file format.

Virtualization enables format evolution. To illustrate how the Active Data Lake's virtualization capabilities enable the evolution of file formats, consider the scenario depicted in Figure 1. Query engine B has implemented file format X as a replacement for Apache Parquet, which is the current data file format for a particular table. Meanwhile, query engine A still only supports Parquet and has no plans to implement support for format X, for example due to its lack of adoption. Without virtualization, interoperability breaks as soon as engine B starts writing data in the new format X. When using the Active Data Lake, however, query engine A still has the ability to access the entire table, with the data represented as a collection

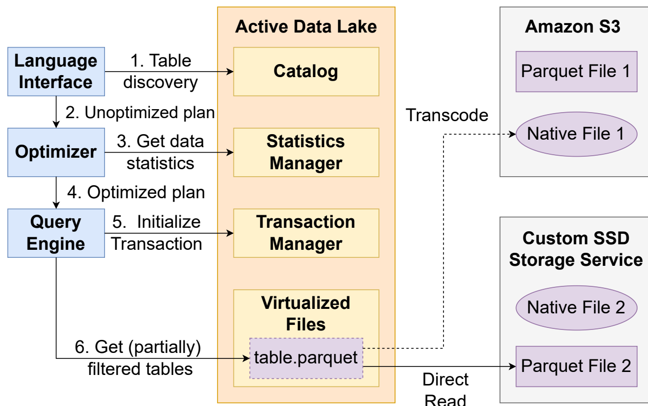

Figure 2: Read query life cycle in the Active Data Lake.

of Parquet files. Years later, when format X has gained adoption, a newly developed query engine C may support only format X and not Parquet. Although query engine A has still not added support for X, query engines A and C can work on the same physical copy of the data through the Active Data Lake's virtualization capabilities. While optimal performance still requires support for the physical file format, it is no longer needed for interoperability.

3.1 The Active Data Lake in Action

To better understand how the Active Data Lake interacts with other components, we examine the lifecycle of a read query as illustrated in Figure 2. The Active Data Lake sits between the execution engines and the physically stored data, allowing it to observe all changes to the data. In Section 8.1, we discuss the impact of such a central component on bandwidth, scalability, and availability. Its interface exposes an internal catalog, which allows the ecosystem to discover available tables and views. The Active Data Lake is also responsible for collecting up-to-date statistics, which can be used by the optimizer to generate an optimized plan for the current query. The query engine initiates a transaction, communicating what tables it intends to read together with their filter predicates. The Active Data Lake then provides a consistent snapshot of the data for the duration of the transaction. The query engine then accesses the already partially filtered data from the Active Data Lake, reducing the amount of data that needs to be transferred and allowing the Active Data Lake to use advanced data pruning and indexing techniques internally.

3.2 Architectural Simplification

The attentive reader might have noticed that the internal architecture of the Active Data Lake shown in Figure 2 closely resembles the internal architecture of a traditional database system. In this section, we discuss how the Active Data Lake architecture deviates from the architecture of existing data lakes, a return to the layered architecture of database systems, and highlight key differences between the Active Data Lake and classical database systems. We focus on the storage layer. Other database system components such as the execution engine, optimizer, and parser are summarized as the query layer. We distinguish between logical data, access paths, physical data, and storage device integration. Figure 3 illustrates our analysis and sketches the architectures of data warehouse, data lake, lakehouse, DuckLake, and the Active Data Lake. Arrows indicate necessary integrations with non-adjacent layers.

Database systems. Database systems and data warehouses are inherently complex systems. To manage this complexity, they traditionally have a layered architecture. Each layer adds some form of abstraction and only needs to integrate with the layer directly below it. This architecture allows the evolution of individual components without requiring changes to the entire system. In particular, applications only need to integrate with the logical data representation, namely the relational model, through SQL, giving the warehouse the ability to evolve the internal data format and processing engine.

Traditional interoperable architectures. Traditional architectures for interoperability depart from a layered architecture and require tight coupling between all clients and both the chosen data file format and underlying storage medium. In pure data lakes, each engine must independently handle access path selection and maintain the mapping between logical and physical data. Lakehouse architectures shift this responsibility to the Open Table Format (OTF) and the catalog, requiring additional integration efforts. DuckLake unifies the two layers into a relational database with a specific schema but still requires every client to integrate with the chosen database schema as well as file format and used storage device.

Active Data Lake. Our architecture elevates the storage layer to a fully-fledged active component. This makes it possible to define a clean interface between query and storage layer which does not necessitate a tight integration of each client with access paths, physical data, and storage devices, simplifying integration with the Active Data Lake. The design follows our learnings from Section 2: We emphasize the active component as it allows us to reintroduce additional database features that are not possible in the lakehouse architecture. Internally, the Active Data Lake can use a clean layered architecture just like the lower levels of a database system.

Active Data Lake vs DBMS. Despite their similar internal architectures, fundamental differences between Active Data Lakes and database systems remain. Unlike database systems, which tightly couple storage and a proprietary query engine, the Active Data Lake only handles the storage layer. This separation of concerns continues the trend of composable data systems and enables interoperability between specialized data processing frameworks on a single copy of the data.

3.3 Additional Benefits

Data access path innovations. Indexing capabilities of existing data lakes are limited: Iceberg and Delta Lake only support partitioning and zone maps to eliminate entire data files. While Hudi supports indexes such as bloom filters, hash-partition-based primary keys, and storing user-defined indexes as black box binary objects [35], the responsibility for using the indexes is again put on the client, leading to the risk that some clients can not use the index structures. This risk is particularly high for user-defined indexes. Not only do these clients miss out on the performance benefits of a particular index, but their inability to update the index leads to stale index structures. The Active Data Lake, in contrast, can use and maintain any index internally without requiring implementation effort from query engines, enabling fast innovation.

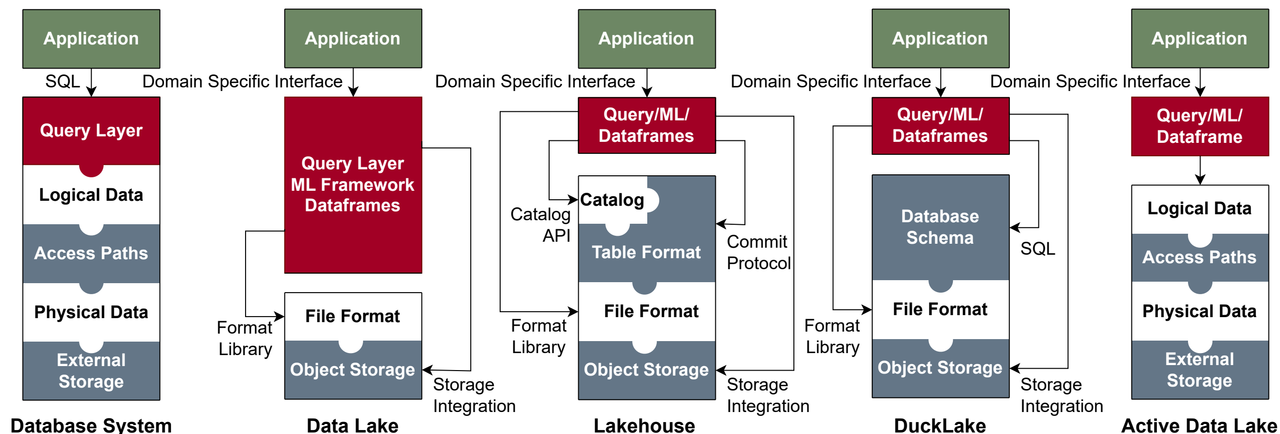

Figure 3: Only database systems and Active Data Lakes maintain clear layering (arrows indicate cross-layer interactions).

