International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 06 Issue: 07 | July - 2022 SJIF Rating: 7.185 ISSN: 2582-3930

Comparison of Table formats for Data Warehouse

Arjun Reddy Lingala
arjunreddy.lingala@gmail.com

***Abstract*—Modern data warehouses are developed on distributed file system and object storage that offers scalability, data availability and performance. Table formats define how the data files are organized and stored on the file system. The evolution of data warehousing has given rise to diverse table formats with unique architectures and capabilities aiming at query performance, scalability and storage optimization. Hive table format is the foundational component of Hadoop ecosystem which uses centralized metastore and manual partitioning but the query performance is hindered in cases requiring incremental updates or complex query patterns. Hive table format fixed schema structure requires downtime and manual interventions for schema changes. Also, query planning for tables that have huge number of partitions takes lot of time. Iceberg table format addresses these issues with decentralized metadata management, snapshot isolation, and hidden partitioning. Iceberg supports dynamic schema adjustments with version control and backward compatibility. Further, Iceberg supports atomic commit capabilities which ensure consistency in high concurrent environments. This paper discusses how the data files are stored, how read and write patterns work, discuss the pain points in Hive table format and discuss in detail Iceberg table format, how it manages the files on the file system, how it addresses the challenges in Hive format. The comparison and overview aim to guide organizations in transitioning towards table formats that align with modern analytics requirements while ensuring long-term scalability and performance.**

***Keywords*—Hive, Iceberg, Table Formats, Data Warehousing, Apache Hadoop, Schema Evolution, Performance, Scalability**

I. Introduction

The rapid growth of big data has transformed the landscape of data analytics, making scalable and efficient data warehouses essential for processing and analyzing massive datasets. Data warehouses serve as the backbone of decision-making in industries supporting diverse use cases like real-time analytics, machine learning, and business intelligence. At the core of these systems are table formats, which define how data is stored, organized, and queried. The choice of a table format significantly impacts performance, scalability, and ease of data management. The Hive table format, introduced as part of the Hadoop ecosystem, has been a pioneer in providing SQL-like querying capabilities for data stored in Hadoop Distributed File System (HDFS) [7]. Hive [1] organizes data into partitions and uses a centralized metastore to manage metadata, making it a reliable choice for traditional data warehousing. Its simplicity and integration with a wide range of tools have contributed to its widespread adoption. With evolution of data analytics and increase in data size for analytics, limitations with Hive table format have became more common. Issues such as complex schema evolution, inefficient handling of incremental data updates, and lack of advanced metadata management impact query performance and scalability. These challenges have led to the development of newer table formats and Apache Iceberg table format has been widely adapted. Apache Iceberg [2] was designed to address the shortcomings of older table formats by introducing features such as decentralized metadata, hidden partitioning, and snapshot isolation. These innovations make Iceberg particularly suited for cloud-native environments and modern big data applications requiring high performance, flexibility, and scalability. Iceberg also introduces advanced capabilities like time-travel queries and atomic commits, enabling consistent and reliable operations even in high-concurrency scenarios.

The remainder of this paper is structured as follows: Section 2 provides background information on Hive table format, discussing its origins, architectures, read and write patterns, challenges. Section 3 outlines Iceberg table format, architecture, read and write patterns. Finally, Section 4 concludes the paper with recommendations and potential directions for future research.

II. Hive Table Format

Apache Hive is a foundational table format designed to enable SQL-like querying capabilities on top of the Hadoop Distributed File System (HDFS) [7]. Its architecture and data patterns are optimized for batch processing and large-scale data analytics.

A. Overview and Architecture

Hive's table format architecture revolves around organizing data into a structured schema and partitioning it for efficient querying. Its core components include

1) Metastore: Hive [7] uses centralized metastore which is usually MySQL database to maintain metadata about tables, partitions, columns and file locations. Any processing engine like Apache Spark [6] and Presto [12] or any new engine is supported by unified metadata layer supported by metastore. Metastore is the single entry point for query processing and it can become a bottleneck as the number of tables and partitions grow.

2) Storage Model: Hive table format relies on HDFS for data storage and organizes data into directories and files. Hive uses schema on read approach, allowing data to be stored in raw format and during query execution schema is interpreted based on table schema.

3) Partitioning and Bucketing: Hive table format partitioning divides data into directories based on partition keys, enabling pruning during query execution to minimize the amount of data scanned. Within partitions, data can be divided

into buckets based on hash values of a specified column, improving join and aggregation performance.

B. Read Pattern

In Hive, data is stored in HDFS, and the Hive table format organizes this data into partitions and files. Queries are written in HiveQL (SQL-like syntax) and executed using a query engine, such as Hive's native execution engine, Apache Tez [5], or Spark [6]. The query execution process involves multiple stages of optimization and data retrieval.

