Petabyte-Scale Row-Level Operations in Data Lakehouses

Anton Okolnychyi*
Apple
aokolnychyi@apache.org

Chao Sun*
Apple
sunchao@apache.org

Kazuyuki Tanimura
Apple
ktanimura@apple.com

Russell Spitzer*
Apple
russellspitzer@apache.org

Ryan Blue†
Tabular
blue@apache.org

Szehon Ho
Apple
szehon_ho@apple.com

Yufei Gu*
Apple
yufei@apache.org

Vishwanath Lakkundi
Apple
vishwa@apple.com

DB Tsai
Apple
dbtsai@apple.com

ABSTRACT

Data lakehouses combine the almost infinite scale and diverse tooling of a data lake with the reliability and functionality of a data warehouse. This paper presents extensions that enhance data lakehouses using Apache Iceberg and Apache Spark with performant petabyte-scale row-level operations. The framework is capable of handling both high-density and sparse modifications by either materializing changes at the file level during writes or producing equality and position deletes that are lazily merged with existing data during reads. The paper also outlines essential improvements in determining and applying row-level changes: eliminating expensive shuffles with storage-partitioned joins, minimizing write amplification with runtime filtering, and optimizing the layout of output data with adaptive writes. Our evaluation demonstrates the relative strengths and weaknesses of the various materialization strategies, highlighting the use cases that require each technique. We also show an order of magnitude improvement in performance after our enhancements.

1 INTRODUCTION

In the evolving landscape of data management, data lakes have become the prevalent solution for storing and analyzing massive datasets [29]. Technologies such as MapReduce [42] and Apache Hadoop [1] gained popularity because of the inability of traditional data warehouses to cope with the size and variety of generated data. Projects like Apache Hive [56][36] and Apache Spark [58][57][32] addressed the initial limitations of the Hadoop ecosystem by providing SQL support, superior performance, and a better user experience. Even so, early data lakes often prioritized scalability and fault tolerance over transactional guarantees and ease of use [28]. These trade-offs forced organizations to adopt both data lakes and data warehouses, requiring data replication, increasing costs, and creating data silos.

The rise of cloud computing revolutionized these earlier systems by providing almost infinite decoupled storage and compute [29]. Cloud object stores were able to manage unprecedented volumes of data but lacked features Hadoop relied on, like atomic renames and consistent list operations. This caused critical performance degradation and exacerbated correctness issues in data lakes. Data warehouses also needed to be redesigned for the cloud, enabling solutions like Snowflake [40], Redshift [47], and BigQuery [50]. These systems provided scalable and efficient alternatives to traditional data warehouses, albeit with some limitations. Like their predecessors, cloud data warehouses relied on proprietary formats and were not designed for sharing storage, preventing users from leveraging emerging technologies and confining analytics to the capabilities of a specific query engine.

Open source data lakes were limited by reliance on Hive as the de facto standard for managing tables. Hive stored a portion of the table metadata, such as schema and a list of directories representing partitions, in a central metastore while using the underlying storage's list operation to determine what files to read for a query. The decision to track partitions, instead of individual files, facilitated scalability but limited functionality [44]. Iceberg [8] was designed to replace the Hive table format and enable previously impossible functionality. Iceberg's design brought data lakes in line with established SQL behavior: it incorporated transactional guarantees, avoided correctness problems, and addressed performance issues in the cloud. Unlike Hive, Iceberg captured extensive metadata about each file and defined how to make atomic changes to table state. These architectural decisions enabled features like ACID transactions, reliable schema evolution, file skipping based on column statistics [46] [51], time travel, rollback, and hidden partitioning.

The advent of Iceberg and other table formats like Delta Lake [31] brought data warehouse features and guarantees to data lakes, establishing the data lakehouse architecture [59]. Data lakehouses combined the reliability and behavior of a data warehouse with the almost infinite scale and diverse tooling of a data lake. The growing popularity of this architecture can be attributed to its commitment to open standards and modularity, permitting concurrent access to the same table from different query engines without compromising ACID guarantees or performance.

The ability to modify tables upon ingestion is an integral part of modern analytics. Efficient row-level DELETE, UPDATE, and MERGE operations used to be an exclusive feature of data warehouses, making the migration to data lakehouses difficult. Use cases like change data capture, incremental materialized view refresh, and certain types of slowly changing dimensions often involve handling not just new records but also updates and deletes. A frequently used approach for handling row-level changes in data lakes was to replace entire table partitions, forcing data engineers to develop complex and error-prone data transformation pipelines. These pipelines involved querying affected table partitions, merging their state with incoming changes, and then overwriting the existing records with the results of these computations. This approach presented several fundamental problems. It often turned fairly simple business logic into complex queries with joins and aggregations, increasing the probability of bugs or unhandled edge cases. The need to rewrite entire partitions required processing large volumes of unmodified data even when minimal changes were being made, making such transformations profoundly inefficient and resource-intensive. As a result, data engineers frequently found themselves managing cluster workloads and optimizing shuffles. The process of replacing partitions was also inherently unsafe, as concurrent modifications could silently corrupt the table. Partition replacements often required a separate external locking mechanism that had to be honored by all writers but could not be enforced by the underlying metadata layer. Readers were required to participate in this locking scheme resulting in degraded query performance. Nearly all query engines chose to avoid this approach because of the added complexity and performance implications.

While neither Iceberg nor Spark initially supported row-level operations, they possessed a set of features key to adding such functionality. Iceberg provided a commit protocol based on serializable snapshot isolation [35], supported atomic file swaps, and had a mechanism to track diverse metadata structures for each version of the table. Spark could reliably handle tens of petabytes of data and supported advanced database features like adaptive execution and runtime filtering. These capabilities made Iceberg and Spark particularly well-suited for incorporating support for DELETE, UPDATE, and MERGE commands.

The primary contributions of this paper are the foundational design of Iceberg and extensions to Spark that enable efficient petabyte-scale row-level operations. Section 2 starts with an overview of Iceberg and its metadata layout. This section concerns work done both by the authors of this paper as well as contributions from others in the Iceberg community. It also discusses the requirements and constraints that drove the more recently added support for row-level operations. Section 3 presents different ways to encode changes in Iceberg, allowing the table format to handle both high-density and sparse updates in batch as well as streaming applications. We explain how Iceberg plans jobs, implements differential structures, resolves write conflicts, and performs table maintenance. Section 4 describes Spark enhancements required to support row-level operations and achieve the desired query performance. Section 5 covers evaluation setup, results, and analysis. Sections 6 and 7 outline future and related work.

2 BACKGROUND

2.1 Iceberg Metadata

The Iceberg project consists of a formal specification [10] defining the structure, behavior, and operations supported by the table format, and a set of libraries that implement the spec in different programming languages such as Java [9] and Python [12].

The Iceberg metadata layout (Figure 1) is a persistent tree data structure. The state of an Iceberg table is prescribed by a catalog mapping an identifier to a specific root metadata file. Modifications to the table, such as adding data files or updating properties, result in the creation of a new metadata tree under a new root metadata file. This new revision is committed via an atomic swap, updating the pointer in the catalog to the new metadata file. This process ensures that the new metadata is always based on the latest version, thereby maintaining a linear history, as each root metadata file is replaced exactly once. The metadata file contains information about various aspects of the table, including its schema, partition specs, sort order, properties, and a list of valid snapshots [10].

A *snapshot* of a table provides a read-only view of its data at a given point in time. It contains a complete listing of all files required to read the table. Readers maintain isolation by only using the files from a single snapshot when scanning the table. Data files in a snapshot are tracked by one or more *manifest* files that capture diverse information about data files such as their location, column statistics, and partition values. Manifests are not bound to specific partitions and can belong to multiple snapshots to avoid rewriting unchanged metadata. This key design principle enables Iceberg to quickly produce a new snapshot by inheriting all unchanged metadata. Iceberg indexes manifests for a snapshot in a *manifest list* file, enabling quick navigation through large chunks of metadata without the need for a distributed query engine. If the manifest skipping is not effective and an operation requires processing a large number of manifests, query engines can also leverage cluster resources by sending manifest metadata directly to compute to be read and evaluated remotely. Currently Spark uses this technique for distributed maintenance and planning. In the future, similar capabilities can be added for commits.