Logical data independence. Logical data independence allows changing logical structures, e.g., the schema of a table, without breaking applications. Views are crucial for logical data independence by acting as a stable abstraction layer between the physical schema and application logic. Support for views in OTFs is limited; Hudi does not support them at all, Iceberg and Delta Lake support defining views within the catalog [17, 34]. Their implementation is limited to storing a SQL string and passing it to the client for evaluation. The client is responsible for assembling the view, making it vulnerable to schema changes in the underlying table. Consequently, these formats fall short of full logical data independence. In contrast, the Active Data Lake handles views internally, enabling true logical data independence.

Flexible storage media. The virtualization offered by the Active Data Lake abstracts the physical location of the data. This not only allows clients to access data stored on devices they do not natively support, but also opens up further optimization opportunities. For instance, an Active Data Lake implementation could implement a cross-client caching layer, which is completely transparent to all involved engines. While current efforts in Hudi [36] also work towards cross-client caching, direct access necessitates integration with all participating clients. More generally, the Active Data Lake can transparently implement tiered storage, using low-latency storage for frequently accessed data and metadata, cloud object storage for cold data, and archival storage services such as Amazon S3 Glacier [5] for historical data. In Section 5, we show how the Active Data Lake can be used to improve the performance of clients for environments they are not designed for.

Workload-aware optimizations. Current data layout optimizations are either best effort or user defined. The reason for this is that individual clients have limited knowledge of the total workload on the data lake. In an extreme, but realistic case, a particular client's sole responsibility might be to load data into the data lake while entirely different clients are used to process the imported data. Thus, it is difficult to assess whether data should be optimized for frequent reads, or should be compressed aggressively to reduce storage costs. In contrast, the Active Data Lake is a central component involved in all read and write operations. As such, it can keep track of global workload statistics across all clients, such as how often data files are accessed and whether they contained any data matching the query. Based on this information, the Active Data Lake can accurately estimate the impact of index structures and can use techniques such as Smooth Predicate Acceleration [11] to automatically create only the indexes that result in a reduction of total workload cost.

Governance. Many organizations require the ability to define and enforce access control policies to ensure compliance with legal, organizational, or ethical standards, for instance to restrict or monitor access to sensitive information. While the Iceberg REST specification allows catalogs to use vended credentials to enforce access control at table granularity by only granting clients temporary access to the files that are part of the table, this approach only works at a file granularity. Current proposals to allow column or row-level access control can be divided into two categories: (1) trusting every query engine to enforce the access policies correctly or (2) having a (set of) trusted query engine(s) running in an isolated environment that pre-filter or mask sensitive data. Both approaches have significant drawbacks. Trusting every client opens additional attack vectors as for instance the UDF implementation could be used to inject malicious code or engine side caching could be exploited by malicious programs running on the same machine. The alternative approach relies on engine-specific interfaces to communicate intended filtering and transfer data. One example of this approach is Databricks Lakeguard [41], which uses Spark Connect API [72].

Optimizations apply to all query engines. Individual vendors such as Snowflake [23] or BigQuery [53] collect statistics on top of the information stored in OTFs to improve performance. Dremio and Starburst go a step further by creating variants of a materialized view that is transparently used at query time, but only accessible to their proprietary engines [21, 75]. The vendor specific storage layer enhancements can lead to a performance based vendor lock-in as competing query engines can not leverage these vendor specific optimizations. The Active Data Lake allows all query engines to leverage internal optimizations without integration effort.

3.4 A Compatible Way Forward

Despite the many benefits offered by the Active Data Lake architecture, one might worry about the effort and cost associated with migrating to a new architecture. Fortunately, a gradual transition towards Active Data Lakes is possible. Management of existing tables can be transferred to the Active Data Lake by simply informing it of the location of current metadata in OTF format and deactivating direct access to the transferred tables. The Active Data Lake then uses the current data layout as a starting point for optimization without having to physically copy data. The Active Data Lake then analyzes the workload and gradually applies optimizations such as rewriting the data in a more efficient format. To be more precise, we envision a transition in two stages:

1. Emulation of Iceberg metadata. By implementing the Iceberg REST Catalog API [29], the Active Data Lake can communicate the latest metadata location to the clients. This location can be any service compatible with the interface of cloud object storage, including the Active Data Lake itself. By redirecting the metadata and subsequent data file requests to itself, the Active Data Lake can produce the requested files on demand. This enables the flexibility of abstracting away the actual location and format of the data. However, the lack of communicated intent for the requests prevents the Active Data Lake from pruning any data.

2. Iceberg scan planning API. Integrating with the yet to be adopted Iceberg Scan Planning API allows moving data pruning abilities from the query engines to the Active Data Lake.

4 BENEFIT 1: FILE VIRTUALIZATION

4.1 Motivation and Background

Network bandwidth improvements. Early cloud environments were limited by low-bandwidth networks, making general-purpose compression schemes such as Snappy or LZ4 crucial. However, with recent advances in cloud networking, single instances now have network bandwidths of up to 600 Gbit/s [76]. As a result, the decompression speed of general-purpose compression schemes has replaced network bandwidth as the main performance bottleneck [49]. While general-purpose compression schemes are optional in Parquet, they are essential for achieving significant compression.

Emergence of new file formats. To overcome the decompression bottleneck limiting scan throughput, recent file formats have emerged that apply lightweight, recursive encoding schemes instead of general-purpose compression. These include BtrBlocks [49], FastLanes [1, 2], Nimble [26], Lance [52], and Vortex [73].

Parquet is here to stay. Despite these technical advances, none of the newer formats has achieved widespread adoption. This is not due to the technical superiority of existing systems but because the ecosystem around data processing is resistant to change. Introducing a new file format breaks compatibility with the existing ecosystem. Thus, no engine has an incentive to adopt a new format unilaterally and query engines prioritize optimizing support for established formats, where improvements yield immediate benefits.

Goal. In this section, we show that we can efficiently translate between different formats and encodings, enabling the use of different encodings for the on-wire representation than for the on-disk format. This capability allows internal formats to evolve even with encodings that not all engines can understand. In this section, we show that the Active Data Lake enables saving data in a new format while maintaining compatibility with engines that are only capable of reading Parquet files without significantly impacting their performance. To ensure this, we have the following goals:

(1) Throughput comparable to direct reads without upfront latency.
(2) Complete transparency to the querying engine.

The second point not only ensures compatibility with existing engines but also allows us to leverage the engine's built-in performance optimizations for Parquet files such as projection and pruning based on the footer zone maps. Leveraging the engine's native capabilities is essential for good performance, as the Active Data Lake, at this stage, is not aware of the intent/semantics of the current query and thus is unable to pre-filter the data. As we can not predict which row groups are pruned, and engines might have optimizations that combine requests for nearby column chunks, we need to support answering requests for arbitrary ranges.

Parquet. Parquet partitions a table horizontally into row groups. Within a row group, data is vertically partitioned into column chunks, which are further divided into data pages. Each data page may use a different encoding and is compressed independently using general-purpose techniques. As a self-describing format, each Parquet file includes a variable-sized footer containing the data schema and column chunk statistics to support pruning. Metadata is serialized using Apache Thrift's [68] TCompactProtocol [78].

BtrBlocks. BtrBlocks offers higher decompression speeds than Parquet while maintaining a comparable compression ratio [49] by replacing general-purpose compression with recursively applied lightweight encoding schemes. Unlike Parquet, BtrBlocks stores each column in a separate file and splits columns into parts of equal post-compression size. This splitting occurs at the granularity of chunks, each containing 65,536 tuples and using its own encoding configuration. Each chunk is encoded individually using its own set of encodings. A separate metadata file stores the table schema, maps chunk indices to files, and records per-chunk min/max statistics.

4.2 Implementation

Naive approach. The emergence of Apache Arrow [30] as a common in-memory format for columnar data has led to widespread support across file format libraries for reading from and writing to Arrow. This makes converting between file formats appear straightforward: load the source file into Arrow, convert it to the target format using existing libraries, and serve the result over the network. However, this approach faces several practical limitations: (1) Full file reads: Query engines first request the file footer containing column chunk offsets. Since Parquet's compressed chunk sizes are unknown without actually performing the conversion, the entire file must be read and converted. However, this eliminates the benefits of PAX [3] based file formats and introduces high upfront latency, violating our first design principle. (2) Limited memory size: Materializing the full file in memory requires loading all columns and row groups, even those irrelevant to the query. For large files or concurrent queries, this can exhaust memory and trigger disk spills, adding further latency. Therefore, the naive eager conversion strategy is not suitable. Instead, we need to read and convert only the requested parts of a file on-the-fly.

