A close-up of a glacier

Description automatically generated

**Tip of the (Db2) Iceberg**By: Liam Finnie

Data Lake tables provide an easy way to exchange data within your business - they are typically stored in an Open Data Format (ODF) on Object Storage such as IBM Cloud Object Storage or Amazon S3. Db2 Warehouse enables [querying of Data Lake tables](https://www.ibm.com/docs/en/db2/12.1?topic=121-using-datalake-tables-in-db2-warehouse) combined with your other Db2 tables from a single database engine. This integration of all forms of data is a strategic direction for Db2, and so Db2 has been steadily increasing functionality and usability of Data Lake tables. Since Data Lake tables are stored in an ODF and not in a proprietary format, that same data can be accessed and/or operated upon by different database engines. This data could also be generated from a different business unit, or originate from older data archived from a transactional database, or it could even be publicly available data that you want to combine with proprietary data from your Db2 data warehouse to use for a new application.

We have already started a blog series on Data Lake tables, showing how to [work with them in your Db2 Warehouse](https://medium.com/@ausclifford/multi-engine-interoperability-in-a-data-lakehouse-4ed9819dd296) and how to leverage a [cache of MQTs in your Db2 Warehouse to speed up analytical queries over Data Lake tables](https://community.ibm.com/community/user/datamanagement/blogs/christian-garcia-arellano/2024/03/25/accelerating-your-datalake-tables-with-a-cache-of). In this blog, we dig deeper into what happens under the covers and discuss some of the cool functionality that we have added since Data Lake tables were introduced. So if you are interested in getting your understanding of Data Lake tables in Db2 Warehouse to the next level, continue on!

We’re going to focus on Iceberg tables, which is one particular specialization of Data Lake tables. Why Iceberg tables? [Iceberg](https://iceberg.apache.org) itself is really just a specification – it describes how the table metadata should be laid out and operated upon. It’s up to individual database vendors to implement the functionality described by the Iceberg specification. Since the table specification entirely concerns the on-disk representation of tables, there are no hard requirements on preferred Catalog services to control or manage access to Iceberg tables as well, which should make them easier to integrate into diverse environments. The Iceberg community is also very active – there are many adopters of the Iceberg specification, and new versions of the specification are in development to continuously add new features demanded by customers. One other important detail that makes Iceberg tables very interesting for database environments is that the Iceberg specification supports transactions and ensures ACID compliance, meaning changes to Iceberg tables are transactionally consistent. It is important to note that although the Iceberg specification ensures ACID compliance, operations on Iceberg tables are outside of Db2 transactional control. Whenever a transactional change is applied to an Iceberg table, a new Iceberg snapshot is created to represent the new state and contents of the table.

**What’s in a Snapshot?**

Unlike Db2 tables, Iceberg tables do not use write-ahead logging. Instead, the Iceberg format relies on the ability of the underlying storage to perform an atomic file swap operation. Whenever there is a transactional operation on an Iceberg table, a new ‘snapshot’ of the table is generated, and the transaction performing this change is responsible for writing the on-disk information describing the new snapshot. Older snapshots are not affected – rather when a transactional change is applied to the table, the delta changes and metadata describing the new snapshot are first written out, and then the existing metadata file is atomically swapped with the new table metadata file. This also means that ‘Time Travel’ is inherently baked into the Iceberg specification, any query can report on contents of an Iceberg table at any point in time if the desired ‘snapshot’ from that point in time is still available on disk. Another simplifying factor is that Iceberg transactions are more constrained – although Db2 tables support transactions containing multiple insert, update, or delete statements on the same and/or different tables, Db2 restricts each Iceberg table transaction to a single statement. This is important to keep in mind if you intend to include an insert or delete statement against an Iceberg table within a Db2 transaction, since that Iceberg insert or delete statement will end up being treated like a separate transaction, and rolling back your overall Db2 transaction will not roll back the changes applied to the Iceberg table. This also means you cannot easily preserve integrity between two separate Iceberg tables – each table is transactionally independent, there is no 2-phase commit protocol available to ensure updates to multiple Iceberg tables are applied atomically.

Since Iceberg is a type of ODF table, any database engine claiming support for Iceberg tables must follow the Iceberg specification. As long as they all follow the specification properly, it’s entirely reasonable for one database engine to be creating a new ‘snapshot’ of an Iceberg table at the same time another database engine is running queries against older snapshots of that same table. Or, for a new query starting from a different database engine to immediately pick up changes applied in a new table snapshot. When operating with Iceberg tables in a Db2 Data Warehouse, queries will by default be working on the newest Iceberg table snapshot at the time the table was first referenced in the transaction.