Every Iceberg commit can be seen as a series of actions, such as adding or removing data files, and a set of requirements, such as not finding any new data matching a predicate. The commit operation only succeeds if all applicable requirements are met, ensuring the table always transitions from one valid state to another. Each commit attempt creates a new metadata tree that will replace the current one. If the table state has changed during the action, the commit attempt fails and must be retried. Iceberg's metadata layout enables each retry to reuse all applicable work from earlier commit attempts. If a concurrent operation alters the table in a way that invalidates the pending commit, Iceberg discards the

pending metadata and reapplies all actions on top of the new base table version, making the retry process transparent to the user. The Iceberg spec imposes no isolation requirements and permits committing new table versions as long as they are coherent. However, Iceberg libraries provide mechanisms for query engines to maintain diverse levels of isolation, such as serializable or snapshot. The isolation levels are configured for each operation type (e.g. DELETE or MERGE) at the table level with the ability to override that configuration for particular invocations. Iceberg validates that changes meet isolation level requirements before allowing a commit. This can be used as a building block for multi-table transactions, but will require corresponding changes in catalogs and query engines. Instead of two-phase locking and serial execution, Iceberg relies on optimistic concurrency and does not require readers and writers to acquire locks. Serializable isolation is achieved by detecting snapshot isolation anomalies at commit time, as discussed in [35].

When planning a scan, Iceberg first determines which snapshot of the table must be read and finds all relevant manifests, pruning them based on partition summaries stored in the manifest list file. Iceberg then, either locally or remotely, scans these manifests to find matching data files, evaluating the scan predicate against the partition values and column statistics of each data file. The outcome of this step is a set of file scan tasks that are further split and grouped according to the desired split size.

Iceberg also allows tables to be partitioned on a subset of columns so similar rows can be clustered together which improves query performance through partition pruning. Unlike Hive, Iceberg adopts a technique called *hidden partitioning*, which decouples the values used for partitioning from actual column values. This is achieved via partition transforms [22] which are invoked transparently during reads or writes. Partition transforms are similar to generated columns since the transform output is derived from other column values. All valid partition transform are explicitly defined in the Iceberg spec for consistent implementation between engines. Consider the following SQL statements using partition transforms:

```
CREATE TABLE my_table (
  id BIGINT,
  data STRING,
  ts TIMESTAMP)
USING iceberg
PARTITIONED BY (bucket(16, id), days(ts));

SELECT data FROM my_table WHERE id = '42'
AND ts >= '2024-03-01 12:00:00';
```

In the above example, `my_table` is partitioned on two columns, `id` and `ts`, with partition transforms `bucket` and `days` respectively. Iceberg transparently clusters rows being written into this table after evaluating the transforms on the column values. During reads, Iceberg also uses the transforms to decide which partitions need to be read. Users can express filters without knowing how the rows are clustered, and Iceberg will transform those filters to match the partitioning. For instance, given a timestamp column `ts` partitioned by `hour(ts)`, a filter on a timestamp column `ts` will be automatically converted to a filter on `hour(ts)` internally. Iceberg tables can have multiple partition specs, allowing the partitioning to evolve without the need for rewriting data files.
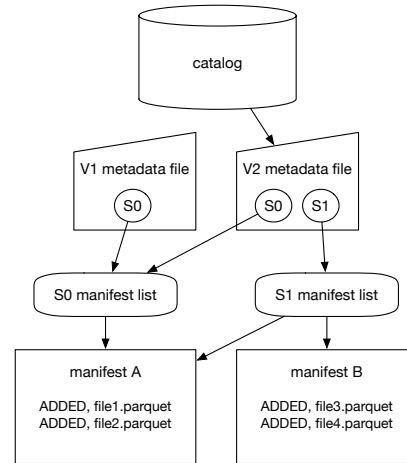

Figure 1: Iceberg metadata

2.2 Requirements and Constraints

There were multiple constraints and requirements that affected the design of row-level operations in Iceberg and Spark:

- **ACID transactions with a configurable isolation level.** Our primary target was online analytical processing (OLAP) use cases with a limited number of concurrent writers and many concurrent readers. Supporting millions of transactions per minute that modify a few records, common in online transaction processing (OLTP), was beyond the scope of this work.

- **Ability to handle tens of petabytes of highly-compressed columnar data in a single table.** Integrating support for row-level operations must have been achieved without compromising scalability.

- **Read and write-optimized ways to handle changes,** similar to projects like C-Store [55]. The solution had to support daily batch jobs targeting a large portion of records in a subset of table partitions, hourly micro-batches modifying a small set of records, and streaming upserts.

- **No long-running processes holding differential structures in memory.** Write-intensive databases frequently leverage data structures like LSM-trees [52] to buffer incoming changes in memory and periodically offload them to disk. This technique allows them to incorporate a fast write store to sustain a high frequency of updates and a larger read-optimized store on disk for compressed and compacted data. While we realized the need for differential structures like Positional Delta Trees [48] to handle sparse updates, it was important to avoid an in-memory component because of the complexity associated with ensuring durability and consistency.

- **Ability to sort the data according to a user-defined sort order.** Iceberg allowed users to configure a sort order for writes and kept lower and upper bounds for columns to skip irrelevant data files during planning. It was important to preserve this functionality for fast selective queries and not be tempted to order records in a way that is better suitable for handling changes.

- **Must be compatible with existing file formats like Apache Avro [2], Apache Parquet [4], and Apache ORC [3].** Developing a custom file format would harm the adoption and require lots of resources to maintain.
- **No extra information in data files.** We preferred to rely on implicit row IDs to not increase the storage footprint [30], as opposed to solutions that store extra metadata adjacent to data columns.
- **SQL support for DELETE, UPDATE, and MERGE commands.** Our goal was to hide the complexity that comes with row-level operations behind a standard SQL API. We also aimed to integrate the support for row-level operations into Spark's analyzer and optimizer. This was required to provide a clear view of how such operations would be executed and to perform advanced optimizations.

3 ICEBERG ROW-LEVEL OPERATIONS

We extended Iceberg to support three ways to encode row-level mutations, with the ability to combine them in a single table. This flexibility allows users to pick the right set of trade-offs for each workload. Each method can be thought of as several distinct phases: planning, scan, write, and commit.

3.1 Eager Materialization

Iceberg can eagerly materialize changes by rewriting and swapping data files that need to be modified, similar to block replacements in BigQuery [44] and micro-partition overwrites in Snowflake [27]. This strategy is internally called *copy-on-write*, following the naming convention in projects like VectorH [39]. The primary advantage of such materialization lies in its simplicity and absence of additional overhead during reads. This method is particularly effective for bulk updates targeting a substantial number of rows across specific data files, as it allows for the changes to be materialized once at the time of writing, rather than having each query reconcile differences with existing data files. The main downside is its inefficiency in handling sparse updates, as all unmatched rows in modified data files must be copied over, leading to increased write amplification. Updating a record in each data file of the table requires rewriting the entire table, making this strategy impractical for sparse changes at scale.

3.1.1 Planning. The planning mostly follows the same procedure as in regular Iceberg queries (see Iceberg Metadata). The only important difference is tracking additional snapshot information needed to ensure the configured isolation level.

3.1.2 Scans. Scans in row-level operations that replace data files have a few notable distinctions compared to normal queries. Query engines are required to project metadata columns, like partition or file name, in order to cluster and order records prior to writing. Column pruning and predicate pushdown beyond file filtering are also impossible because all unmatched records must be copied in their entirety into the new snapshot.

3.1.3 Writes. The write phase of the operations involves the creation of new data files that reflect the changed state of the table. Although the output data files are no different than the ones produced by appends, the write process requires special attention.

Poorly managed writes can drastically impact subsequent jobs and would require an aggressive compaction strategy to maintain stable query performance. We extended Iceberg with a set of relevant features to let query engines control different aspects of the write process. First, Iceberg allows users to configure a sort order for incoming records to enable efficient data skipping. Query engines are encouraged but not required to enforce the table sort order. The metadata for each added file is annotated with its sort order ID, enabling various read-time optimizations such as skipping unnecessary sorts during sort-merge joins. Second, Iceberg provides query engines with optimal instructions on how to distribute and cluster the data. Third, Iceberg libraries offer two types of file writers: clustered and fan-out. Clustered writers keep only a single file handle open at a time but require the input to be ordered by partition. This type of writer offers predictable memory pressure but necessitates a local sort prior to beginning. This local sort can facilitate better compression but is a frequent cause of expensive data spills from memory to disk. Fan-out writers do not require a sort and instead keep a file handle open per partition until the end of the task. This type of writer produces the same number of files as clustered writers but requires the system to keep more file handles open during the write.

3.1.4 Commits. Commits produce new snapshots by replacing the files read in the scan phase with the files created during the write phase (Figure 2). Commits currently support serializable and snapshot isolation. Under serializable isolation, Iceberg validates that no concurrent modifications have added new records matching the operation condition and that none of the scanned data files have been removed or have new delete files applied to them (see Lazy Materialization). This ensures that transactions are completely isolated from one another, serializing their execution. Under snapshot isolation, Iceberg permits write skew [33]. Commits will still succeed even if a concurrent operation adds new records that match the operation predicate as long as all replaced data files have not been removed and remain unmodified. Iceberg validates isolation by checking the partition and column statistics of newly added files enabling operations to concurrently modify the same partitions as long as it can be proven using the metadata that the concurrent transactions operate on disjoint sets of records. Other isolation levels and record-level conflict resolution can be added in the future.

