PIZENITH

SOFTWARE DEVELOPMENT BEST PRACTICES

Jan 2025

Version 1.0

Best practices: Delta Lake

Prepared For

Public

Prepared By

Pizenith Technologies LLP

Jan 20, 2025

General Use Case

INDEX

Best practices: Delta Lake On DataBricks

Delta Lake is an enhanced storage layer that serves as the backbone for tables within a lakehouse architecture on Databricks. As an open-source project, Delta Lake extends Parquet files by adding a file-based transaction log, enabling ACID transactions and efficient metadata management. It integrates seamlessly with Apache Spark APIs and is designed to work closely with Structured Streaming, allowing the use of a single data copy for both batch and streaming operations, while offering scalable incremental processing.

On Databricks, Delta Lake is the default format for all tables, unless specified otherwise. All Databricks tables are, by default, Delta tables. The Delta Lake protocol was originally created by Databricks and continues to be actively maintained within the open-source community. Many of the enhancements and features available on the Databricks platform leverage the guarantees offered by Apache Spark and Delta Lake. For further details on optimization techniques within Databricks, refer to the platform's optimization guidelines.

Remove Legacy Delta Configurations

- [ ] Databricks advises eliminating most legacy Delta configurations from both Spark settings and table properties when upgrading to a newer Databricks Runtime version. Retaining these old configurations may block the application of new optimizations and default settings introduced by Databricks, potentially affecting migrated workloads.

Use liquid clustering for optimized data skipping

- [ ] Databricks suggests using liquid clustering instead of partitioning, Z-ordering, or other data organization techniques to enhance data layout for efficient data skipping. For more details, refer to the guide on using liquid clustering for Delta tables.

Replace the content or schema of a table

There may be situations where you need to replace a Delta table, such as when you discover incorrect data and need to update the content, or when making incompatible schema changes like altering column types. Although it is possible to delete the entire directory of a Delta table and create a new one at the same path, this approach is not recommended due to the following reasons:

- Deleting a directory can be time-consuming, especially if it contains large files, potentially taking hours or even days.

- Deleting files results in losing all content, making it difficult to recover if the wrong table is deleted.

- The directory deletion process is not atomic, so while the table is being deleted, a concurrent query could either fail or encounter a partial version of the table.

If you don't need to alter the schema, you can simply remove the data from a Delta table and insert or update the data to correct any issues. However, if schema changes are necessary, you can atomically replace the entire table.

For example:

```
dataframe.write \
  .mode("overwrite") \
  .option("overwriteSchema", "true") \
  .saveAsTable("<your-table>") # Managed table

dataframe.write \
  .mode("overwrite") \
  .option("overwriteSchema", "true") \
  .option("path", "<your-table-path>") \
  .saveAsTable("<your-table>") # External table
```

Spark caching

Databricks does not recommend that you use Spark caching for the following reasons:

- You lose any data skipping that can come from additional filters added on top of the cached DataFrame.
- The data that gets cached might not be updated if the table is accessed using a different identifier.

Improve performance for Delta Lake merge

To speed up the merge operation, you can use several strategies:

- **Reduce the search space for matches**: By default, the merge operation scans the entire Delta table to find matches in the source table. You can optimize this by adding known constraints to the match condition to limit the search space. For instance, if your table is partitioned by country and date, and you want to merge data for a specific country and date, adding conditions like `events.date = current_date() AND events.country = 'USA'` ensures the query only checks relevant partitions, speeding up the process. Additionally, this reduces the likelihood of conflicts with concurrent operations. For more on isolation levels and write conflicts, see Databricks documentation.

- **Compact files**: If the data is spread across many small files, searching for matches can be slow. Compacting smaller files into larger ones can improve read throughput. Refer to the section on optimizing data file layout for more details.

- **Control shuffle partitions for writes**: During a merge, data is shuffled multiple times to compute and write the updates. The number of shuffle tasks is controlled by the Spark configuration `spark.sql.shuffle.partitions`. Adjusting this parameter can affect both parallelism and the number of output files. Increasing the value boosts parallelism but also creates smaller files.

- **Enable optimized writes**: In partitioned tables, a merge can generate many small files, potentially leading to performance bottlenecks. Enabling optimized writes helps reduce the number of files created during the merge process. For more details, see the section on optimized writes for Delta Lake.

- **Tune file sizes in the table**: Databricks automatically detects if frequent merge operations are rewriting files and adjusts the size of rewritten files accordingly to improve future operations. Check the tuning file sizes section for more details.

- **Low Shuffle Merge**: Low Shuffle Merge is an optimized MERGE implementation that improves performance for most use cases. It preserves data layout optimizations, such as Z-ordering, for unmodified data, providing better overall performance.

Enhanced checkpoints for low-latency queries

Delta Lake creates checkpoints to store an aggregated state of a Delta table at an optimized frequency. These checkpoints serve as a reference point for computing the most up-to-date state of the table. Without checkpoints, Delta Lake would need to process numerous JSON files ("delta" files) that represent commits to the transaction log, which could be inefficient. Additionally, the column-level statistics used for data skipping are stored within the checkpoint, enhancing performance during queries.

Enable enhanced checkpoints for Structured Streaming queries

If your Structured Streaming workloads don't have low latency requirements (subminute latencies), you can enable enhanced checkpoints by running the following SQL command:

```
ALTER TABLE <table-name> SET TBLPROPERTIES
('delta.checkpoint.writeStatsAsStruct' = 'true')
```

You can also improve the checkpoint write latency by setting the following table properties:

```
ALTER TABLE <table-name> SET TBLPROPERTIES

(

 'delta.checkpoint.writeStatsAsStruct' = 'true',

 'delta.checkpoint.writeStatsAsJson' = 'false'
)
```

If data skipping is not useful in your application, you can set both properties to false. Then no statistics are collected or written. Databricks does not recommend this configuration.

For more information and read the original source check here