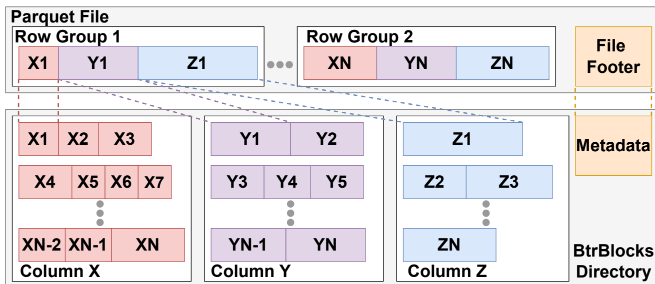

Figure 4: Each chunk in BtrBlocks, corresponds to a column chunk containing a singular data page in Parquet.

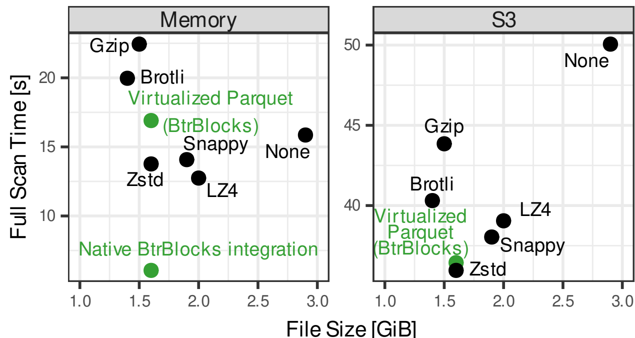

Figure 5: Performance and size of file formats (*lineitem* table at scale factor 10, 1 DuckDB worker thread).

On-the-fly conversion. The requirement of having to compute the entire metadata upfront imposes the constraint of being able to predict the location of individual column chunks within the file before actually reading and converting any data. As the offset of a column chunk depends on the sizes of all preceding column chunks, we need to be able to accurately predict the size of each column chunk from the metadata alone. However, this is not possible for general-purpose compression schemes, as their compression ratio depends on the exact data distribution. The same reasoning prohibits the use of encoding schemes where the size of the resulting data can not be predicted from metadata. Thus, our on-the-fly conversion does not use compression or most encoding schemes, resulting in the virtualized Parquet file being much larger than its native counterparts. Due to the increase in network bandwidth in public clouds, transferring data is no longer the main bottleneck, making this approach viable. Figure 4 illustrates the mapping from data in the BtrBlocks format into the right position in the Parquet file format.

Calculating metadata byte size. Parquet row groups include only minimal metadata : data page headers, repetition levels for null values, and definition levels for nested structures. Data page headers use variable-length integer encoding via the Thrift TCompactProtocol's uleb128 scheme, causing their size to be variable. However, the byte size of an uleb128-encoded integer n can be efficiently computed as $\lceil \frac{log_2(n)}{7} \rceil$. For each page, we need to apply this to just two values: the number of values and the (un-)compressed size.

Exploiting dictionary encoding. Dictionary encoding is a powerful encoding supported by both Parquet and BtrBlocks. It eliminates duplicated values and replaces their occurrences with indices into a list of unique values. As the indices are fixed-size, the resulting size of the encoded data can easily be predicted when the total length of the unique values is known. Decompressing dictionary-encoded data is especially wasteful as even in-memory representations (i.e. Arrow, DuckDB) support dictionary encoding.

Bitpacked dictionary indices. However, Parquet does not store dictionary indices as plain values. Instead, it encodes them using a hybrid of run-length encoding (RLE) and bit-packing. Since RLE size depends on the value distribution, it is unsuitable for our optimization, where predictable sizes are required. Bit-packing, on the other hand, is an encoding with predictable size.

4.3 Evaluation

Setup. Our prototype Active Data Lake implementation emulates the S3 storage API [65]. We use DuckDB v.1.2.1 as a query engine and use its native support for Parquet and querying S3 to access native and virtualized Parquet files from our server. We conduct experiments in two experimental setups: In the first, we want to isolate conversion performance. For this purpose, the HTTP server has the requested data files already loaded in memory. In the second setup, we simulate the common case where data resides in S3 and must be fetched by the Active Data Lake before answering range requests from the DuckDB client. As data, we use the *lineitem* table from TPC-H at scale factor 10. Both DuckDB and the HTTP server are running on separate c5n.9xlarge instances within the same availability zone on AWS with 50Gib/s network bandwidth.

Full table scan. Our first experiment compares the end-to-end full table scan performance between our virtualized Parquet file and general-purpose compression schemes supported in the Arrow Parquet library. To give a picture of total workload, we include the on-disk size of the data using the different techniques as a second axis for Figure 5. Due to virtualization, the visualized on-disk size for BtrBlocks does not correspond to the amount of data that needs to be transferred between Active Data Lake and the query engine. While the BtrBlocks *lineitem* data is only 1.58 GiB, 4.3 GiB needs to be transferred with dictionary encoding enabled, and 6 GiB without. When data is already loaded in the Active Data Lake's memory, there is a noticeable virtualization overhead of 22% compared to the similarly compact Zstd-compressed Parquet file. Brotli-compressed Parquet, the only configuration more compact than the BtrBlocks representation, is 17% slower than accessing the virtualized file. To show the potential benefits of newer file formats, we have also implemented native support for BtrBlocks in DuckDB as an extension. Natively reading the format results in a 2.26x speed-up compared to the Zstd-compressed Parquet file. When data resides on S3, the differences between general-purpose compression schemes shrink as more heavyweight compression schemes benefit from their higher compression ratio, reducing the amount of data that needs to be fetched from S3. The virtualized Parquet file is the second fastest option, only 1.3% slower than the best native Parquet setting, Zstd, which strikes the best balance between decompression speed and the amount of transferred data.

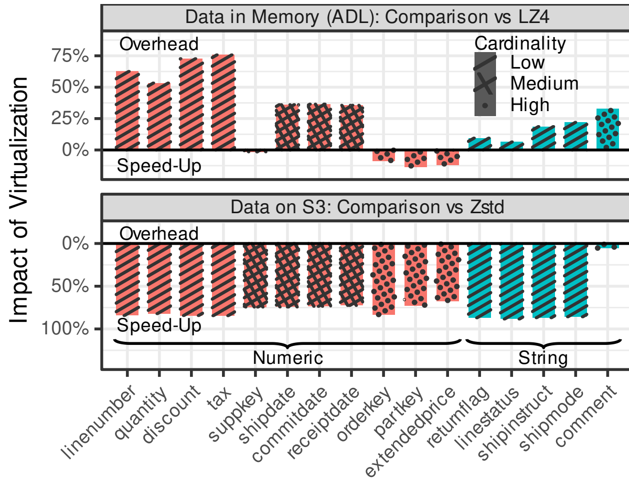

Figure 6: Overhead of virtualization (*lineitem* columns at scale-factor 10, 1 thread).

While the Zstd compressed Parquet file is roughly as large as the BtrBlocks files, the overhead of virtualization shrinks. The reason is that BtrBlocks files are sized at 16 MB, an efficient size for S3 objects, whereas Zstd makes requests that are 27 MB on average.