3.1.5 Table Maintenance. Tables modified by eagerly rewriting data files do not require any special maintenance as long as the engine can apply the table's distribution and ordering at write time. Whenever write-time distribution or ordering are not possible, Iceberg libraries come with support for on-demand bin-packing, sorting, and multi-dimensional clustering.

3.2 Lazy Materialization

Given that changes can be scattered across all data files and rewriting the entire table to handle a small set of changes is impractical at scale, we also added support for lazy materialization. This strategy is called *merge-on-read*, a popular term in data lakehouses introduced by Apache Hudi [6]. Iceberg's data files are immutable so lazy materialization relies on a variant of differential files [53] to
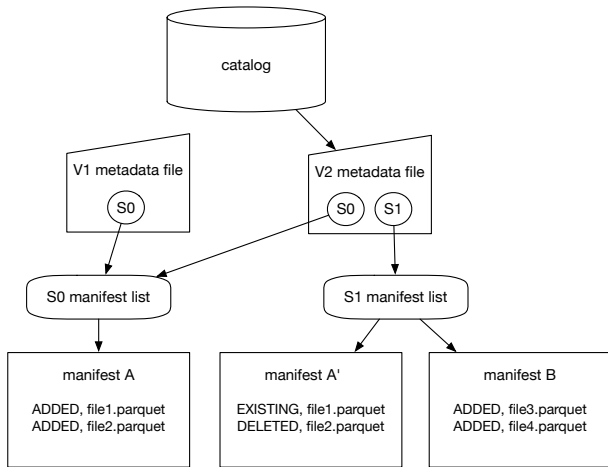
Figure 2: A row-level operation eagerly materializes changes by rewriting and replacing file2 from snapshot S0 with file3 and file4 in snapshot S1.

Figure 3: Partition-scoped position delete file

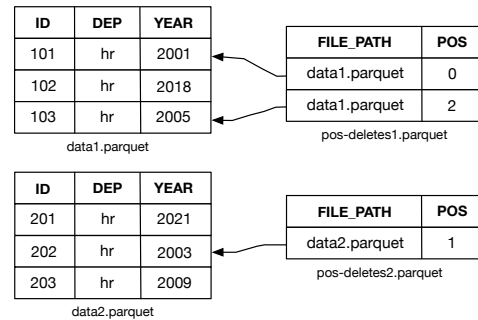
Figure 4: File-scoped position delete files

encode differences between existing data files and the newly desired state. These differences must be merged at read time with the existing data files to get a consistent view of the table. Unlike data file replacements, this approach required substantial changing of Iceberg's metadata layout and adding the concept of delete files. Delete files contain information about which rows are no longer present in the table. Updates are represented as a delete followed by an insert, which is common in columnar databases [55] [34]. While query engines may have an in-memory component to buffer changes similar to LSM-trees [52], it is not strictly required. Iceberg supports both position and equality delete files, allowing query engines to chose the right set of trade-offs for optimal performance.

3.2.1 Sequence Numbers. Iceberg determines the applicability of delete files based on sequence numbers which represent their relative age. Data and delete files distinguish two types of sequence numbers: file and data. Each snapshot in a table is assigned a sequence number during commit and all manifests, data files, and delete files created for the snapshot implicitly inherit its sequence number as their file sequence number. The file sequence number can then be used for incremental processing and table maintenance. The data sequence number exists to indicate the age of the content of the file. This allows table maintenance operations to replace existing files while still retaining metadata about when the contents was originally added to the table. The data sequence number can be used to limit the scope of delete files regardless of whether the file was added by a write or was the result of compaction. Sequence numbers are assigned via inheritance and do not cause contention.

3.2.2 Position Delete Files. A position delete file consists of a list of files and the row positions within those files that have been deleted. All position delete files are scoped to a particular partition and may have either file or partition granularity. Under partition granularity, writers group deletes for multiple data files within a partition into one delete file (Figure 3). This strategy tends to reduce the total number of delete files in the table. However, a
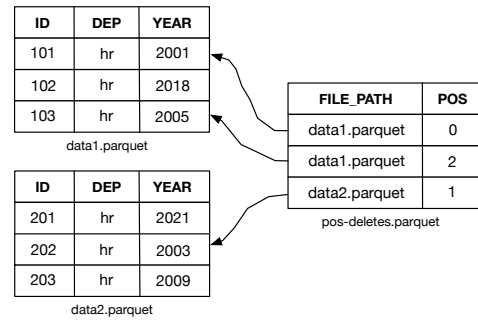
scan for a single data file will require reading delete information for multiple data files even if those other files are not required for the scan. Irrelevant deletes will be discarded but fetching this extra information will cause overhead. Under file granularity, delete writers always create a separate delete file for each modified data file (Figure 4). This strategy ensures only information required to read a data file will be loaded during a scan. However, it also increases the total number of delete files in the table and may require a more aggressive approach for delete file compaction. A position delete file must be applied to a data file if its data sequence number is *greater than or equal* to the data file's data sequence number. This allows writing delete files that are committed simultaneously with data files they modify, which helps when a single batch of changes contains multiple modifications for the same row.

3.2.3 Equality Delete Files. An equality delete file marks a row as deleted by referencing one or more column values, which identify rows that have to be removed (Figure 5). Equality delete files can be global or limited to particular partitions and store either entire deleted rows or just identity columns. Iceberg keeps track of identity columns for each equality delete file, allowing row-level operations to use different sets of columns to identify deleted rows. Iceberg also always stores lower and upper bounds for identity columns to facilitate file filtering during planning. An equality delete file must be applied to a data file if its data sequence number is *strictly greater* than the data file's sequence number and either their partitions match or the deletes belong to an unpartitioned spec. The latter allows having global deletes without the need to produce a delete file for each partition.
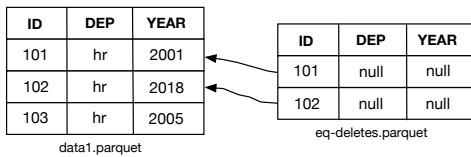
Figure 5: Equality delete file

3.2.4 Delete Manifests. Similarly to data files, delete files in a snapshot are tracked by one or more manifests. The content of each manifest (i.e. deletes or data) is persisted in the manifest list file, allowing Iceberg to manage metadata for data and deletes separately.

3.2.5 Planning. The introduction of delete files led to substantial changes in the planning process. Iceberg employs a two-phase approach to compute scan tasks if the table contains deletes. First, the Iceberg library finds delete files that match the operation condition and builds an in-memory delete file index. Second, Iceberg prunes data files and adds associations with relevant delete files found in the first phase. The result of this step is a set of scan tasks with data files marked with delete files that apply to them.

3.2.6 Scans. The scan phase of lazy materialization does not have the same limitations as eager materialization. Predicate pushdown can be applied meaning Iceberg can discard files or parts of the files that are not being modified. Iceberg can also utilize column pruning and only needs to project columns that either are part of the modification condition or are necessary to build the new state of updated rows. For instance, upserts only need columns for evaluating the upsert condition as the new state of the rows is fully derived from the upsert data.

Iceberg supports vectorized reads with position and equality deletes but the two delete types have different performance impacts on data file reads. Position deletes are almost always less expensive than equality deletes. Position deletes are loaded into a Roaring bitmap [38], a form of compressed bitmap, which acts like a validity vector. This approach adds minimal overhead to data file reads but is still sensitive to the number of associated delete files. Equality deletes are more expensive to apply as readers must compare values of identity columns in data and delete files. There are multiple options for applying equality deletes. Readers can load them into a set and discard all matching data records. Alternatively, query engines can sort-merge data and deletes if they have the same sort order. It is also possible to anti-join data and deletes using a distributed operation.

3.2.7 Writes. In addition to new data files containing updated and added records, writes also create delete files marking the records no longer present in the dataset. As in eager materialization, Iceberg recommends a way query engines can distribute the records to minimize the number of output files and provides dedicated clustered and fan-out delete writers.

