There are two approaches to handle deletes and updates in the data lakehouse: copy-on-write (COW) and merge-on-read (MOR).

Like with almost everything in computing, there isn’t a one-size-fits-all approach – each strategy has trade-offs that make it the better choice in certain situations. The considerations largely come down to latency on the read versus write side. These considerations aren't unique to Iceberg or data lakes in general, the same considerations and trade-offs exist in many other places, such as lambda architecture.

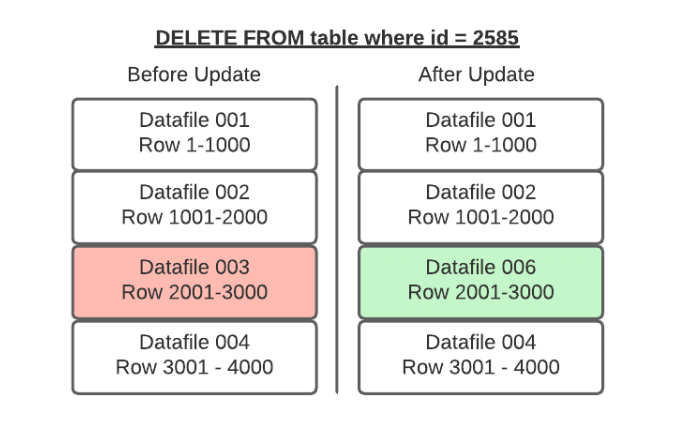
**Copy-On-Write (COW) – Best for tables with frequent reads, infrequent writes/updates, or large batch updates**

*Summary: During update & delete, entirely new data file is written that includes the changes*

With COW, when a change is made to delete or update a particular row or rows, the datafiles with those rows are duplicated, but the new version has the updated rows. This makes writes slower depending on how many data files must be re-written which can lead to concurrent writes having conflicts and potentially exceeding the number of reattempts and failing.

If updating a large number of rows, COW is ideal. However, if updating just a few rows you still have to rewrite the entire data file, making small or frequent changes expensive.

On the read side, COW is ideal as there is no additional data processing needed for reads – the read query has nice big files to read with high throughput.



**Merge-On-Read (MOR) – Best for tables with frequent writes/updates**

*Summary: During update & delete, new datafile is not written. Only delete file, which is a metadata file is written. At the time of read, original data file is merged with the delete file.*

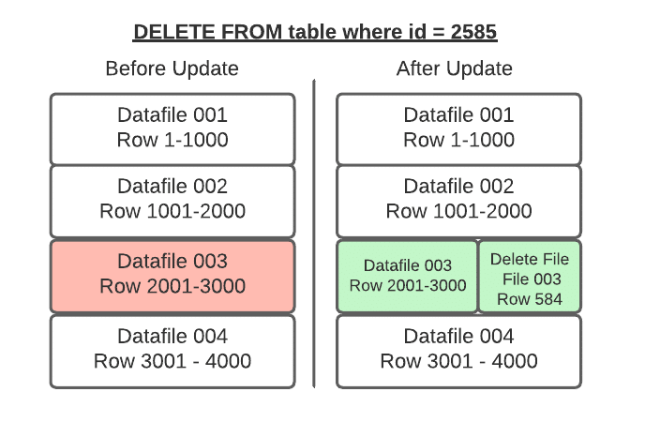
With merge-on-read, the file is not rewritten, instead the changes metadata are written to a new file. Then when the data is read, the changes are applied or merged to the original data file to form the new state of the data during processing. This makes writing the changes much quicker, but also means more work must be done when the data is read.

In Apache Iceberg tables, this pattern is implemented through the use of delete files that track updates to existing data files.

If you delete a row, it gets added to a delete file and reconciled on each subsequent read till the files undergo compaction which will rewrite all the data into new files that won’t require the need for the delete file.

If you update a row, that row is tracked via a delete file so future reads ignore it from the old data file and the updated row is added to a new data file. Again, once compaction is run, all the data will be in fewer data files and the delete files will no longer be needed.

So when a query is underway the changes listed in the delete files will be applied to the appropriate data files before executing the query.



*Position Deletes*

*Summary: During update & delete: Original Data file is read + delete file is written*

Position deletes still read files to determine which records are deleted, but instead of rewriting the data files after the read, it only writes a delete file that tracks the file and position in that file of records to be deleted. This strategy greatly reduces write times for updates and deletes, and there is a minor cost to merge the delete files at read time. The screenshot above is for this strategy.

*Equality Deletes:*

*Summary: During update & delete: Only delete file is written*

When using equality deletes, you save even more time during the write by avoiding reading any files at all. Instead, the delete file is written to include the fields and values that are targeted by the query. This makes update/delete writes much faster than using position deletes. However, there is a much higher cost on the read time since it will have to match the delete criteria against all scanned rows to reconcile at read, which can be quite costly.