Single column scan. To verify the performance of native projections, we have measured the latency of single, full column scans for all Parquet file types and all columns present in the *lineitem* table. We observe that the average speed-up/slowdown remains extremely similar to those measured for the full table scan. Thus, our goal of enabling effective and transparent projections through the query engine has been reached. As there is a significant difference between columns using the same compression in relative performance, we investigate the reasons for this variation in Figure 6. We compare the relative performance of our virtualized column scan to the fastest compression scheme in each setup: When data is in memory, we compare to LZ4, when data needs to be fetched from S3, we compare to Zstd. We categorize the columns by their cardinality as well as data type. For data residing in memory, the virtualized file exceeds native performance in the case of high-cardinality integer columns, since Parquet does not have any helpful encodings in this case. For every other column type, the virtualization approach has some overhead. We observe the biggest overhead (70%) for low-cardinality (less than 100 distinct values) integer columns for which Parquet preserves dictionary encoding while BtrBlocks decompresses to a flat array, resulting in more data being transmitted. File formats that can preserve dictionary encodings for numeric types should not suffer the same penalties. Note that without preserving dictionary compression for strings, we observe a 10x slowdown in the case of low-cardinality string columns. When data needs to be fetched from S3 first, our virtualization approach shines. For single column scans, BtrBlocks has a superior access pattern compared to Parquet. While Parquet needs to issue one request per column-chunk, BtrBlocks groups chunks of the same column together into 16 MB part files. For columns where each value is only a few bytes, this reduces the number of necessary S3 requests dramatically. For instance, for the single-character *linestatus* column with only 4 distinct values, the number of necessary requests is only 3 compared to 60 for the Parquet file, resulting in a speed-up of 88% compared to the native Parquet file. Other columns achieve similar speed-up, for the only high-cardinality string column, which has merely a 5.6% speed-up needs 44 requests.

Native predicate pushdown. In the last experiment, we use the query engine's native pruning ability on min/max statistics by introducing a *less* predicate on the sorted *orderkey* column of varying selectivity. As expected, both transferred data and execution time scale nearly perfectly with the rowgroup-level selectivity.

4.4 Discussion

Our experiments show that virtualizing existing file formats can work completely transparently to the query engine interacting with the Active Data Lake maintaining query engines native projection and pruning capabilities. The overhead of having to translate between file formats is moderate, with some column categories even being able to outperform native performance.

Caching. To save computation resources, it may be helpful to pre-compute and cache frequently accessed parts of the data in the target format. As the size of the cached data is known after initial computation, the restriction to predictable encodings is lifted, thereby enabling data to be cached in more compact formats.

Other formats. Our approach generalizes to other file formats as long as they fulfill the following requirements:

(1) Source and target formats have horizontal partitions.
(2) The target format has (some) predictable encodings.
(3) The source's metadata allows predicting the uncompressed size.

5 BENEFIT 2: PREFETCHING

Motivation. The Active Data Lake abstracts the physical location of the data and allows engines fast access, even if they do not support the chosen storage device. In the long term, this ability allows the integration of computational storage or cross-engine caching. In this section, we show that the Active Data Lake can help achieve good performance even though the engine does not optimize for the physical storage medium. We do this by accelerating DuckDB's performance for data stored on S3 by implementing server-side prefetching. While DuckDB already natively supports querying data in cloud object storage, it is not optimized for the characteristics of object stores and always uses synchronous I/O.

Cloud object storage. Network optimized virtual machines can achieve a bandwidth of up to 600 Gbit/s reading from cloud object storage. However, individual requests have a significant latency ranging from tens to hundreds of milliseconds. As a result, many requests have to be made concurrently to satisfy the achievable bandwidth. Durner et al. [24] have shown that requests in the range of 8-16 MiB are the most cost-effective for analytical workloads.

DuckDB and remote storage. DuckDB uses synchronous I/O operations exclusively. As a result, worker threads sit idle while data is fetched from cloud object storage. The recommended workaround is to increase the number of worker threads, resulting in a higher number of concurrent requests. While this approach improves performance for I/O-heavy queries, the OS context switching between threads may reduce performance for compute-intensive queries.

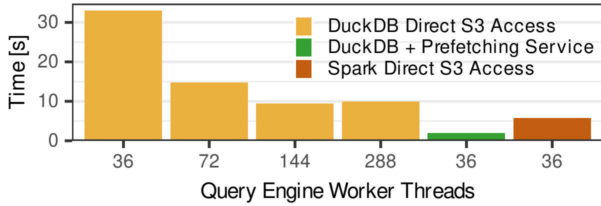

Figure 7: DuckDB scan performance using S3 directly or prefetching service (*lineitem*, SF=50, native DuckDB format).

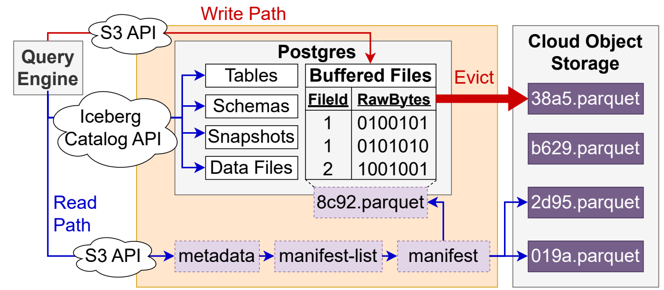

Figure 8: Read and Write path of the Active Data Lake with small file buffering in PostgreSQL.

Prefetching service. We implement a prefetching service that acts as a transparent proxy between DuckDB and object storage to accelerate the performance over remote native database files. When DuckDB sends the first head request to figure out the size of the file, the prefetching service starts downloading the entire file using fixed-size requests of 8 MiB. Individual chunks are buffered in memory and used to answer subsequent range requests from DuckDB. Our implementation uses the AWS C++ SDK to issue requests to S3. In particular, it uses the S3Crt client for asynchronous requests.

Experimental setup. To evaluate the performance of our implementation, we measure the end-to-end query latency for a full table scan over the *lineitem* table at scale factor 50, stored in the native DuckDB format on AWS S3. Both DuckDB v1.2.1 and our prefetching service are running on separate network-optimized c5n.9xlarge instances in the same availability zone. Additionally, we compare our performance against Apache Spark 4.0.1 directly reading a Parquet file from S3. Before running the benchmark, we issue additional requests to ensure the accessed object is hot.

Results. Figure 7 shows the query latencies. While native DuckDB, using the default number of 36 worker threads (hardware threads on c5n.9xlarge), takes 33 seconds to execute the full table scan, performance improves by 17× when using the prefetching service. While increasing the number of DuckDB worker threads to 144 decreases execution time to 9.47 seconds, this ideal configuration is still 4.87 times slower than using the prefetching service. The additional measurement of Apache Spark directly reading Parquet from S3 shows that our prefetching service is more than capable of making DuckDB competitive with engines that have native support for asynchronous, high-bandwidth reads.

6 BENEFIT 3: FAST TRICKLING INSERTS

Motivation. Many modern data-intensive applications require the ingestion of continuously arriving data with low latency. While Apache Iceberg is a widely adopted destination for such data due to its compatibility with analytical tools, its commit protocol is not suited for high-frequency writes. Each commit involves creating multiple metadata files and coordination with a catalog, resulting in significant overhead. Thus, ingestion systems often buffer data and commit in batches, introducing a trade-off between data freshness and write efficiency. In this section, we show how the Active Data Lake removes the need to batch transactions, enabling high-throughput ingestion while preserving compatibility with Iceberg and Parquet ecosystems.

6.1 Background

Commits in Apache Iceberg. For every commit in Apache Iceberg, the query engine needs to write at least three files: The added rows are written into one or more data files, which in turn are tracked in a manifest file. Additionally, a manifest-list file tracks all the manifest files contained in the snapshot. After writing, the query engine reports the changes to the metadata to the catalog in JSON format. The catalog then applies those changes to the current metadata and writes out a new version to cloud object storage. After the write operation, the catalog checks that the current metadata version is still the version expected by the query engine, meaning that no other transaction has occured in the meantime. Thus, any transactions on the same table conflict with each other.

Theoretical upper bound. This limitation puts an upper limit on the achievable transaction throughput even with infinite workers and logically independent transactions to $\frac{1}{\text{storage latency}}$. Assuming a latency of 100ms for S3, this fundamentally limits throughput to 10 transactions a second even ignoring practical factors such as having to compute and materialize the new snapshot first.