3.2.8 Commits. Commits produce new snapshots with added data and delete files (Figure 6). Under serializable isolation, Iceberg validates that concurrent modifications do not add new records matching the operation condition, data files referenced by position deletes are still part of the table, and there are no new deletes for records
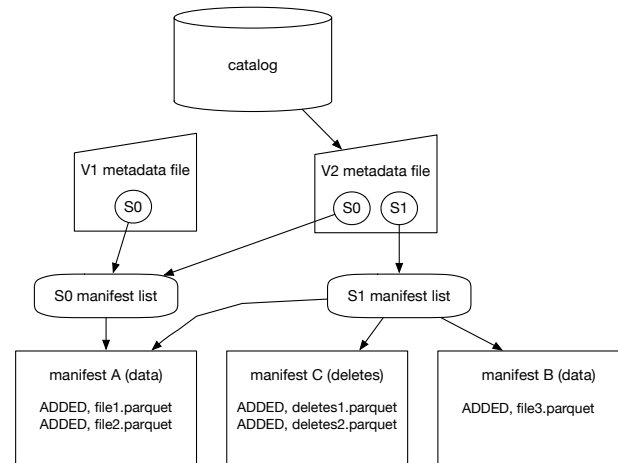

Figure 6: A lazy row-level operation indicates which records to remove in deletes1 and deletes2 and stores updates and inserts in file3.

that are being updated. Under snapshot isolation, the validation is similar except tolerating write skew. Equality deletes never conflict with data compaction or row-level operations that replace data files, making them a great choice for streaming applications. Position deletes, on the other hand, may conflict with a concurrent operation if that operation removes referenced data files from the table.

3.2.9 Table Maintenance. Lazy materialization demands additional maintenance not necessary with eager materialization. Procedures for expiring old snapshots and rewriting manifests have to take into account both delete and data files. Data compaction needs to take care of merging and removing obsolete delete files. Delete files themselves call for new table management procedures. For instance, position delete files can be compacted together to reduce read overhead without rewriting data files. This type of compaction is efficient because it does not require reading data. As of now, equality deletes cannot be compacted across different sequence numbers, but they can be converted into position deletes. This type of compaction is more expensive as it involves reading data files to determine deleted positions. It is possible to completely avoid the data and delete compaction by combining lazy and eager materialization. For instance, a job can execute a series of lazy operations followed by an eager operation, which would produce a new set of data files without deletes.

4 SPARK ENHANCEMENTS

Adding support for row-level operations to Spark required new connector APIs, additional planning components, and performance optimizations.

4.1 Connector API

Our goal was to create a generic API that would allow Spark to handle row-level changes across different data sources, regardless of what materialization strategy they support. The existing connector API provided mechanisms for controlling independent scans and

writes. However, in row-level operations writes depend on scans. For instance, Iceberg commits must be aware of what snapshot was scanned and the list of scanned data files to ensure a specific level of isolation, which is only possible by having access to the scan information during writes. To accomplish this we introduced a new API, RowLevelOperation [16], allowing connectors to coordinate scans and writes. This API also instructs Spark on what materialization strategy to use while rewriting DELETE, UPDATE, and MERGE statements in the analyzer. In the future, we plan to allow Spark to pick the right materialization approach automatically based on the nature of each operation. We also designed a set of new writer APIs for handling updates and deletes [15].

4.2 Runtime Filtering

Query engines frequently leverage runtime filtering [24] to dynamically prune data using predicates derived from analyzing or evaluating segments of the query plan. Spark already incorporated a few runtime filtering mechanisms, such as dynamic partition pruning and Bloom filter joins. We designed row-level operations to benefit from these optimizations, reducing the amount of processed and rewritten data. Connectors are able to utilize provided static filters when planning row-level operation scans. Unfortunately, such data skipping cannot be used in conjunction with more complex predicates. Complex predicates (e.g., id IN (SELECT value FROM source)) can only be evaluated in Spark and cannot be pushed down to connectors for data skipping. Since eagerly rewriting groups of data is costly, we added new filtering capabilities at runtime. Spark can evaluate a filter subquery to determine matching groups in the row-level operation scan. The cost of scanning the table multiple times is offset by only projecting columns required to evaluate the operation condition. This pre-filtering approach is equivalent to a lookup in an inverted index represented by another table. This functionality allows Iceberg to only rewrite data files that actually have matches. Inspired by a similar technique in Delta Lake [31], our implementation enriches this optimization by integrating the filtering step into the query execution, facilitating stage and exchange reuse. We are exploring a similar approach for plans with lazy materialization but that will require bigger changes to the runtime filtering framework.

4.3 Executor Cache

Connectors leverage executor-level caches to reduce the computation and IO overhead in tasks. In Iceberg we cache deletes to avoid repetitive work, as the same delete file can be read from several tasks. Caching is most effective when the delete file matches multiple data files or if different parts of the same data file are being processed separately. The executor cache has proven to greatly reduce the overhead of applying delete files during reads and defer the need for a major data compaction.

4.4 Storage-Partitioned Joins

IO and network communication are often the bottleneck in distributed parallel processing engines such as Spark [58][54]. In particular, during a shuffled join operation, Spark requires data to be re-partitioned, materialized on local disks for better fault tolerance, and later transmitted across network, so that the join can be divided into
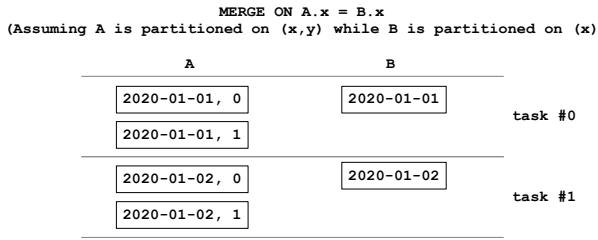
a set of smaller joins for which data is co-located on multiple machines. To be scalable and performant, the Spark shuffle operation also often must sort output rows according to the hashed value of their partition keys and materialize the result into a single file [41]. This process incurs significant overhead if the amount of data to be shuffled is large.

Traditionally, a co-partitioned join [43][45] is an effective technique to eliminate shuffle costs. The two tables of a join can be pre-partitioned based on the join keys, and, as data is co-located, it does not have to be re-shuffled across the network. Spark includes a similar idea called bucketing, which is inspired by Hive [36][56]. In the bucketing approach, a table is partitioned based on the hash of one or more columns into a fixed number of "buckets". A join operation can avoid shuffling if all the join keys are also bucket columns for the tables involved in the join, and the number of buckets is exactly the same or one side is a multiple of the other. This type of join is called a bucket join.
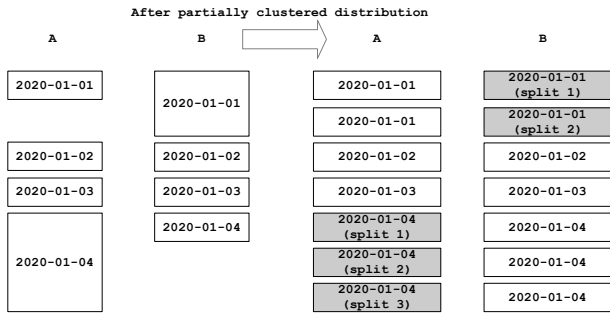
The bucketing approach has several limitations. First, all data has to be bucketed according to the hash value, and stored physically the same way as it is bucketed. Specifically in Hive and Spark, data for a particular bucket needs to reside in a single directory. Both engines also use different hash functions which makes bucketed tables maintained by one engine incompatible with another. Second, the set of join keys must match the full set of bucket columns. For instance, if two tables are joined on two columns, then both tables need to be bucketed on those exact two columns. Lastly, bucketing via hashing can often introduce data skew, where majority of the data could be concentrated in a few buckets which would bottleneck the algorithm. These limitations make the bucket approach not applicable to many use cases.

Storage-partitioned joins [13] (SPJ) generalize bucket joins by removing the above constraints. Data no longer has to be bucketed by hash, but instead can be clustered by any mechanism specified via connector's partition transform functions [22] registered via Spark's function catalog (e.g., days in Iceberg, which converts a timestamp value into a date value). The set of join keys does not have to match the full set of partition columns. Instead, Spark will ensure that partitions from both sides of a join match and can be co-located to Spark tasks. For instance, when the set of join keys is a subset of the partition keys, Spark will group partitions based only on the partition columns that are being used in the join. To illustrate, in Figure 7, assuming Spark is executing a MERGE operation where ON condition is on the x column from two tables A and B. Even though table A is partitioned on column x and y, Spark is able to group the partitions according to the ON condition, and ensure data can be co-located to the tasks #0 and #1.

