Iceberg Interview Q&A

Q.) What is a data lakehouse?

A.) A data lakehouse is an architectural pattern where most of the workloads often associated with a data warehouse take place on the data lake. This reduces the duplication of data and complexity of data pipelines for better regulatory compliance, consistency of data, and self-service delivery of that data.

Q.) What is a data lakehouse table format?

A.) A table format allows tools to look at files in your data lake storage and recognize groups of those files as a single table to enable you to query and transform the data performantly directly on the data lake, enabling data warehouse-like workloads (and more) on the data lake (also known as a data lakehouse).

Table formats are not new or specific to data lakes — they've been around since System R, Multics, and Oracle first implemented Edgar Codd's relational model, although "table format" wasn't the term used at the time. A data lakehouse table format is different from those because multiple engines need to interact with it.

Blog ->
https://www.dremio.com/resources/guides/apache-iceberg-an-architectural-look-under-the-covers/

Q.) What are metadata files?

A.) Every time a table is created or data is added, deleted, or updated from a table a new metadata file is created. The job of the metadata file is to track the high-level definition of the table such as current/past snapshots, current/past schemas, current/past partitioning schemas, and more.

Blog ->
https://www.dremio.com/blog/a-hands-on-look-at-the-structure-of-an-apache-iceberg-table/#metadatafile

Q.) What are manifest lists?

A.) Each snapshot of the table is tracked in a file called a manifest list. These files track which manifest files make up the table at that particular snapshot along with metadata on those files to allow query engines to filter out unnecessary files through techniques like partition pruning.

Blog ->
https://www.dremio.com/blog/a-hands-on-look-at-the-structure-of-an-apache-iceberg-table/#manifestlist

Q.) What are delete files?

A.) Delete files are files that track rows that have been deleted and should be ignored from certain data files. Delete files are used when a table sets its row-level operations (update, delete, merge) to Merge-on-Read (MOR). Delete files come in two styles: position deletes that track the position of deleted records in the file and equality deletes that track the deleted rows by the rows' column values.

Blog ->
https://www.dremio.com/blog/row-level-changes-on-the-lakehouse-copy-on-write-vs-merge-on-read-in-apache-iceberg/

Q.) What is partition evolution?

A.) In most table formats if you decide to change how you partition data to improve query times, you have to rewrite the entire table. This can be pretty expensive and time-consuming at scale. Apache Iceberg has the unique feature of partition evolution where you can change the partitioning scheme of the data going forward without having to rewrite all the data partitioned by the old partition scheme.

Blog ->
https://www.dremio.com/blog/future-proof-partitioning-and-fewer-table-rewrites-with-apache-iceberg/

Q.) What is hidden partitioning?

A.) Partition evolution works because instead of tracking partitions by columns, Iceberg tracks based on two factors: a column and a transform. This pattern also unlocks another feature unique to Iceberg, hidden partitioning. The result is data consumers don't have to worry about how the table is partitioned to take advantage of that partitioning, unlike other formats that usually require creating extra "partition" columns that have to be included in filters to take advantage of.

Blog ->
https://www.dremio.com/blog/fewer-accidental-full-table-scans-brought-to-you-by-apache-icebergs-hidden-partitioning/

Q.) What is time travel?

A.) Since Iceberg's metadata tracks the snapshots of the table, you can query the table as it was at any particular time; this is known as time travel. This is great for testing machine learning algorithms on the same data that previous versions may have been tested against for better comparison or to run historical analytics.

Blog -> https://www.dremio.com/blog/time-travel-with-dremio-and-apache-iceberg/

Q.) What is schema evolution?

A.) Schema evolution is the ability to change the schema of the table. Without a table format like Apache Iceberg, updating a table's schema on the data lake may require a rewrite of the table. Apache Iceberg allows you to add columns, update column types, rename columns, and remove columns from a table without the need to rewrite it.

Blog -> https://iceberg.apache.org/docs/latest/evolution/#schema-evolution

Q.) How does Iceberg handle multiple concurrent writes?

A.) Iceberg uses "optimistic concurrency" to provide transactional guarantees during multiple concurrent writes. When multiple writers attempt to write to an Iceberg table, they'll project the filename or hash for the next snapshot. Before committing the next snapshot, they'll check that the projected file name/hash still doesn't exist. If it doesn't, it will commit the table update and if it does, it will create a new projection and reprocess the update until successful or use up the maximum reattempts.

Docs -> https://iceberg.apache.org/docs/latest/reliability/#concurrent-write-operations
Docs -> https://iceberg.apache.org/spec/#file-system-tables

Q.) What is Copy-on-Write and Merge-on-Read and when should I use one vs. the other?

A.) Copy-on-Write (COW) and Merge-on-Read (MOR) are two strategies for handling row-level updates for a table (update, delete, merge). In COW, files with updated records are rewritten, which is optimal for reads but slowest for writes. In MOR, files with updated records are kept, and instead a delete file is written that tracks updated/deleted rows and is then merged together when the table is read. Fast writes with a cost at read time.

Blog ->
https://www.dremio.com/blog/row-level-changes-on-the-lakehouse-copy-on-write-vs-merge-on-read-in-apache-iceberg/

Q.) How do I ensure a record is hard deleted for regulatory purposes like GDPR?
Apache Iceberg data files are deleted when they are no longer associated with a valid snapshot.

A.) When using Copy-on-Write, data files containing changed data are rewritten, so expiring all snapshots from the deletion and prior will result in the deletion of the datafiles with those deleted records.

When using Merge-on-Read, data files aren't rewritten so they may be associated with snapshots after the deletion occurred. To hard delete the files you will want to run a compaction job and then expire all snapshots before the compaction job which will delete all files that include those deleted records.