1) Parsing and Optimization: When a query is submitted to Hive query engine, it parses the input HiveQL query into abstract syntax tree (AST). Semantic analysis is performed to validate the query against the table schema and metadata. Next, query optimizer generates a logical plan and applies optimizations like predicate pushdown and partition pruning. Additional optimizations like vectorized execution is applied based on execution engine.

2) Partition Pruning: Hive partition pruning identifies the partitions relevant to query predicates and retrieves only specific partitions avoiding full scan of the table. Partition pruning is achieved by evaluating partition metadata stored in Hive metastore which reduces I/O overhead significantly.

3) File Scanning: Hive reads raw data from files and applies the table schema to parse and interpret it during query execution. This flexibility allows Hive to work with semi-structured and evolving data. Columnar file formats like ORC and Parquet enable efficient file scanning by reading only relevant columns and skipping irrelevant rows using stats saved in columnar file formats.

4) Data Transformation: Hive performs predicate pushdown which pushes down the filters to storage layer, reducing the amount of data to read from disk. During transformations, intermediate results are often written to temporary storage before the final aggregation. Hive uses broadcast joins or bucketing based joins when multiple tables are involved to minimize data shuffling. After transformations, the processed results are usually written to a file or stored in another table.

C. Write Pattern

In Hive, write operations typically include data transformations, partitioning, and metadata updates. Hive's write operations are designed for high-throughput rather than low-latency writes.

1) Data Transformation: Hive applies transformations to ensure that it matches the schema and structure of the target table by converting input data types to match the column definitions, makes sure incoming data aligns with table schema, derives partition keys from input data for partitioned tables.

2) Partition and Bucketing: Partitions improve query performance by limiting the data scanned but add complexity to write operations. Data is divided into directories based on partition keys. Data within a partition can be further divided into buckets based on a hash function applied to a specified column.

3) Write to HDFS: Hive writes large batches of data to minimize the overhead of small writes. Data is first written to temporary locations before being committed to the final directory. Compressed writes reduce storage space and improve read performance but may add overhead during write operations. Common compression codecs include Snappy [10], Gzip [11], and Zlib.

4) Metadata Updates: After data is written to HDFS, the Hive metastore is updated with information about new partitions, files, and table schema with newly created partitions are registered in metastore and schema is validated.

D. Challenges

1) Metadata Management: Hive relies heavily on the Hive Metastore to store metadata about tables, partitions, and schemas. Inefficient partition listing, high overhead in querying and updating metadata will become a bottleneck in large datasets with more partitions or small files. In addition to this, query performance also degrades as the number of partitions of files increases.

2) Small file Problem: Hive often writes data in small files, especially when ingesting data in micro-batches or through frequent writes. HDFS inefficiencies arise as each small file requires its own metadata leading to increased load on the name node, slower query performance due to higher file scan overhead

3) Snapshot Isolation and Transaction Support: Hive is inherently append-only, making it unsuitable for use cases requiring updates or deletes. While ACID transactions were introduced, they come with significant performance overhead due to compaction processes and limitations in handling concurrent modifications. Hive lacks native support for snapshot isolation, leading to potential data inconsistency during concurrent read/write operations.

4) Schema Evolution: Schema changes in Hive, such as adding, renaming, or deleting columns, are difficult to handle. Compatibility issues arise when downstream systems consume data with older schemas. Hive lacks built-in mechanisms to manage schema versions effectively.

5) Query Performance: Hive table format's directory and file-based partitioning results leads to high overhead during query planning and execution, full table scans when partitions are not properly pruned.

6) Time Travel: Hive does not natively support time travel, making it difficult to query historical data states. Without versioning, rollback or audit of data changes is cumbersome and requires external tooling.

7) Real Time: Hive is optimized for batch processing and struggles with real-time or near-real-time streaming data ingestion. High write latency and lack of efficient small file handling hinder its applicability for modern streaming use cases.

III. Iceberg Table Format

Apache Iceberg is a table format for big data environments that supports schema evolution, multi-engine interoperability,

and efficient querying. It provides a robust layer for organizing datasets stored in distributed file systems or object stores, enabling reliable analytics at scale. Key features of Iceberg include hidden partitioning, schema evolution, snapshot isolation, time travel, efficient metadata management

A. Overview and Architecture

Iceberg's architecture is designed to improve query performance and manageability by separating metadata from the physical layout of data. Iceberg [2] tables abstract the underlying data storage and expose a SQL-like interface. Tables are defined using schemas and partition specifications. Metadata layer is the key component of Iceberg's architecture which ensures efficient operations on large scale datasets. Table metadata maintains the top-level information about the table including schema, partition specification, list of snapshots and current snapshot reference saved as a single JSON file named *metadata.json*. Snapshots represent point-in-time views of a table which contains references to manifest files and data files at a specific state enabling time travel and rollback. Manifest files track metadata for data files in the table that provides file level details like file path, record count, file size, and partition information. Manifest files are small and designed for faster access. Manifest lists summarize metadata about manifests files and they point to individual manifest files aggregating file stats. Iceberg stores data files in distributed file systems or object stores and it uses metadata to decouple logical table structure from physical file locations.