SPJ also doesn't require transform functions on both sides of a join to be identical, but only that they are *compatible* with each other. In the bucketing case, two bucket functions are considered compatible if the number of buckets from one is divisible by the other, and that both use the same hash function. The concept, however, can also be extended to any transform functions provided by connectors, such as hours versus days in Iceberg, as long as they implement corresponding APIs in Spark. When Spark detects that two transform functions are compatible in a join operation, it uses the API to convert the partition values from one side of the join to the other. So far, SPJ allows "coalescing" partition values that are

```
                    MERGE ON A.x = B.x
    (Assuming A is partitioned on (x,y) while B is partitioned on (x))

            A                    B

    ┌──────────────────┐    ┌────────────────┐
    │ 2020-01-01, 0    │    │  2020-01-01    │
    ├──────────────────┤    └────────────────┘        task #0
    │ 2020-01-01, 1    │
    └──────────────────┘

    ┌──────────────────┐    ┌────────────────┐
    │ 2020-01-02, 0    │    │  2020-01-02    │
    ├──────────────────┤    └────────────────┘        task #1
    │ 2020-01-02, 1    │
    └──────────────────┘
```

Figure 7: Partition grouping

Figure 8: Partially clustered distribution

more granular into values that are less granular, e.g., converting an hour value since epoch time into a day value, and applies grouping to allow the partitions be co-located to Spark tasks.

Another common issue in original bucket joins is data skew, where a few tasks need to process the majority of the data, and therefore bottlenecking performance. This can happen in particular after applying the partition grouping as described above. To address this we introduced a new concept called "partially clustered distribution", where on one side of the join, data doesn't need to be fully clustered according to the join keys. Under this mechanism, a skewed partition can be divided into multiple smaller splits with each new split is compared against the full partition from the other side of the join. This improves parallelism because each new split can be processed independently at the cost of re-reading the opposing partition. This approach only works for inner joins, left outer joins when the right side is chosen as the side for splitting, and vice versa for right outer joins.

To illustrate the above, in Figure 8 table A is skewed on partition `2020-01-04`, while table B is skewed on `2020-01-01`. After applying partially clustered distribution, partition `2020-01-04` is divided into 3 smaller splits, each of which is mapped to the corresponding partition from table B. Similarly partition `2020-01-01` is divided into 2 smaller splits, each of which is mapped to the corresponding partition from table A. This increases both the number of splits as well as normalizing their size.

4.5 Cardinality Check

The SQL standard requires query engines to validate the cardinality of MERGE operations. The standard states that if the ON condition matches a single row from the target table with multiple rows

in the incoming changes, the outcome of the operation is undefined and an exception must be thrown. This validation ensures correctness but is frequently made optional in implementations because of the additional overhead [20]. In our implementation the operator performing the merge of target and incoming records always validates the cardinality. Because all matches for a given row in the target table must be in the same task, the check can be done locally without distributed computation and can avoid the overhead of other implementations. Our approach uses an existing expression in Spark for generating unique 64-bit integer row IDs to track matches. The initial implementation pre-sorted each partition of the joined dataset by the synthetic row ID and then checked for duplicates while emitting merged rows. The local sort performed poorly and led to costly spills to disk. We switched to using a compressed bitmap instead, bypassing the local sort and reducing the memory footprint.

4.6 Adaptive Writes

Significant effort was invested in allowing connectors to control various aspects of the writing process, as it is the most resource-intensive phase of any row-level operation and significantly impacts the performance of subsequent queries. We extended the connector API with the ability to request a specific distribution and ordering for incoming data. Three distribution types are supported: ordered, clustered, and unspecified. The ordered distribution instructs Spark to range-partition the data being written according to a given list of ordering expressions. This type of distribution handles data skew and ensures ordering across output files for efficient queries but requires expensive sampling to perform. The clustered distribution guarantees that records sharing the same values for the clustering expressions are co-located to the same task. This type of distribution is implemented using hash-based shuffles that are cheaper than range-based shuffles but may lead to less optimal data locality compared to the ordered distribution. Finally, the unspecified distribution covers scenarios when the data should be passed to connectors as is. Each distribution offers unique advantages and can reference both data and metadata columns. We also fully integrated the writing process into Spark's adaptive query execution framework. This allowed row-level operations to adjust the degree of parallelism based on statistics gathered at runtime and produce properly sized output files even if there was data skew.

5 EVALUATION

This section examines the efficiency of encoding changes and the impact on query performance of each materialization strategy. We populated the `store_sales` table from the TPC-DS benchmark with 2.8B records (scale factor 1000) and evaluated three upsert-like pipelines with various percentages of updated records. Each pipeline consisted of 10 consecutive upsert operations to assess the stability of write performance. We also repeatedly executed a simple aggregate query between the iterations to measure the read performance degradation. We explicitly chose a query that would be maximally sensitive to differences in table scan speeds. More complicated queries would be less impacted as the portion of time actually reading the underlying data becomes a smaller fraction of the total query time. For writes we report the time

taken for the operation. For reads we show the average query time of five executions after excluding the min and max times. We partitioned the target table by bucket(256, ss_ticket_number) and consumed changes from a temporary table with a compatible partitioning in order to benefit from storage-partitioned joins. In future releases, Spark will be able to apply a compatible partitioning on the fly. We also configured Spark to use hash joins to avoid expensive local sorts. We ran our benchmarks with Spark 3.5.1 and Iceberg 1.5.0 (with PR [23] to fix system predicate pushdown) on an EKS cluster of 8 nodes with 16 cores and 64GB RAM per node. File system tables were used via the Hadoop Catalog API in Iceberg.

5.1 Streaming Operations

Figure 9 (a) shows how different materialization strategies handled a streaming scenario with sparse and infrequent modifications. Each iteration contained 25 updates and 25 new records for every partition of the table (Case 1). Equality deletes offered the fastest way to encode the changes with constant write performance across iterations because differences were produced without scanning the target table. The implementation relying on position deletes also performed well but writes took longer and became slower with each iteration. This is expected as the operations required reading the target table to find affected positions and the number of files in the table increased with each iteration. The eager materialization approach was the slowest option for encoding the changes but the write time was constant across iterations. The read query time (Figure 9 (b)) slowly degraded under both lazy materialization strategies as the number of files containing differences increased (Figure 10). Each iteration contained updates and inserts for all partitions of the table and produced a data and a delete file per partition. In a production environment the data and delete files containing differences would have to be periodically rewritten and eventually compacted with the base data to restore performance. The eager approach preserved the original query performance (i.e., query time before any data modification is made: iteration=0) and would not require any table maintenance.

5.2 Micro-Batch Operations

Figure 11 (a) depicts a micro-batch pipeline where each operation contained 28M updates and around 6.5K inserts that were evenly distributed across all 256 buckets of the table (Case 2). The lazy materialization approaches significantly outperformed eager materialization during writes. In both the streaming and micro-batch use cases the eager strategy modified the exact same set of data files causing performance to be identical even though the number of modifications increased. Read performance of the lazy approaches was worse than the eager approach. Figure 11 (b) shows the rapidly increasing cost of applying equality deletes during reads. This accumulation of equality delete files would require frequent compactions to restore performance. The implementation using position deletes caused a much smaller degradation during reads, and its write performance was still almost 7 times faster than the eager implementation even after 10 iterations without any table maintenance. Figure 12 shows the cost and impact on query performance of a minor compaction of position deletes after 10 iterations. The compaction took only 23% of the time required for a single

eager iteration and decreased the read query time by 45%. After the compaction the query time was only 14% slower than the 0th iteration even though the table still contained more than 80M deletes. Minor compaction is limited in its ability to restore performance and eventually delete files must be merged into the base data files with major compaction. Use cases which modify the majority of data files in the table can avoid major compaction by switching from the lazy to the eager approach for one iteration of updates. Minor compaction for equality deletes is not yet supported in the Iceberg connector for Spark so it was not examined.

5.3 Batch Operations

Figures 13 (a) (b) represent a batch scenario where an iteration contained 137.5M updates and around 6.5K inserts that applied to only 25 out of 256 partitions, meaning each operation updated around 50% of all records in a subset of partitions (Case 3). While write performance of the eager approach was relatively constant, the lazy approach degraded proportionally to the cumulative amount of changes, leading to the inversion in their relative performance. Read performance with position deletes degraded faster than in the previous two use cases because of the increased volume of deletes in a small subset of partitions. The approach with equality deletes was excluded from the comparison because the Iceberg connector for Spark currently leverages a predicate-based approach for applying equality deletes, making it too resource-intensive for this use case.

5.4 Storage-Partitioned Joins

Figure 14 shows the impact of SPJ on write performance in 1st iteration of Case 1 (Figure 9 (a)). Both the eager and lazy materialization approaches show roughly an order of magnitude improvement in write performance with SPJ enabled. Row-level operations require shuffling the target and source tables so enabling SPJ improves write performance. We also observed that disabling SPJ required manually tuning the Spark advisory shuffle partition size to avoid creating undersized files. Enabling SPJ not only improved write performance but also facilitated stability.