Blog -> https://www.dremio.com/blog/apache-iceberg-and-the-right-to-be-forgotten/

Q.) What is compaction and how do I do it?

A.) Compaction allows you to take smaller less efficient data files and rewrite them into fewer larger data files for improved query performance. This can be done using the rewriteDataFiles action.

Docs -> https://iceberg.apache.org/docs/latest/spark-procedures/#rewrite_data_files

Q.) What are the rewrite/clustering strategies available for compacting your data files?

A.) Iceberg allows you to perform rewrites with the rrewrite_data_files procedure and it supports two strategies: bin-pack or sort. The default is bin-pack. Using sort allows you to cluster your data based on one or more fields to group like data together for more efficient data scans. Iceberg also supports z-order clustering.

Blog ->
https://www.dremio.com/blog/compaction-in-apache-iceberg-fine-tuning-your-iceberg-tables-data-files/
Blog ->
https://www.dremio.com/blog/how-z-ordering-in-apache-iceberg-helps-improve-performance/

Q.) Do I need to retain every metadata file?

A.) A new metadata file is created on every CREATE/INSERT/UPDATE/DELETE and can build up over time. You can set a max number of metadata files so that it deletes the oldest metadata file whenever a new metadata file is created.

Docs -> https://iceberg.apache.org/docs/latest/maintenance/#remove-old-metadata-files

Q.) How do I automatically clean metadata files?

A.) To automatically clean metadata files, set write.metadata.delete-after-commit.enabled=true in table properties. This will keep some metadata files (up to

write.metadata.previous-versions-max) and will delete the oldest metadata file after each new one is created.

Q.) Explain the core metadata architecture of an Iceberg table and how it enables atomic commits and snapshot isolation.

A.) At its heart, Iceberg separates table metadata (the “table of contents”) from the actual data files. When you create a table, Iceberg initializes a metadata JSON (e.g. metadata.json) that points to a snapshot of the table. Each snapshot records:

- A list of manifest lists (JSON or Avro files)
- Table properties (schema, partition spec, sort order, timestamps)

Each manifest list then points to multiple manifest files, which themselves list data files along with file-level metrics (row counts, partition values, lower/upper bounds).

Atomic commits are achieved by writing new metadata files—never in-place updates. When you append, rewrite, or delete data, Iceberg writes new manifests (and possibly a new manifest list) and then creates a brand-new metadata JSON that points to the updated snapshot. Finally, it atomically replaces the old metadata.json with the new one (e.g. via an atomic rename).

Because readers always start from the latest metadata.json, they either see the entire old snapshot or the entire new one—never a mix—guaranteeing snapshot isolation. Time travel is simply “read from metadata as of snapshot N,” and concurrent writers coordinate through optimistic concurrency on the metadata file itself.

Q.) How does Iceberg support schema and partition evolution without rewrites of existing data files?

A.) Schema evolution in Iceberg is managed via the table’s metadata JSON, which maintains a schema version history. Operations like ADD/DROP/RENAME COLUMN update the metadata but do not touch the underlying Parquet/ORC files. At read time, the Iceberg reader:

- Projects only the columns present in the file
- Fills missing new columns with null/default values
- Ignores dropped columns

This lets you add or deprecate fields on the fly.

Partition evolution works similarly. Iceberg defines a partition spec (e.g. year = year(order_date), bucket(16, user_id)) stored in metadata. When you change the spec—say, switching from year to (year, month)—Iceberg writes a new spec ID in metadata. It does not reorganize old data files; instead, the engine:

- Uses the partition transform logic of the spec that originally wrote each file
- Applies filtering and pruning based on the old spec’s recorded partition values stored in the manifest

This hidden partitioning means you can evolve how you slice data without rewriting the entire table. Only when you explicitly run a rewriteDataFiles job (for optimization) do old files get reorganized.

Q.) Describe how Iceberg uses file-level statistics to enable efficient partition pruning and predicate pushdown.

A.) Each manifest file in Iceberg contains entries for its data files, and for each file Iceberg records lightweight statistics:

- Partition values for each partition field
- Lower/upper bounds for each column (min/max)
- Null counts, row counts, and sometimes value counts

When you run a query with filter predicates (e.g. WHERE order_date BETWEEN '2024-01-01' AND '2024-01-31' AND total_amount > 0), Iceberg's planner:

1. Manifest-level pruning: Scans manifest files to drop entire data files whose partition values don't overlap the filter range—no need to open the file.
2. Metrics-based pruning: Reads file-level min/max stats to skip files where the predicate can never be true (e.g. max(date) < '2024-01-01').

This two-stage pruning drastically reduces I/O. Unlike Hive's static partitions, Iceberg's file-level metadata means every file benefits from predicate pushdown, even if unpartitioned by that column.

Q.) What isolation levels and concurrent-write guarantees does Iceberg provide, and how are write conflicts resolved?

A.) Iceberg enforces snapshot isolation for readers and optimistic concurrency for writers. When a writer begins a commit (append, overwrite, delete), it:

1. Reads the current metadata JSON to get the base snapshot ID.
2. Generates new manifests/metadata for its changes.
3. Attempts to atomically replace the metadata JSON file.

If another writer has already replaced the metadata JSON in the meantime, the atomic rename fails. At that point, the writer's job is to refresh: it reloads the latest metadata, reapplies its changes against the new snapshot (recomputing manifests if necessary), and retries the commit. This ensures:

- No lost updates: each writer sees intervening commits and merges them with its own.
- Linearizability: commits appear in a total order as they mutate metadata.
- Isolation: readers never see partial writes—only complete snapshots.

By combining metadata-file-level atomic renames with retry logic, Iceberg offers robust concurrency guarantees without heavyweight locking or coordination services.