B. Read Pattern

Iceberg's read pattern is designed to be metadata-driven, minimizing the need to scan unnecessary files or partitions. By leveraging rich metadata, predicate pushdown, and manifest files, Iceberg ensures that queries read only the data required for computation.

1) Metadata: The read process starts by loading metadata file which contains schema definition, partition specifications, list of snapshots, and current snapshot reference to identify the relevant snapshot for the query. Metadata file is very compact making it lightweight to load and parse and has minimal latency.

2) Snapshot Resolution: Reader determines the active snapshot based on the metadata file. Snapshots enable quick access to consistent table states, avoiding the need to traverse the entire file system.

3) Manifest Files: Reader next loads the manifest files to identify candidate data files. Manifest files contain metadata about the data files which includes file paths, partition values, and file level statistics like minimum value, maximum value, number of null values. Manifest files allow the query engine to eliminate irrelevant partitions or files early in the process.

4) Predicate Pushdown: Iceberg performs predicate pushdown at the metadata level, applying query filters to the manifest files to prune unnecessary data files. Predicate pushdown reduces the number of files that need to be scanned, minimizing I/O operations. This approach is particularly effective for large datasets with many partitions.

C. Write Pattern

Write pattern in Iceberg is designed to meet the needs of modern data processing, ensuring *Atomicity, Consistency, Scalability, Flexibility*. The process of writing data to an Iceberg table involves multiple stages, ensuring data consistency and efficiency.

1) Transaction Init: The writer fetches the current metadata snapshot to ensure that changes are applied to the latest table state and starts with initialization of a transaction. Snapshot isolation ensures that write process is isolated from concurrent reads and writes.

2) Data File Generation: Each writer generates multiple small, independent data files to optimize parallelism and writes data in columnar formats such as Parquet, Avro and ORC. Columnar formats enables efficient compression and encoding supporting predicate pushdown for future queries. Data files are organized based on the table's partition spec, ensuring efficient pruning during reads and typically files of size of 128MB to 1GB are created to balance storage efficiency and read performance.

3) Metadata Creation: Each data file creates its metadata which includes file path, record count, partition values, and column stats with minimum, maximum, null values. It updates manifest files that contain metadata for set of data files including statistics for filtering and pruning. It also updates manifest list pointing to all manifest files in current snapshot.

4) Atomic Commit: Once all data files and metadata are written, the changes are committed atomically. The writer checks that the current metadata snapshot has not changed during the write process. A new snapshot is created, referencing the updated manifest list and is updated as current snapshot. If validation fails, the write operation is retried.

TABLE I
COMPARISON OF HIVE AND ICEBERG TABLE FORMATS

Feature	Hive Table Format	Iceberg Table Format
Atomicity	Limited	Full Atomic Commit
Partition Management	Directory based	Metadata based
Schema Evolution	Limited	Fully Supported
Concurrency Support	Limited	Optimistic concurrency control
Write Modes (Append, Overwrite)	Basic	Flexible

IV. HOW ICEBERG ADDRESSES CHALLENGES

Apache Iceberg with its modern table format designed to handle large scale datasets in distributed environments

addresses the challenges face with Hive table format.

1) Advanced Metadata Management: Iceberg maintains a manifest file structure, replacing Hive's dependency on the metastore for partition and file listing. Metadata is stored in a highly optimized format and it only takes O(1) partition discovery regardless of the number of partitions with fast metadata lookups. Iceberg avoids Hive's overhead of querying and updating the metastore frequently, improving scalability.

2) Small File Problem: Iceberg consolidates metadata for all data files into manifest lists and provides automated file compaction ensuring efficient handling of micro-batches, reduced load on HDFS Name Node, optimal file sizes for both storage and query performance.

3) Transactions: Iceberg provides snapshot isolation, allowing multiple concurrent readers and writers without conflicts. Transactions in Iceberg are atomic, ensuring data consistency during writes. Unlike Hive ACID, Iceberg does not rely on compaction, eliminating associated performance overhead.

4) Efficient Partition Management: Iceberg implements hidden partitioning, decoupling partition management from the physical directory structure. Faster partition pruning and reduced complexity in managing high-cardinality partitions is achieved with metadata tracking partitions rather than relying on directory names. Iceberg's partitioning system supports advanced techniques like transformations (e.g., bucketing, truncation) to optimize query performance.