5.5 Runtime Filtering

Figure 15 shows the impact of runtime file filtering during row-level operations. A small number of eagerly materialized updates were applied to only 25% of data files in the table (Case 4) with runtime filtering both enabled and disabled. The command did not contain any static predicates that could be pushed down during planning, highlighting the importance of dynamically determining the minimal set of files to rewrite at runtime. This use case was executed with only 32 cores in the cluster.

5.6 Summary

Our evaluation highlights the importance of supporting multiple ways to encode changes to efficiently handle diverse use cases. Lazy materialization with equality deletes is the best way to handle very sparse modifications in upsert operations as the differences can be produced without scanning the target table. Encoding 100 modifications in tables with 1GB or 10PB will require the same amount of resources during writes. Position deletes are much cheaper to apply during reads but require query engines to find positions of
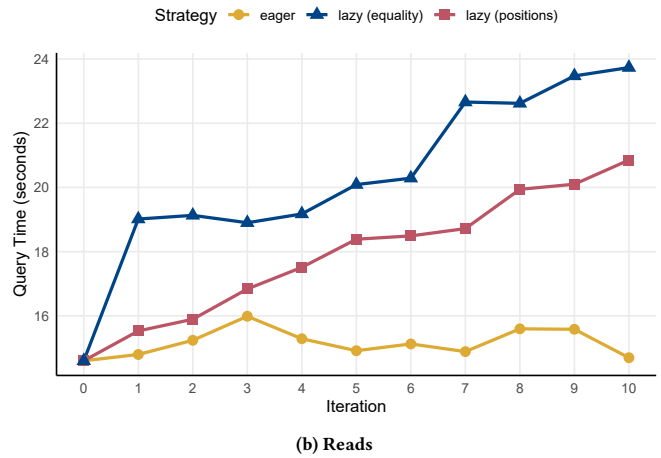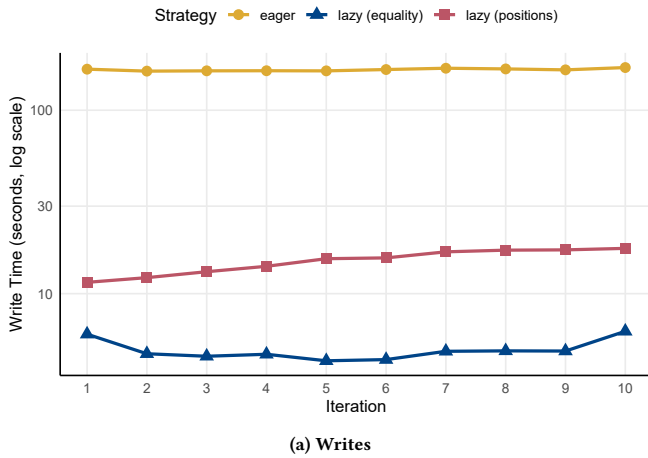
(a) Writes
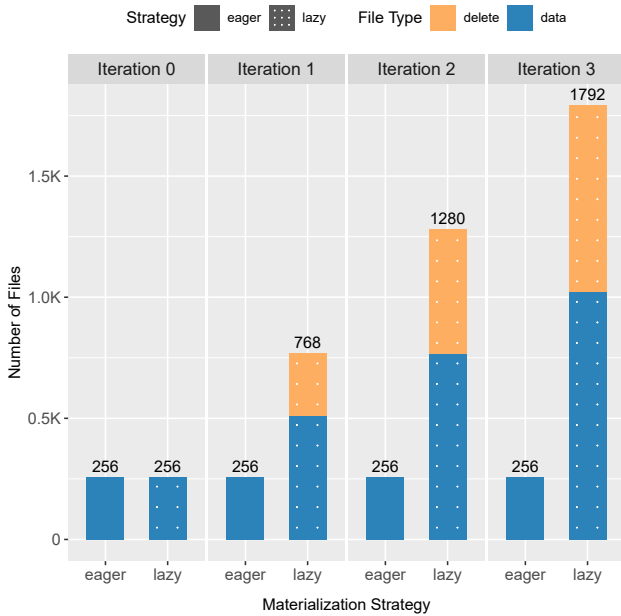
(b) Reads

Figure 9: Write and read time (Case 1)

Figure 10: Growth of file count (Case 1)

deleted and updated rows during writes. This materialization is best suited for micro-batch operations where each iteration targets a small percentage of rows across a large number of data files. Both delete types require additional table maintenance but position deletes can sustain a much higher percent of modifications before requiring any compaction. Eager materialization is well suited to daily bulk operations that target a large portion of the table. It does not impact query performance and does not require any special table maintenance. We also demonstrate that without key Spark optimizations these operations would take an order of magnitude longer. Introducing storage-partitioned joins, coupled with hash joins, eliminated sorts and shuffles and unlocked highly performant row-level operations.

6 FUTURE WORK

The current solution is widely deployed in production with tables containing tens of petabytes of data and tens of millions of files but there are some important areas for future work:

- **Ease of use.** We are working on enabling various optimizations such as storage-partitioned joins and runtime filtering seamlessly. We also plan to enhance the adaptive query execution in Spark to automatically pick an optimal advisory shuffle size for final writes. These changes will reduce the tuning required to achieve the desired performance.
- **Hybrid materialization.** Iceberg allows query engines to combine eager data file rewrites with adding position and equality deletes in one transaction. Our goal is to leverage this functionality in Spark to handle row-level operations that are dense in one subset of files and very sparse otherwise. In addition the selection of algorithm used for materialization can be determined automatically at runtime, removing any user configuration.
- **Multi-table transactions.** While Iceberg provides ways to ensure that changes in single table adhere to specific isolation requirements, it cannot guarantee the same isolation level for a DELETE or MERGE operation as a whole without the support for multi-table transactions in query engines and catalogs.
- **Secondary indexing.** We are actively exploring different ways to incorporate secondary indexes into Iceberg to further improve its data filtering capabilities and avoid scanning the target table to find affected positions in the lazy materialization strategy.
- **Alternative representations for position deletes on disk.** Iceberg currently uses Roaring bitmaps [38] to hold position deletes in memory and delegates the disk representation and compression to the underlying file format. We are evaluating options to either persist the Roaring bitmap directly or provide a custom serialization scheme similar to PFOR [60].
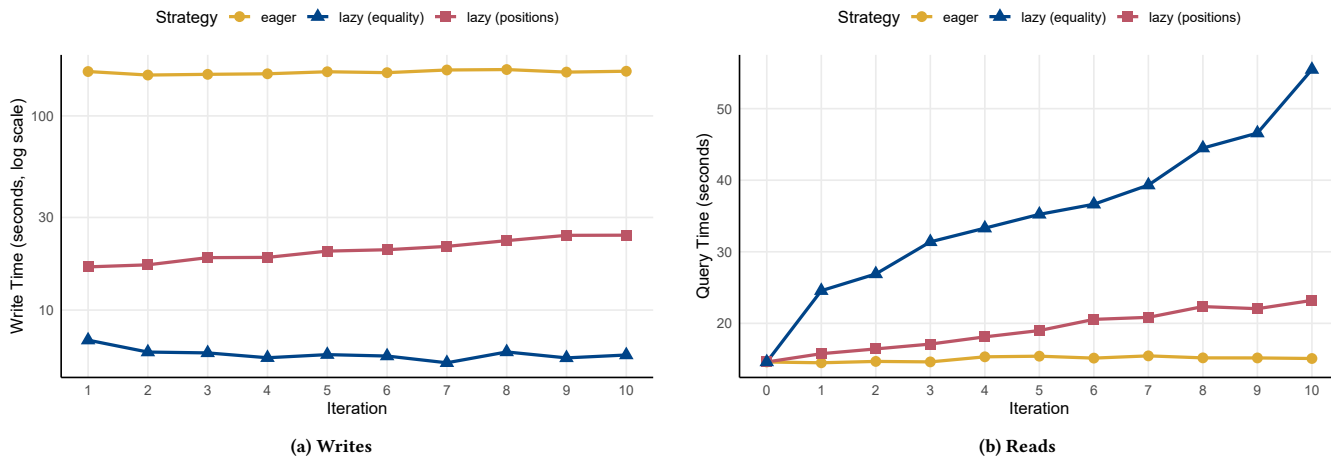
4168

(a) Writes

(b) Reads

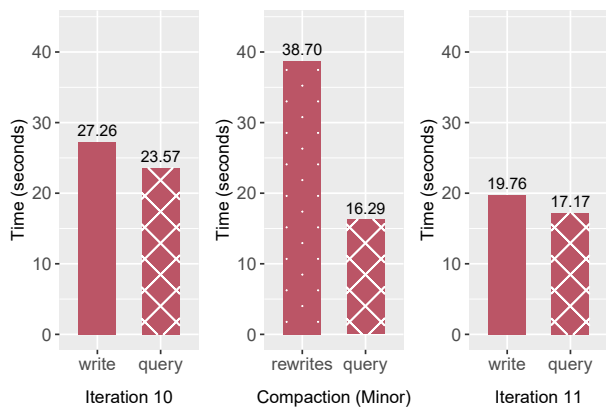Figure 11: Write and read time (Case 2)

Figure 12: The cost and impact on query performance of a minor compaction of position deletes (Case 2)

- **Enhanced runtime filtering.** We plan to extend the runtime filtering capabilities for approaches that work with deltas of rows to support Parquet row group and page filtering, which is not possible with eager data file rewrites.
- **Native execution.** We intend to integrate our solution with Apache DataFusion Comet [19] which is a pluggable Spark execution engine based on Arrow DataFusion [7], to further speed up row-level operations with the help of native Parquet readers and writers as well as highly efficient native Spark execution.