Small file problem. Even with the discussed limitations, commits are still too frequent to produce optimized Parquet files, resulting in many small files and unnecessary metadata bloat. Both factors reduce read performance. The small file problem is addressed by background services that periodically merge multiple small files into a single optimized Parquet file.

6.2 Implementation

Our prototype supports insert and delete operations with low latency. The effect of these operations is visible immediately to analytical query engines. In contrast, for existing systems that offer low-latency inserts, such as Databricks Zerobus [19], inserted data only becomes queryable after it is flushed into OTF format. Additionally, the prototype eliminates conflicts between independent append operations and requires no background services. Figure 8 illustrates how both write and read operations work.

Write path. Both insert and delete operations result in a single data file being uploaded to our prototype. Internally, we buffer small files in PostgreSQL as binary objects until enough data has accumulated to produce optimized Parquet files, which are written asynchronously to cloud object storage. Buffering data in a relational database provides durability guarantees. In case stronger guarantees are necessary, a distributed database can be used at the

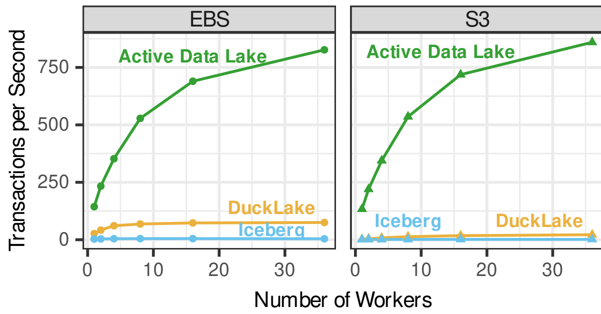

Figure 9: Transaction throughput of data lake architectures (single row operations, 95% inserts and 5% deletes).

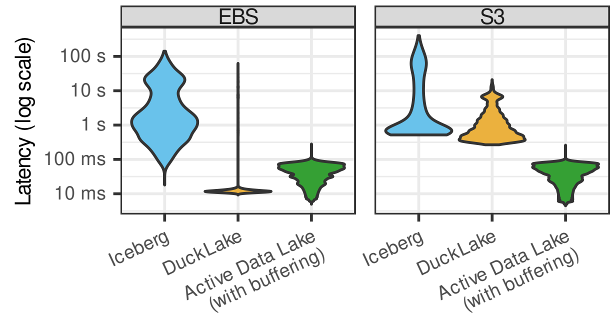

Figure 10: Latency of transactions of data lake architectures (36 clients, single row operations, 95% inserts and 5% deletes).

cost of higher commit latency. Each commit also stores metadata in PostgreSQL relational tables to track the table history. Deletes are handled similarly. Our implementation uses the Iceberg concept of equality deletes. Following the specification, a delete file is created in Parquet format, which stores all column values that identify deleted rows. Just like data files, delete files are buffered in PostgreSQL until enough rows have accumulated to write an optimized Parquet file. Data and delete files are then reconciled at read time by the query engine.

Read path. Reads follow the Iceberg specification, but metadata files are virtualized on demand from the table history tracked in PostgreSQL and are not materialized on object storage. Data and delete files are read either from object storage (if already flushed) or directly from PostgreSQL, enabling immediate access after ingestion. All reads go through the S3 API for full compatibility.

6.3 Evaluation

We compare the transaction throughput of our Active Data Lake prototype with Iceberg and DuckLake. Our benchmark represents an extreme scenario where multiple clients concurrently try to apply single-row modifications to the same table. 95% of operations insert a new row and 5% delet an existing row. We use PyIceberg [38] as a lightweight writer for Iceberg and configure DuckLake to use PostgreSQL as the relational database. To show that the issues with Iceberg's commit protocol are not exclusively due to the latency of cloud object storage, we run the experiment both on elastic block storage and on cloud object storage.

EBS. We run the experiment on a single c5n.9xlarge machine on AWS. The machine runs both the client and the catalog, respectively the Active Data Lake. Primary storage for (meta-)data is the attached EBS volume. Figure 9 shows the throughput scalability for all approaches. With a single client, our prototype achieves a throughput of around 144 transactions per second (tps). Meanwhile, DuckLake manages 26 tps, and Iceberg achieves only 3 tps due to the number of files written and overhead in the PyIceberg library. At 8 writers, DuckLake improves to 68 tps but is close to the limit of its scalability, reaching only 75 tps at 36 writers. Iceberg barely increases its throughput when adding more writers, achieving 4 tps at 36 writers since all transactions on the same table result in a conflict. DuckLake conceptually manages to isolate independent appends, yet keeps track of table-level statistics in a single row, which all write operations need to update. The Active Data Lake avoids conflicts altogether and scales linearly until the limit of provisioned IOPs is reached. At 36 writers, it manages to perform 827 tps, 11 times the throughput of DuckLake and 200 times the throughput of Iceberg. Figure 10 (left) shows the latency of individual transactions. The Active Data Lake experiences a higher base latency as all data is sent twice over TCP even on a single machine, first from client to the Active Data Lake, then to PostgreSQL. DuckLake employs an exponential backoff strategy when encountering transaction conflicts, leading to measured times of up to a minute between transactions. In the event of conflicts, Iceberg gives an advantage to the conflict winner: While losers need to first gather up-to-date metadata before attempting the next commit, the winner gets the current information as part of the success response. Thus, there are many cases of Iceberg clients not managing a single successful transaction during the measurement period. Latency is not reported in these cases. A similar problem exists for deletes in both Iceberg and DuckLake. As deletes are slower than inserts, only 0.8% of successful operations in DuckLake are delete operations at 36 writers. For Iceberg, all delete operations fail when using more than 2 writers.

S3. We run the same experiment for remote storage in the form of S3 using two c5n.9xlarge machines within the same availability zone in AWS. One machine runs the clients while the other runs the catalog, respectively the Active Data Lake. Primary data storage is S3. The Active Data Lake manages to avoid the latency of S3 for small append operations through its internal buffering, maintaining 133 tps with a single writer while both DuckLake and Iceberg manage only 2.7 and 0.8 tps, respectively. The increased latency for Iceberg and DuckLake can also be observed in Figure 10 (right). Due to the lower transaction rate, DuckLake experiences fewer transaction conflicts and thus does not use backoff as aggressively, leading to a lower maximum observed latency. DuckLake manages to scale better with the number of writers due to the increased latency of producing data files compared to the conflict period of updating a single row in PostgreSQL. At 36 writers, DuckLake achieves 22 tps, still well below the performance in a local setting. Iceberg only manages to scale to 1.6 tps at 36 writers. The Active Data Lake manages to scale throughput just as effectively as in

the local storage experiment, managing 860 tps at 36 writers. At 16 and 36 workers, the remote setting even manages to overtake the throughput of the local setting, presumably due to the fact that compute resources are no longer shared between client and Active Data Lake.

Discussion. Our experiments show that current approaches have clear limitations in terms of throughput. Our prototype demonstrates that, even when adhering to the interface of OTFs and using cloud object storage as the primary storage location, transaction throughput can be significantly increased.

7 DEPLOYMENT AND COST

The presented optimizations have shown that the Active Data Lake can improve performance. However, as an additional server, the Active Data Lake is associated with some cost. In this section, we want to take a closer look at the implications of running an Active Data Lake on the total cost. To do this, we first need to discuss the different ways to deploy and operate an Active Data Lake before finally demonstrating through a cost model that the Active Data Lake architecture can result in cost savings.

7.1 Deployment Modes

Single tenant cluster. Each organization can deploy and manage its own cluster of Active Data Lake nodes. This approach allows organizations to have full control over the Active Data Lake and their data, making it an attractive option in cases with strict privacy requirements. However, because the cluster needs to be sized to handle sudden workload spikes, the approach can result in low resource utilization.