**Inserts and deletes are all about Snapshots!**

Inserts into and deletes from Iceberg tables are a little more involved, so let’s dive into some details and see how Iceberg tables differ from what you may be used to when using Db2 Warehouse. We’ll start by looking at what happens to the Iceberg table on disk when new data is inserted. Iceberg tables have two main types of files – metadata files and data files. The metadata and related manifest files describe the table's schema, current snapshot, and which data files are included in each snapshot. The data files contain the actual table data – i.e. the rows in your table – and are typically stored in a columnar format, such as Parquet or ORC. Since each table modification creates a new snapshot, and old snapshots should not be affected by the new snapshot, inserting a new row means a new data file is created to contain the row contents. The metadata will also be updated to assign a new snapshot, and the manifest updated to show that the ‘new’ snapshot includes the new data file (for the newly inserted row), as well as all data files included in the previous snapshot.

So, if snapshots can be used to query data from previous points in time, how are deletes handled? Deleting a row from an existing data file would invalidate earlier snapshots, so another approach is needed. Version 1 of the Iceberg table specification did not describe how deletes should be handled at all. This support first appeared in version 2 of the Iceberg table specification version – and version 3 of the spec promises more goodies around efficient handling of deletes, so DELETE support for Iceberg tables is still a relatively new and evolving area. From the version 2 specification, 3 approaches for deleting data are available. The first approach for deleting data is often referred to as ‘copy-on-write', where an existing data file is re-written into a new data file, omitting the deleted row (or rows). This approach is useful when very large numbers of rows are deleted at once. The two other approaches are referred to as ‘merge-on-read' approaches and make use of a special type of data file called a ‘delete file’ to describe which rows have been deleted. The second delete approach uses ‘positional delete’ files, where the delete file specifies exactly which row or rows should be marked as deleted and ignored by future table scans. This approach is useful when deleting a small number of rows, since it means the DELETE operation does not need to re-create possibly huge data files if only a very small number of rows in those affected data files have been deleted. The third delete approach uses ‘equality delete files’, where the delete file essentially contains a ‘delete predicate’, where at query time the delete predicate would need to be evaluated against the corresponding data files to filter out any records that match the delete predicate. Db2 Warehouse by default uses the second approach, where positional delete files are created to specify exactly which rows have been deleted in a particular table snapshot. This default was chosen as a compromise between delete performance and query performance – deletes are reasonably fast and efficient since existing data files do not need to be re-written. And while future queries will need to read the delete files to exclude deleted rows, they will not need to evaluate any ‘delete predicates’ on all other valid data rows.

That covers only the basics for insert and delete operations for Iceberg tables, but we can already see they are handled very differently than what you may be used to with Db2 Warehouse. If this is your first introduction to the Iceberg world, you might be wondering how all these Iceberg files are managed when dealing with concurrent workloads.

**Concurrent Inserts?**

First, let’s think about how concurrent inserts can be handled for Iceberg tables. The Iceberg specification does not provide support for unique indexes, so if two different insert transactions are inserting data, even if they contain the same contents, both transactions are allowed. This makes concurrent inserts straightforward – the 1st one to commit will create the 1st new snapshot that includes the 1st new row (or rows), and the 2nd one to commit will create the 2nd new snapshot, where this 2nd new snapshot includes the new row (or rows) from the 1st snapshot as well as the new row (or rows) from the 2nd transaction. There are internal details that must be handled to make sure the Iceberg table metadata for the 2nd new snapshot includes the new rows from transactions that have committed (even if that commit was only a fraction of a second earlier), but there are no possible unique key violations that would prevent the 2nd insert transaction from committing. Due to the lack of write-ahead logging, the extra snapshot file concurrency required at transaction commit time will limit insert concurrency – so it’s preferable to have fewer transactions inserting multiple rows than to have singleton insert transactions.

A diagram of a diagram of a computer

Description automatically generated

Let’s look a little more closely at what happens when two concurrent inserts are trying to insert 1 new row each. The starting snapshot is S1, and the first insert transaction to commit T1 creates a new snapshot S2 that includes the original data as well as the new row. T1 is now done! At the same time though, T2 was also trying to create snapshot S2 that only includes the original data from S1 as well as its new row. That was rejected though because the commit was based off the old snapshot S1 which is no longer the most current. Instead, T2 will automatically retry on the latest snapshot S2, and so now attempts to create new snapshot S3 that includes the original data from S1 as well as the new data from S2. This retry succeeds, and now the new snapshot S3 is the most current snapshot and includes all data files from the original snapshot S1 as well as the new rows inserted by T1 as well as T2.

**Concurrent Deletes – Now the Real Fun Begins!**

Delete operations are much more complex and knowing how delete operations are handled for Iceberg is important when designing applications that operate on Iceberg tables. Similar to when there are multiple concurrent insert transactions, there are no inherent issues if two concurrent delete transactions are both committing around the same time, even if they are both attempting to delete the same rows. Consider if one delete transaction is ‘delete from myIcebergTable where C1 < 100’, and the other is ‘delete from myIcebergTable where C1 < 200’. In either case, regardless of how those two delete transactions are ordered, all rows ‘where C1 < 200’ will be deleted after both transactions are committed.