7 RELATED WORK

VectorH [39] is a SQL-on-Hadoop system that added support for row-level operations to Hadoop-based data lakes. The solution is proprietary and requires in-memory differential structures.

Apache CarbonData [5] is another system that supports row-level modifications in data lakes. The project relies on a custom file format, implements only position-based lazy materialization strategy, and lacks serializable isolation.

Hive ACID [17] was an attempt to rethink the table format in data lakes, focused primarily on Hive and ORC. It relies on delta files and requires all records to be sorted by a synthetic key, which is persisted in data files next to user-defined columns, making indexing hard as all tables have a fixed clustering layout not aligned with query patterns. This table format heavily depends on the Hive metastore and maintains only snapshot isolation. Because of these limitations, newer Hive versions adopted Iceberg and support its eager and lazy materialization strategies [25].

Apache Hudi [6] was specifically designed for updates and deletes in Hadoop. It introduced copy-on-write and merge-on-read terms to data lakes. Similar to Hive, the project was initially focused on HDFS and depended on the list operation in the underlying storage. Hudi supports eager and equality-based lazy materialization. However, these strategies cannot be mixed in the same table.

Delta Lake [31] originally supported only data file replacement, until the recent addition of delete vectors, a form of position-based lazy materialization. The eager and position-based lazy materialization approaches in Delta Lake and Iceberg are similar. We designed the Spark extensions so that they can be used by both systems. Delta Lake supports snapshot, write serializable, and serializable isolation. Row-level operations in Delta Lake are executed as a sequence of separate actions and are not integrated with the analyzer and optimizer in Spark, which makes debugging the execution hard and prohibits some advanced optimizations like stage or exchange reuse during runtime filtering. The open source version of the project also has limitations such as no support for subqueries in DELETE and UPDATE commands [14]. While Delta Lake does not yet support storage-partitioned joins, the proprietary Databricks Runtime offers low shuffle MERGEs [26] to reduce the amount of shuffled data. Unlike Iceberg, Delta Lake relies on storage's list operation to determine the last version file in the transaction log, which requires performant and consistent lists operations [11].
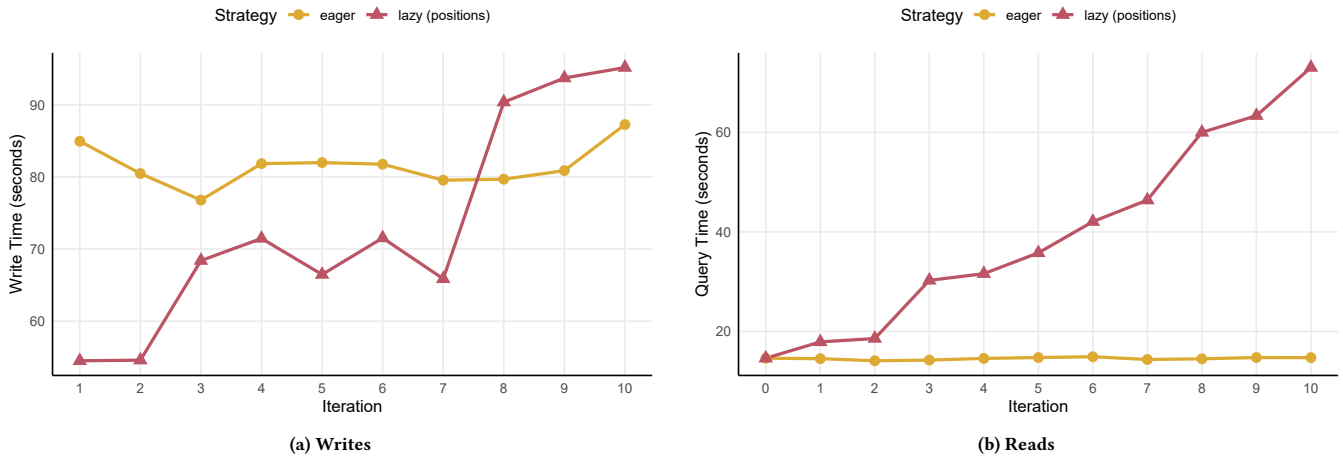
(a) Writes

(b) Reads

Figure 13: Write and read time (Case 3)

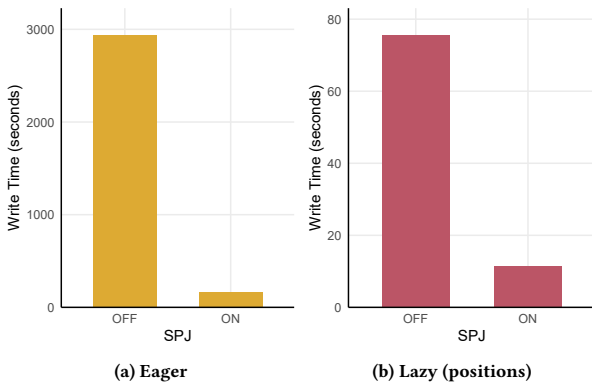(a) Eager

(b) Lazy (positions)

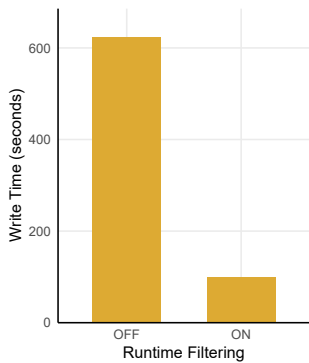**Figure 14: Impact of SPJ on write performance (Case 1)**

Figure 15: Impact of runtime-filtering (Case 4)

Recent studies [49] [37] within the research community have compared row-level operations in data lakehouses. Both papers found that Spark produced a bigger number of output files when executing Iceberg operations leading to worse read performance after modifying the table. While these studies uncovered issues in the default Iceberg and Spark configurations, they did not delve into the underlying reasons for the observed issues or discuss whether the performance could be improved by configuring the table format or query engine. The performance degradation could be attributed to Spark's adaptive query execution framework selecting an inadequately small shuffle partition size for final writes [18], or to the table being set up with a suboptimal distribution mode [21]. Delta Lake and Hudi don't support adaptive writes so they were not affected by the current limitation of the adaptive query execution framework in Spark. Our evaluation addresses the configuration issue and focuses exclusively on Iceberg and Spark to highlight the full potential of these technologies.

ACKNOWLEDGMENTS

REFERENCES

[1] 2006. *Apache Hadoop.* Retrieved February 2, 2024 from https://hadoop.apache.org

[2] 2009. *Apache Avro.* Retrieved February 2, 2024 from https://avro.apache.org

[3] 2013. *Apache ORC.* Retrieved February 2, 2024 from https://orc.apache.org

[4] 2013. *Apache Parquet.* Retrieved February 2, 2024 from https://parquet.apache.org

[5] 2016. *Apache CarbonData.* Retrieved Feb 2, 2024 from https://carbondata.apache.org

[6] 2016. *Apache Hudi.* Retrieved February 2, 2024 from https://hudi.apache.org

[7] 2018. *Apache DataFusion.* Retrieved May 1, 2024 from https://github.com/apache/datafusion

[8] 2018. *Apache Iceberg.* Retrieved February 2, 2024 from https://iceberg.apache.org

[9] 2018. *Apache Iceberg Java Library.* Retrieved Feb 2, 2024 from https://github.com/apache/iceberg

[10] 2018. *Apache Iceberg Spec.* Retrieved Feb 2, 2024 from https://iceberg.apache.org/spec/

[11] 2019. *Delta Lake S3 Writes.* Retrieved February 20, 2024 from https://github.com/delta-io/delta/issues/39