Function-as-a-Service based architecture. To address the issue of resource utilization, one could turn to the Function-as-a-Service offerings such as Amazon Lambda. In this design, for each range request of a virtualized file to the Active Data Lake, a function is spawned. This function determines which data needs to be read, actually fetches the data, and transcodes it into the target format, before returning the result to the caller. While this leads to perfect resource utilization, Lambda functions do not have permanent state, which prevents some optimizations such as caching.

Multi-tenant service. Another way of increasing resource utilization is using multi-tenancy to absorb workload spikes of individual users. This architecture enables all optimizations and has good resource utilization, although it is non-trivial to implement.

7.2 Cost Model

We believe that a multi-tenant system is the most promising deployment mode for the Active Data Lake. In this section, we demonstrate that the Active Data Lake can reduce the total workload cost compared to direct file access using a cost model. The model assumes a multi-tenant architecture that achieves a resource utilization of 80% and includes two optimizations: file virtualization and workload-adaptive caching.

File virtualization. File virtualization enables the use of additional encoding and compression methods, resulting in more compact files and reduced storage costs. The downside is that computational power is required upon access to transform the compressed file back into its original state. We assume that each thread is capable of

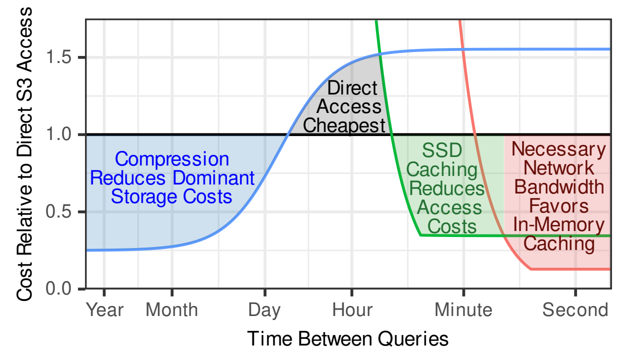

Figure 11: Relative cost of Active Data Lake optimizations compared to directly accessing data on S3 (1 MB objects).

decoding 27 MB/s, which can even be achieved by general-purpose compression schemes (depending on the dataset) and that the compressed file is four times more compact. As a result, storage costs can be reduced by a factor of 4, but computational power is needed to transform data from one format to another. Based on these assumptions, c6g instances provide the most cost-efficient data access.

Workload-Adaptive Caching. We model a workload-adaptive caching method that tracks access frequency for each object and decides whether caching can reduce workload cost. This approach differs from traditional caching methods in that it does not have a fixed size and needs to decide which objects to keep, instead it scales the capacity to fit all objects that are worth caching. Our model includes both SSD-based caching (m6idn instances) and in-memory caching (c8gn instances). As we do not model the effects of reduced access latency, the benefit of in-memory caching comes from the fact that the instances with the most network bandwidth per dollar do not come with SSDs.

Result discussion. Figure 11 shows the results of our model for a workload consisting of 1 MB objects. If objects are not accessed frequently, storage costs dominate the total workload cost and compressing objects can thus reduce costs significantly. However, if the time between accesses to the same object is between 3.5 hours and 15 minutes, none of the presented optimizations is cheaper than reading the object directly from S3. The additional cost can reach up to 50% of the original cost. If accesses occur at least every 15 minutes, caching begins to pay off. Initially, it is cheapest to cache on instance SSDs as the available storage is the bottleneck. This shifts to network bandwidth as the bottleneck when access frequency increases to one access every 5 minutes. Starting at 4 requests a minute, the cheapest option is to cache in-memory on network-optimized c8gn instances with 600 Gbit/s network bandwidth. This results in cost reductions of 70%.

Conclusion. No single optimization always pays off, but most of the time there is an optimization that can reduce the total cost compared to simply accessing objects directly from S3. An Active Data Lake can implement multiple optimizations and use workload statistics to apply the most economical one. Despite the overhead of an additional running service, this cost model shows that the Active Data Lake can reduce total costs in many workloads.

8 DISCUSSION AND FUTURE WORK

We introduce the Active Data Lake, an architecture that provides both interoperability and physical data independence. By decoupling the interoperability interfaces from the physical data layout, it overcomes the innovation challenges at the file format, the access path, and the storage levels.

8.1 Addressing potential concerns

The Lakehouse paper briefly discusses the alternative approach of building a massively parallel serving layer [82] similar to the Active Data Lake. However, the authors dismiss the idea, calling it a step down from raw access to cloud object storage in terms of cost, bandwidth, availability, and lock-in. Let us address these concerns.
Cost. Section 7.2 shows that the Active Data Lake architecture is cheaper than directly accessing object storage for many workloads.
Bandwidth. The high bandwidth of cloud object storage is the result of good engineering and can therefore be matched by an efficient Active Data Lake implementation. Furthermore, the Active Data Lake architecture enables further optimizations for reducing the transferred data, such as pre-filtering and partial aggregation.
Availability. The central positioning of the Active Data Lake replaces the catalog in current lakehouse architectures as a single point of failure. However this is very similar for Iceberg catalogs. Even ignoring the issue of discoverability, any downtime of the catalog means (for Iceberg) the loss of ACID transactions and thus a high risk of data corruption if write operations continue. Thus, write operations should be avoided, a significant restriction, as write operations constitute around 40% of all queries [81]. If governance is enforced through catalog credential vending, even read queries are not possible without a running catalog.
Scalability. Compared to cloud data warehouses, the Active Data Lake only handles a subset of operations. To be precise, it only transcodes, filters, and pre-aggregates data. All of these operations are embarrassingly parallel. In fact, some cloud data warehouses use dynamically-sized clusters for these operations, despite using fixed-size clusters in general [70].
Lock-in. Given the added functionality of Active Data Lakes, one might ask whether this comes at the cost of vendor lock-in. However, by leveraging virtualization, users can easily export both data and metadata to their own storage in standard open-source formats.

8.2 Drawbacks of Library-Based Data Lakes

Data lake and lakehouse architectures depend on libraries embedded in every client to lower the implementation burden. This raises the question of whether the benefits of the Active Data Lake can be achieved through a library rather than an additional server.
The upgrade problem. When a library introduces additional functionality in a new version, the added features must either be backwards compatible, so that older versions can still understand the data produced by newer versions, or the entire ecosystem must atomically upgrade to the new version of the library. In an ecosystem as diverse as the data lake, atomically switching all clients to new library versions simultaneously is not feasible. Features that are not backwards compatible, such as the introduction of new encodings in Parquet, therefore suffer from limited adoption [48].
Inherent limitations. Some capabilities of the Active Data Lake cannot be realized through client-side libraries. For instance, global workload-driven optimizations are impossible as clients continue to observe only a part of the total workload. Similarly, governance and specifically fine-grained access control cannot be securely enforced on the client side. Even if the library masks sensitive data before passing it to the application, the fact that sensitive data has been transferred to the client opens up additional attack vectors.

8.3 Layout-Oblivious Interfaces

File-based interfaces ensure compatibility with the existing data lake ecosystem. The optimizations in this paper demonstrate that the Active Data Lake can provide significant benefits even when maintaining the interfaces of Parquet and Iceberg. Embracing an additional interface tailored to the Active Data Lake enables even more optimizations. For instance, row-level filtering is not as practical in file-based interfaces, as the Active Data Lake needs to define the number of files and their respective sizes early. An interface that does not rely on the layout of the data into files would remove this requirement and thus improve performance.
Existing layout-oblivious interfaces. Multiple such interfaces already exist: Arrow Flight [56] allows the transfer of Arrow record batches, the Big Query Storage API [10, 53] allows extracting data as Arrow or Avro streams from Google's proprietary storage, and Spark Connect [72] allows transferring data between Spark clusters. The number of distinct interfaces is growing as more protocols (i.e. [54]) are being suggested, but no standard has yet emerged.
Interoperability challenges. In order to minimize network traffic, information about projections, predicates, and pre-aggregations needs to be transferred to the Active Data Lake. Current solutions either send a SQL query string, suffering from distinct dialects, or custom JSON, which fragments the ecosystem. Coral [57] and SQLGlot [55] try to convert between SQL dialects, while Substrait [77] defines a serialization standard for query plans. However, neither addresses the issue that different systems might have different semantics for the same operator. A recent academic proposal is to move the abstraction level from operators to suboperators [7, 47] for more flexibility. This could not only bridge the semantic gap between systems, but also expand the interface of the Active Data Lake beyond SQL.