What happens if you are deleting rows in one transaction at the same time you are inserting rows in another transaction? This is where the real fun begins! To understand what happens, we need to first understand how a DELETE statement is actually performed. There are really two internal phases for a DELETE operation, the first phase performs a table scan that applies all the DELETE predicates to generate the target set of rows that will be deleted, and the second phase then deletes those target rows and commits the changes by creating a new snapshot. Remember how we talked about how Iceberg manages all transactions as snapshots? When the first phase of the delete operation starts (the table scan), it will use the most recent table snapshot to generate the set of rows to be deleted. However, nothing is stopping another concurrent insert operation from committing and generating a new snapshot before we get to the second phase of the DELETE operation and create our new snapshot.

A diagram of a diagram of a conflict

Description automatically generated

Let’s work through a concurrent INSERT/DELETE scenario to hopefully make these concepts much more clear. Both the INSERT transaction T1 and the DELETE transaction T2 start around the same time using the most recent Iceberg table snapshot S1. The INSERT transaction T1 then commits a new row, creating a new snapshot S2. Our DELETE statement has meanwhile scanned the rows from S1 and determined which rows need to be deleted. It is now ready to write the new DELETE file describing those deleted rows and commit to create a new snapshot. Since our DELETE transaction’s scan was based on S1, it then tries to create a new snapshot S2, however S2 already exists! No problem, we can just create snapshot S3, right? Not so fast... What happens if the newly inserted row in snapshot S2 matches the DELETE predicates – should it be deleted or not? The Iceberg commit and snapshot protocol is used to resolve this conflict – the initial DELETE commit is internally failed since the initial scan was based off an old snapshot S1. An internal retry is initiated to see if the DELETE changes can simply be applied on top of the new current snapshot S2 without requiring a new DELETE scan on the entire table snapshot S2. This is done by comparing the DELETE predicates to the newly inserted row(s), and if the DELETE predicates exclude all new rows inserted since the DELETE scan of snapshot S1, then the DELETE is allowed to commit, and will create a new snapshot S3. However, if the DELETE predicates cannot exclude all newly inserted rows (e.g., at least one of the newly inserted rows would have been deleted, had the newly-inserted rows existed in the original DELETE snapshot scan), there are two approaches that can be taken:

1. Start a new DELETE scan using the new snapshot S2, where the newly-inserted row will match the DELETE predicates and so be included in the list of rows to be deleted. Then, try to commit again, hoping that no more eligible new rows have been inserted in the meantime.
2. Return an error indicating there is a write-write conflict.

Db2’s Iceberg support takes this second approach – a SQL0913N error is returned that indicates the DELETE operation could not be completed due to a conflict with another application. This is a very reasonable approach – since the conflicting rows being inserted could have even come from a different database vendor running on different machines (remember, Iceberg tables can be accessed simultaneously by multiple database providers, and data is typically stored on shared cloud object storage). This means you now have the power – you can decide to simply re-issue the same request and hope there are no more conflicting new rows being inserted, or you can decide that there might be a bigger issue, and try to figure out why different applications are trying to generate different versions of the ‘truth’ in the first place.

**Top Tips for Db2 Warehouse Operational Iceberg Workloads**

Hopefully you’re now a little more familiar with some of the key differences between Iceberg and standard Db2 tables. Getting the most out of your Iceberg investment requires some regular Iceberg table maintenance you may not be familiar with coming from a Db2 warehouse background. Here are a few important tips that may be helpful – and be sure to check out the product docs for guidance on how to implement these tips in your environment.

1. Partition! Iceberg partitioning is versatile and powerful – it can speed up queries, as well as reduce DELETE conflicts if data is typically inserted into different partitions than those where data is deleted from. Each partition maintains separate data files, so choose partitioning wisely – for example, do not partition directly on a timestamp column, instead you may want to partition by day or month.

2. Batched operations! Since each INSERT or DELETE statement will generate a new table snapshot as well as new files, query performance will start degrading as the number of active snapshots and small files owned by the table increases. Instead, you may want to group inserted or deleted contents into a separate staging area and perform a batch INSERT or DELETE operation to reduce the number of files generated.

3. Compact your tables! To maintain good performance, it’s best to avoid having many small data files. Regular maintenance on your Iceberg table should include compacting smaller data files into larger ones - particularly if you cannot batch insert or delete operations.

4. Expire old snapshots! Even if you’ve compacted your tables to combine multiple small data files into larger ones, those small data files can not actually be removed if old snapshots still reference them. Regularly expiring old snapshots allows you to clean up these old data files that may be cluttering up your storage system.

What’s your favourite tip to improve operational efficiency with Iceberg tables?

**About the Author**

Liam joined IBM in 1999 as a software developer, and since then has been designing and implementing various features in Db2 LUW. Most recently, Liam has been leading the Machine Learning Infrastructure area for the Db2 AI Query Optimizer.