[12] 2020. *Apache Iceberg Python Library.* Retrieved Feb 2, 2024 from https://github.com/apache/iceberg-python

[13] 2021. *Apache Spark Storage-Partitioned Joins.* Retrieved April 12, 2024 from https://issues.apache.org/jira/browse/SPARK-37375

[14] 2021. *Delta Lake DELETE Subquery Support.* Retrieved June 9, 2024 from https://github.com/delta-io/delta/issues/730

[15] 2022. *Apache Spark Delta Writer.* Retrieved March 25, 2024 from https://github.com/apache/spark/blob/3bb762dc032866cfb304019cba6db01125556c2f/sql/catalyst/src/main/java/org/apache/spark/sql/connector/write/DeltaWriter.java

[16] 2022. *Apache Spark Row-Level Operation.* Retrieved March 25, 2024 from https://github.com/apache/spark/blob/3bb762dc032866cfb304019cba6db01125556c2f/sql/catalyst/src/main/java/org/apache/spark/sql/connector/write/RowLevelOperation.java

[17] 2023. *Apache Hive Transactions.* Retrieved June 14, 2024 from https://cwiki.apache.org/confluence/display/hive/hive+transactions

[18] 2023. *Apache Spark Configurable Advisory Partition Size for Writes.* Retrieved July 17, 2024 from https://issues.apache.org/jira/browse/SPARK-42779

[19] 2024. *Apache DataFusion Comet.* Retrieved June 14, 2024 from https://github.com/apache/datafusion-comet

[20] 2024. *Apache Hive DML Language Manual.* Retrieved June 7, 2024 from https://cwiki.apache.org/confluence/display/hive/languagemanual+dml

[21] 2024. *Apache Iceberg Distribution Modes.* Retrieved July 17, 2024 from https://iceberg.apache.org/docs/latest/spark-writes/#writing-distribution-modes

[22] 2024. *Apache Iceberg Partition Transforms.* Retrieved June 14, 2024 from https://iceberg.apache.org/spec/#partition-transforms

[23] 2024. *Apache Iceberg System Function Pushdown Fix.* Retrieved June 7, 2024 from https://github.com/apache/iceberg/pull/9873

[24] 2024. *Apache Impala Runtime Filtering.* Retrieved June 7, 2024 from https://docs.cloudera.com/runtime/latest/impala-reference/topics/impala-runtime-filtering.html

[25] 2024. *Cloudera Row-Level Operations.* Retrieved July 16, 2024 from https://docs.cloudera.com/cdw-runtime/cloud/iceberg-how-to/topics/iceberg-row-level-ops.html

[26] 2024. *Delta Lake Low Shuffle MERGE on Databricks.* Retrieved June 9, 2024 from https://docs.databricks.com/en/optimizations/low-shuffle-merge.html

[27] 2024. *Snowflake Micro-Partitions.* Retrieved June 11, 2024 from https://docs.snowflake.com/en/user-guide/tables-clustering-micropartitions

[28] Daniel Abadi, Rakesh Agrawal, Anastasia Ailamaki, Magdalena Balazinska, Philip A Bernstein, Michael J Carey, Surajit Chaudhuri, Jeffrey Dean, AnHai Doan, Michael J Franklin, et al. 2016. The Beckman report on database research. *Commun. ACM* 59, 2 (2016), 92–99.

[29] Daniel Abadi, Anastasia Ailamaki, David Andersen, Peter Bailis, Magdalena Balazinska, Philip Bernstein, Peter Boncz, Surajit Chaudhuri, Alvin Cheung, AnHai Doan, et al. 2020. The Seattle report on database research. *ACM Sigmod Record* 48, 4 (2020), 44–53.

[30] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreaos, and Samuel Madden. 2013. . https://doi.org/10.1561/1900000024

[31] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał undefinedwitakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[32] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797

[33] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *SIGMOD Rec.* 24, 2 (may 1995), 1–10. https://doi.org/10.1145/568271.223785

[34] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Conference on Innovative Data Systems Research*. https://api.semanticscholar.org/CorpusID:1379707

[35] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2009. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.* 34, 4, Article 20 (dec 2009), 42 pages. https://doi.org/10.1145/1620585.1620587

[36] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, Deepak Jaiswal, Slim Bouguerra, Nishant Bangarwa, Sankar Hariappan, Anishek Agarwal, Jason Dere, Daniel Dai, Thejas Nair, Nita Dembla, Gopal Vijayaraghavan, and Günther Hagleitner. 2019. Apache Hive: From MapReduce to Enterprise-Grade Big Data Warehousing. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1773–1786. https://doi.org/10.1145/3299869.3314045

[37] Jesús Camacho-Rodríguez, Ashvin Agrawal, Anja Gruenheid, Ashit Gosalia, Cristian Petculescu, Josep Aguilar-Saborit, Avrilia Floratou, Carlo Curino, and Raghu Ramakrishnan. 2023. LST-Bench: Benchmarking Log-Structured Tables in the Cloud. *arXiv preprint arXiv:2305.01120* (2023). https://arxiv.org/abs/2305.01120

[38] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2014. Better bitmap performance with Roaring bitmaps. *Software: Practice and Experience* 46 (2014), 709 – 719. https://api.semanticscholar.org/CorpusID:1139669

[39] Andrei Costea, Adrian Ionescu, Bogdan Răducanu, Michał Switakowski, Cristian Bârca, Juliusz Sompolski, Alicja Łuszczak, Michał Szafrański, Giel De Nijs, and Peter Boncz. 2016. VectorH: taking SQL-on-Hadoop to the next level. In *Proceedings of the 2016 International Conference on Management of Data*. 1105–1117.

[40] Benoit 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 *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[41] Aaron Davidson and Andrew Or. 2016. *Optimizing Shuffle Performance in Spark.* Technical Report. University of California, Berkeley.

[42] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[43] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. 2010. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 515–529. https://doi.org/10.14778/1920841.1920908

[44] Pavan Edara and Mosha Pasumansky. 2021. Big metadata: when metadata is big data. *Proc. VLDB Endow.* 14, 12 (jul 2021), 3083–3095. https://doi.org/10.14778/3476311.3476385

[45] Mohamed Y. Eltabakh, Yuanyuan Tian, Fatma Özcan, Rainer Gemulla, Aljoscha Krettek, and John McPherson. 2011. CoHadoop: flexible data placement and its exploitation in Hadoop. *Proc. VLDB Endow.* 4, 9 (jun 2011), 575–585. https://doi.org/10.14778/2002938.2002943

[46] Goetz Graefe. 2009. Fast Loads and Fast Queries. In *Proceedings of the 11th International Conference on Data Warehousing and Knowledge Discovery* (Linz, Austria) *(DaWaK '09)*. Springer-Verlag, Berlin, Heidelberg, 111–124. https://doi.org/10.1007/978-3-642-03730-6_10

[47] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1917–1923. https://doi.org/10.1145/2723372.2742795

[48] Sándor Héman, Marcin Zukowski, Niels J. Nes, Lefteris Sidirourgos, and Peter Boncz. 2010. Positional update handling in column stores. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 543–554. https://doi.org/10.1145/1807167.1807227

[49] Paras Jain, Peter Kraft, Conor Power, Tathagata Das, Ion Stoica, and Matei A. Zaharia. 2023. Analyzing and Comparing Lakehouse Storage Systems. In *Conference on Innovative Data Systems Research*. https://api.semanticscholar.org/CorpusID:259267242

[50] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: a decade of interactive SQL analysis at web scale. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3461–3472. https://doi.org/10.14778/3415478.3415568

[51] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 476–487.

[52] Patrick E. O'Neil, Edward Y. C. Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385. https://api.semanticscholar.org/CorpusID:12627452

[53] Dennis G. Severance and Guy M. Lohman. 1976. Differential files: their application to the maintenance of large databases. *ACM Trans. Database Syst.* 1 (1976), 256–267. https://api.semanticscholar.org/CorpusID:207632057

[54] Min Shen, Ye Zhou, and Chandni Singh. 2020. Magnet: push-based shuffle service for large-scale data processing. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3382–3395. https://doi.org/10.14778/3415478.3415558

[55] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-store: a column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway) *(VLDB '05)*. VLDB Endowment, 553–564.

[56] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. 2010. Hive - a petabyte scale data warehouse using Hadoop. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 996–1005. https://doi.org/10.1109/ICDE.2010.5447738

[57] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. USENIX Association, USA, 2.

[58] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (oct 2016), 56–65. https://doi.org/10.1145/2934664

[59] Matei A. Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *Conference on Innovative Data Systems Research*. https://api.semanticscholar.org/CorpusID:229576171

[60] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*. IEEE Computer Society, USA, 59. https://doi.org/10.1109/ICDE.2006.150