8.4 Future Work

In this paper, we examined three optimizations separately and showed that each individually can provide significant benefits. In the future, we want to evaluate additional, more complex, optimizations and build a distributed Active Data Lake that addresses the challenges of scalability and availability.

ACKNOWLEDGMENTS

Funded/Co-funded by the European Union (ERC, CODAC, 101041375). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

REFERENCES

[1] Azim Afroozeh and Peter Boncz. 2023. The FastLanes Compression Layout: Decoding >100 Billion Integers per Second with Scalar Code. *Proc. VLDB Endow.* 16, 9 (2023), 2132–2144.

[2] Azim Afroozeh and Peter Boncz. 2025. The FastLanes File Format. *Proc. VLDB Endow.* 18, 11 (2025), 4629–4643.

[3] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance. In *VLDB*. Morgan Kaufmann, 169–180.

[4] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (2020), 3411–3424.

[5] AWS. 2025. Amazon S3 Glacier storage classes. https://aws.amazon.com/s3/storage-classes/glacier Accessed July 31, 2025.

[6] Azure. 2021. List Blobs. https://learn.microsoft.com/en-us/rest/api/storageservices/list-blobs?tabs=microsoft-entra-id Accessed July 31, 2025.

[7] Maximilian Bandle and Jana Giceva. 2021. Database Technology for the Masses: Sub-Operators as First-Class Entities. *Proc. VLDB Endow.* 14, 11 (2021), 2483–2490.

[8] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *SIGMOD Conference*. ACM, 221–230.

[9] Google BigQuery. 2025. Create BigLake external tables for Apache Iceberg. https://cloud.google.com/bigquery/docs/iceberg-external-tables Accessed July 31, 2025.

[10] Google Cloud BigQuery. 2024. BigQuery Storage API. https://cloud.google.com/bigquery/docs/reference/storage/rpc Accessed July 31, 2025.

[11] Peter Boncz, Yannis Chronis, Jan Finis, Stefan Halfpap, Viktor Leis, Thomas Neumann, Anisoara Nica, Caetano Sauer, Knut Stolze, and Marcin Zukowski. 2023. SPA: Economical and Workload-Driven Indexing for Data Analytics in the Cloud. In *ICDE*. IEEE, 3740–3746.

[12] Ramesh Chandra, Haogang Chen, Ray Matharu, Sarah Cai, Jeff Chen, Priyam Dutta, Bogdan Ghita, Todd Greenstein, Gopal Holla, Peng Huang, Yuchen Huo, Adrian Ionescu, Adriana Ispas, Tim Januschowski, Vihang Karajgaonkar, Stefania Leone, David Lewis, Andrew Li, Nong Li, Cheng Lian, Stephen Link, Qing Lu, Yesheng Ma, Chris Pettitt, Vijayan Prabhakaran, Bogdan Raducanu, Kyle Rong, Paul Roome, Samarth Shetty, Sean Smith, Xiaotong Sun, Yuyuan Tang, Weitao Wen, Lei Xia, Junlin Zeng, Ben Zhang, Reynold Xin, and Matei Zaharia. 2025. Unity Catalog: Open and Universal Governance for the Lakehouse and Beyond. In *SIGMOD Conference Companion*. ACM, 310–322.

[13] EF Codd. 1985. How Relational is your Database Management System? *Computer World* (1985). https://reldb.org/c/index.php/twelve-rules/ Accessed February 11.

[14] Apache Iceberg contributors. 2024. Add Scan Planning Endpoints to open api spec. https://github.com/apache/iceberg/pull/9695 Accessed July 31, 2025.

[15] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD Conference*. ACM, 215–226.

[16] Databricks. 2024. Unity Catalog. https://www.unitycatalog.io/ Accessed February 11, 2026.

[17] Databricks. 2025. What is a view? https://docs.databricks.com/aws/en/views/ Accessed July 31, 2025.

[18] Databricks. 2025. What is Delta Lake? https://docs.databricks.com/en/delta/index.html# Accessed July 31, 2025.

[19] Databricks. 2025. Zerobus Ingest connector overview. https://docs.databricks.com/aws/en/ingestion/zerobus-overview Accessed: December 1, 2025.

[20] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*. USENIX Association, 137–150.

[21] Dremio. 2025. Dremio Reflections. https://www.dremio.com/wiki/dremio-reflections/ Accessed July 31, 2025.

[22] DuckDB. 2025. Delta Extension. https://duckdb.org/docs/extensions/delta.html Accessed July 31, 2025.

[23] Vino Duraisamy. 2023. Iceberg Tables on Snowflake: Design considerations and Life of an INSERT query. https://vinodhini-sd.medium.com/iceberg-tables-on-snowflake-design-considerations-and-life-of-an-insert-query-5026ea10bbea Accessed July 31, 2025.

[24] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *Proc. VLDB Endow.* 16, 11 (2023), 2769–2782.

[25] Microsoft Fabric. 2025. Lakehouse and Delta Lake tables. https://learn.microsoft.com/en-us/fabric/data-engineering/lakehouse-and-delta-tables Accessed July 31, 2025.

[26] Facebook. 2025. Nimble. https://github.com/facebookincubator/nimble Accessed July 31, 2025.

[27] The Apache Software Foundation. 2024. Apache Hadoop. https://hadoop.apache.org/ Accessed July 31, 2025.

[28] The Apache Software Foundation. 2024. Apache Hudi. https://hudi.apache.org/ Accessed February 11, 2026.

[29] The Apache Software Foundation. 2024. Apache Iceberg REST Catalog API. https://github.com/apache/iceberg/blob/main/open-api/rest-catalog-open-api.yaml Accessed July 31, 2025.

[30] The Apache Software Foundation. 2025. Apache Arrow. https://arrow.apache.org/ Accessed July 31, 2025.

[31] The Apache Software Foundation. 2025. Apache Iceberg. https://iceberg.apache.org Accessed July 31, 2025.

[32] The Apache Software Foundation. 2025. Apache ORC. https://orc.apache.org/ Accessed July 31, 2025.

[33] The Apache Software Foundation. 2025. Apache Parquet. https://parquet.apache.org/ Accessed July 31, 2025.

[34] The Apache Software Foundation. 2025. Iceberg View Spec: Representations. https://iceberg.apache.org/view-spec/#representations Accessed July 31, 2025.

[35] The Apache Software Foundation. 2025. Indexes | Apache Hudi. https://hudi.apache.org/docs/indexes/ Accessed July 31, 2025.

[36] The Apache Software Foundation. 2025. Lake Cache | Apache Hudi. https://hudi.apache.org/blog/2021/07/21/streaming-data-lake-platform/#lake-cache Accessed July 31, 2025.

[37] The Apache Software Foundation. 2025. Polaris (incubating). https://polaris.apache.org/ Accessed July 31, 2025.

[38] The Apache Software Foundation. 2025. PyIceberg. https://py.iceberg.apache.org/ Accessed July 31, 2025.

[39] Mateusz Gienieczko, Maximilian Kuschewski, Thomas Neumann, Viktor Leis, and Jana Giceva. 2025. AnyBlox: A Framework for Self-Decoding Datasets. *Proc. VLDB Endow.* 18, 11 (2025), 4017–4031.

[40] Tobias Götz, Daniel Ritter, and Jana Giceva. 2025. LakeVilla: Multi-Table Transactions for Lakehouses. *CoRR* abs/2504.20768 (2025).