5) Schema Evolution: Iceberg supports evolutionary schemas which allows additions, renames, and deletions of columns without breaking compatibility and versioning to maintain historical schema states. Downstream consumers can seamlessly handle schema updates without requiring manual interventions.

6) Time Travel: Iceberg provides built-in time travel capabilities, allowing users to query historical states of the dataset. Metadata versions and snapshots enable rollback to specific points in time, ensuring robust auditability and compliance.

7) Real-Time and Streaming: Iceberg integrates with streaming platforms like Apache Kafka, Flink, and Spark Structured Streaming. Its architecture supports real-time ingestion by efficiently handling small incremental writes and compactions.

V. Conclusion

The rapid growth of big data analytics and the increasing reliance on data lakes have brought table formats into sharp focus as a critical component of modern data infrastructure. This paper presented a detailed comparison of the Apache Hive table format and the Apache Iceberg table format, exploring their architectures, read and write patterns, and the challenges they address. The analysis highlights the advancements introduced by Iceberg over the traditional Hive format, emphasizing its suitability for large-scale, cloud-native, and evolving data ecosystems. Iceberg's support for schema and partition evolution, time travel, and incremental reads and writes significantly outpaces Hive's capabilities. These features align with modern data use cases, such as machine learning pipelines, real-time data processing, and regulatory compliance. As the field of data management evolves, the adoption of advanced table formats like Iceberg will likely grow. However, the choice between Hive and Iceberg may depend on organizational constraints, such as existing ecosystem investments and the specific requirements of analytics workloads. Future work could explore hybrid approaches, where Hive and Iceberg coexist within a single architecture, leveraging the strengths of both systems.

References

[1] A. Thusoo, J. Sen Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, N. Zhang, S. Shah, and R. Murthy, "Hive: A Warehousing Solution Over a Map-Reduce Framework," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, Aug. 2009. doi: 10.14778/1687553.1687609.

[2] R. Ryan Blue, A. Goyal, and R. Dayson, "Apache Iceberg: Open Table Format for Huge Analytics Datasets," Apache Software Foundation, 2018. [Online]. Available: https://iceberg.apache.org/

[3] J. Cryans, D. Borthakur, T. Dunning, M. Dong, G. Malamud, and T. Chaitanya, "Apache Parquet: Efficient Columnar Storage for Hadoop and Big Data Ecosystems," Apache Software Foundation, 2013. [Online]. Available: https://parquet.apache.org/

[4] O. Balaban, S. Seth, O. O'Malley, and S. Radia, "Apache ORC: Optimized Row Columnar Storage for Big Data," Apache Software Foundation, 2013. [Online]. Available: https://orc.apache.org/

[5] S. Seth, V. Kumar, S. Radia, B. Eng, and R. Chaiken, "Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications," Apache Software Foundation, 2014. [Online]. Available: https://tez.apache.org/

[6] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational Data Processing in Spark," *Proc. 2015 ACM SIGMOD Int. Conf. Management of Data (SIGMOD'15)*, Melbourne, VIC, Australia, pp. 1383–1394, May 2015. doi: 10.1145/2723372.2742797.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Hadoop Distributed File System," *Proc. 2003 ACM Symp. Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, USA, pp. 107–120, Oct. 2003. doi: 10.1145/945445.945450.

[8] M. S. S. R. Chakka, S. G. Avasarala, A. Goyal, and M. Zadeh, "Apache Hudi: A Distributed Data Lake Framework for Streamlining Data Processing in Big Data Ecosystems," *Proceedings of the 2020 IEEE International Conference on Big Data (Big Data)*, pp. 1724–1734, Dec. 2020. doi: 10.1109/BigData50022.2020.9377790.

[9] D. W. Embley, P. J. M. Van der Aalst, and M. L. De Moura, "Data Lakes: A Survey of Modern Data Storage and Analysis Architectures," *IEEE Access*, vol. 7, pp. 57383–57398, 2019. doi: 10.1109/ACCESS.2019.2917422.

[10] D. L. Meier, J. Dean, and S. Ghemawat, "Snappy: A Fast Compressor for Modern Systems," *Google Inc.*, 2011. [Online]. Available: https://snappy.googlecode.com/

[11] M. J. Rochkind, "The Gzip File Compression Utility," *IEEE Software*, vol. 11, no. 3, pp. 79–85, May 1994. doi: 10.1109/52.286397.

[12] D. R. Oppenheimer, M. C. Isard, and A. S. Ousterhout, "Presto: Distributed SQL Query Engine for Big Data," *Proc. of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*, pp. 11-16, May 2012. doi: 10.1145/2213836.2213841.