[41] Martin Grund, Stefania Leone, Herman Van Hövell, Sven Wagner-Boysen, Sebastian Hillig, Hyukjin Kwon, David Lewis, Jakob Mund, Polo-Francois Poli, Lionel Montrieux, Othon Crelier, Xiao Li, Reynold Xin, Matei Zaharia, Michalis Petropoulos, and Thanos Papathanasiou. 2025. Databricks Lakeguard: Supporting Fine-grained Access Control and Multi-user Capabilities for Apache Spark Workloads. In *SIGMOD Conference Companion*. ACM, 418–430.

[42] Apache Hudi. 2024. RFC-73: Multi-Table Transactions. https://apache.googlesource.com/hudi/+/refs/heads/release-1.0.0-beta1/rfc/rfc-73/rfc-73.md Accessed February 11, 2026.

[43] Ibis-Project. 2025. Ibis. https://ibis-project.org/ Accessed July 31, 2025.

[44] Apache Iceberg. 2024. Multi-Table Transactions in Iceberg. https://docs.google.com/document/d/1UxXifU8iqP_byaW4E2RuKZx1nobxmAvc5urVcWas1B8 Accessed February 11, 2026.

[45] Apache Iceberg. 2025. Flink Connector. https://iceberg.apache.org/docs/latest/flink-connector/ Accessed July 31, 2025.

[46] Prakhar Jain. 2024. [Protocol Change Request] Delta Coordinated Commits. https://github.com/delta-io/delta/issues/2598 Accessed February 11, 2026.

[47] Michael Jungmair and Jana Giceva. 2023. Declarative Sub-Operators for Universal Data Processing. *Proc. VLDB Endow.* 16, 11 (2023), 3461–3474.

[48] Laurens Kuipers. 2025. Query Engines: Gatekeepers of the Parquet File Format. https://duckdb.org/2025/01/22/parquet-encodings.html Accessed July 31, 2025.

[49] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2 (2023), 118:1–118:26.

[50] Delta Lake. 2025. Apache Spark connector. https://docs.delta.io/latest/delta-spark.html Accessed July 31, 2025.

[51] Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *SIGMOD Conference Companion*. ACM, 5–17.

[52] LanceDB. 2025. Lance. https://github.com/lancedb/lance Accessed July 31, 2025.

[53] Justin J. Levandoski, Garrett Casto, Mingge Deng, Rushabh Desai, Pavan Edara, Thibaud Hottelier, Amir Hormati, Anoop Johnson, Jeff Johnson, Dawid Kurzyniec, Sam McVeety, Prem Ramanathan, Gaurav Saxena, Vidya Shanmugan, and Yuri Volobuev. 2024. BigLake: BigQuery's Evolution toward a Multi-Cloud Lakehouse. In *SIGMOD Conference Companion*. ACM, 334–346.

[54] Roman Leventov. 2024. Table transfer protocols: improved Arrow Flight and alternative to Iceberg. https://engineeringideas.substack.com/p/table-transfer-protocols-improved Accessed July 31, 2025.

[55] Toby Mao. 2024. SQLGlot. https://github.com/tobymao/sqlglot Accessed July 31, 2025.

[56] Wes McKinney. 2019. Introducing Apache Arrow Flight: A Framework for Fast Data Transport. https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight/ Accessed July 31, 2025.

[57] Walaa Eldin Moustafa, Wenye Zhang, Sushant Raikar, Raymond Lama, Ron Hu, Shardul Mahadik, Laura Chen, Khai Tran, Chris Chen, and Nagarathnam Muthusamy. 2020. Coral: A SQL translation, analysis, and rewrite engine for

modern data lakehouses. https://www.linkedin.com/blog/engineering/archive/coral Accessed January 23, 2026.
[58] Project Nessie. 2025. Project Nessie: Transactional Catalog for Data Lakes. https://projectnessie.org/ Accessed July 31, 2025.
[59] Oracle. 2025. Query Apache Iceberg Tables. https://docs.oracle.com/en/cloud/paas/autonomous-database/serverless/adbsb/query-external-data-apache-iceberg.html Accessed September 16, 2025.
[60] Pedro Pedreira, Orri Erling, Maria Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *Proc. VLDB Endow.* 15, 12 (2022), 3372–3384.
[61] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satyanarayana R. Valluri, Mohamed Zaït, and Jacques Nadeau. 2023. The Composable Data Management System Manifesto. *Proc. VLDB Endow.* 16, 10 (2023), 2679–2685.
[62] Mark Raasveldt and Hannes Mühleisen. 2025. The DuckLake Manifesto: SQL as a Lakehouse Format. https://ducklake.select/manifesto/ Accessed July 31, 2025.
[63] Amazon Redshift. 2025. Using Apache Iceberg tables with Amazon Redshift. https://docs.aws.amazon.com/redshift/latest/dg/querying-iceberg.html Accessed July 31, 2025.
[64] Amazon Web Services. 2025. ListObjects. https://docs.aws.amazon.com/AmazonS3/latest/API/API_ListObjects.html Accessed July 31, 2025.
[65] Amazon Web Services. 2025. S3 API Reference. https://docs.aws.amazon.com/AmazonS3/latest/API/Type_API_Reference.html Accessed July 31, 2025.
[66] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *MSST*. IEEE Computer Society, 1–10.
[67] Preshant Singh, Russel Spitzer, and Vishwa Lakkundi. 2025. [OSS] Secure Views for dynamic policy enforcement. https://docs.google.com/document/d/1AJicez7xPhzwKXenGZ19h0hngxrwAg3rSajDV1v0x-s/edit?tab=t.0 Accessed July 31, 2025.
[68] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. 2007. Thrift : Scalable Cross-Language Services Implementation. https://thrift.apache.org/static/files/thrift-20070401.pdf Accessed February 11, 2026.
[69] Snowflake. 2025. Apache Iceberg tables. https://docs.snowflake.com/en/user-guide/tables-iceberg Accessed July 31, 2025.
[70] Snowflake. 2025. Using the Query Acceleration Service (QAS). https://docs.snowflake.com/en/user-guide/query-acceleration-service Accessed November 28, 2025.
[71] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. 2014. Orca: a modular query optimizer architecture for big data. In *SIGMOD Conference*. ACM, 337–348.
[72] Apache Spark. 2025. Spark Connect Overview. https://spark.apache.org/docs/latest/spark-connect-overview.html Accessed July 31, 2025.
[73] Spiral. 2025. Vortex. https://github.com/spiraldb/vortex Accessed July 31, 2025.
[74] Russel Spitzer, Yi Fang, and Steven Wu. 2025. Iceberg Single File Commits. https://docs.google.com/document/d/1k4x8utgh41Sn1tr98eynDKCWq035SV_f75rtNHcerVw/ Accessed July 31, 2025.
[75] Starburst. 2024. Starburst Cached Views. https://www.starburst.io/resources/starburst-cached-views/ Accessed July 31, 2025.
[76] Till Steinert, Maximilian Kuschewski, and Viktor Leis. 2026. Cloudspecs: Cloud Hardware Evolution Through the Looking Glass. In *CIDR*. www.cidrdb.org.
[77] substrait io. 2021. Substrait: Cross-Language Serialization for Relational Algebra. https://github.com/substrait-io/substrait. Accessed February 11, 2026.
[78] Apache Thrift. 2024. Thrift Compact protocol encoding. https://github.com/apache/thrift/blob/master/doc/specs/thrift-compact-protocol.md Accessed July 31, 2025.
[79] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 2010. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE*. IEEE Computer Society, 996–1005.
[80] Trino. 2025. IcebergConnector. https://trino.io/docs/current/connector/iceberg.html Accessed July 31, 2025.
[81] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proc. VLDB Endow.* 17, 11 (2024), 3694–3706.
[82] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *CIDR*. www.cidrdb.org.
[83] Xinyu Zeng, Ruijun Meng, Martin Prammer, Wes McKinney, Jignesh M. Patel, Andrew Pavlo, and Huanchen Zhang. 2025. F3: The Open-Source Data File Format for the Future. *Proc. ACM Manag. Data* 3, 4 (2025), 245:1–245:27.