How delta lake computes the latest state

Created a table
000.json

inserting 5 records in a single statement
file1.parquet
001.json (add file1.parquet)

updating one of the record
dv file
file2.parquet
002.json (remove 1.parquet, add 1.parquet with dv, add file2.parquet)

select * from table;

000.json
001.json
002.json

add file1.parquet X
remove file1.parquet X
add file1.parquet with dv
add file2.parquet

Lets say you perform a overwrite

file3.parquet
003.json (remove file1.parquet with dv, remove file2.parquet, add file3.parquet)

select * from table;

add file1.parquet X
remove file1.parquet X
add file1.parquet with dv X
add file2.parquet X
remove file1.parquet with dv X
remove file2.parquet X

add file3.parquet

V1 - T1

T1.5

V2 - T2

V3 - T3

VACUUM
======

vacuum removes data files that are no longer referenced by a delta table version in the last 7 days.

Vacuum command is a maintainance command

predictive optimization (vacuum)

ALTER TABLE tablename SET TBLPROPERTIES ('delta.deletedFileRetentionDuration' = '30 days');

spark.conf.set("spark.databricks.delta.retentionDurationCheck.enabled","false")

Running vacuum regularly is important

- to save on the storage cost
- compliance reasons

vacuum might leave empty directories

folder1 - 1,2,3
folder2 - 4
folder3 - 5,6

when you run vacuum (dry run) file 4 to be deleted

vacuum

folder1 - 1,2,3
folder2 -
folder3 - 5,6

vacuum

folder1 - 1,2,3
folder3 - 5,6

Log files
==========

with multiple json files

the challenge is not more about storage cost

the challenge is about read time

100 parquet files

100 json files

select * from table;

If we have inserted 8 records

8 jsons would be created

1-6 (combining)

7

8

80th version

00000000000000000072.checkpoint.parquet
00000000000000000073.00000000000000000078.compacted.json
79.json
80.json

https://ttmystorageaccount001.blob.core.windows.net/externaldata/orders/_delta_log/00000000000000000073.00000000000000000078.compacted.json

SELECT * FROM TABLE;

00000000000000000072.checkpoint.parquet
00000000000000000073.00000000000000000078.compacted.json
00000000000000000079.00000000000000000084.compacted.json
00000000000000000085.00000000000000000090.compacted.json
00000000000000000091.00000000000000000096.compacted.json
97.JSON
98.JSON
99.JSON
100.JSON

delta.logRetentionDuration

parquet files
delta logs

delta.deletedFileRetentionDuration = "interval 30 days" (this is now matching the default setting for delta.logRetentionDuration)

Deletion Vectors
==========

Storage Setup
- create a volume with name my_demo_volume
- create a folder dv_demo inside that

Question 1) You are managing a large Delta table with billions of rows. Deleting even a few thousand rows used to be very expensive because it required rewriting files.
How do deletion vectors, help solve this problem?

Reference Notes -

With Deletion Vectors disabled
==========

add file 1 - 1 2 3 5 6 7 8
add file 2 - 9 10 11 12 13 14 15 16

Delete a record 12

add file 3 (9 10 11 13 14 15 16)
remove file 2

With Deletion Vectors Enabled
==========

add file 1 - 1 2 3 5 6 7 8
add file 2 - 9 10 11 12 13 14 15 16

Delete a record 12

remove file 2
add file 2 with deletion vector

1,2,3 - add file1

4,5,6 - add file2

delete record 4

remove file2

add file2 with deletion vector

Question 2) Imagine two cases:
Case A: Deleting 100 rows
Case B: Deleting 100 million rows
In which case do deletion vectors bring the most benefit, and when would you consider a file rewrite instead?

Reference Notes -

delta.enableDeletionVectors table property.

Question 3) Your company must comply with the GDPR (right to be forgotton). Since deletion vectors only mark rows as deleted, the data still exists in storage. How would you design a process to ensure deleted records are physically removed for compliance?

Reference Notes -

REORG TABLE events APPLY (PURGE); to rewrite files and drop deleted rows.

Optimize

Running VACUUM with an appropriate retention period to permanently remove old file versions.

This ensures that deleted customer data is not just hidden but physically removed from storage, satisfying compliance requirements.

Question 4) Over time, analysts complain that queries on the Delta table are becoming slower after frequent deletes. How would you confirm whether deletion vectors are the cause, and what steps can you take to restore performance?

Reference Notes -

Inspect the table metadata using DESCRIBE DETAIL or DESCRIBE HISTORY to check for a high count of deletion vectors.

If deletion vectors are the bottleneck, the fix is to periodically compact and apply those deletions physically. In Databricks, that means running REORG TABLE, This rewrites files.
and periodic Optimize commands will help.

Question 5) After months of deletes and updates, you notice storage costs have gone up and queries are slower, even though deletion vectors were meant to save space and time. Why does this happen, and what Databricks operations would you apply to balance storage, performance, and compliance?

Reference Notes -

This happens because deletion vectors are logical deletes — they reduce rewrite costs, but over time, the table accumulates a lot of “dead weight” in files. Queries must scan files full of deleted rows, and storage holds both active and inactive data.

To fix this, I would implement regular maintenance:

Run REORG TABLE ... to rewrite files without deletion vectors

Run OPTIMIZE to compact small files and improve query performance.

Run VACUUM to clean up old file versions and reclaim storage space.

==========

How to get rid of Deletion Vectors.

In the previous session you understood that if there are lot of deletion vectors then reads will be slower, even it will consume more storage.

when we run optimize

deletion vector file is small

REORG TABLE

APPLY (PURGE) only rewrites files that contain soft deleted data

APPLY (UPGRADE) may rewrite all the files

REORG TABLE is an idempotent operation, meaning that if we run it twice on the same dataset, the second run has no effect.

after running apply (purge), the soft deleted data may still exist in the old files.
you can run VACUUM to physically delete the old files.

Upsert / Merge
==========

DML

upsert = update + insert

incoming batch of data - (source)

3, ...
4, ...
5,.....

orders - (target)

1, ...
2, ...
3, ...

order id 3 should be updated

order id 4, 5 should be inserted

target table

and then incoming source data

=========

Constraints / ACID / Optimistic Concurrency control

order_id (NOT NULL)

qty, unit_price > 0 (check)

update (parquet file, deletion vector)
insert (parquet file)

ACID (atomicity, consistency, Isolation, Durability)

Isolation
- pessimistic concurrency control

row level locking

- optimistic concurrency control

version 3 of the table

writer1 - time t1 (version 4)

writer2 - time t2 (version 3)

==========

Schema Enforcement & Schema Evolution

Schema enforcement - is the process of strictly maintaining a predefined schema. rejecting any data that does not conform to it to ensure data quality and consistecy.

Schema evolution - is the flexible process of updating and adapting a dataset's schema to accomodate changes. such as adding new columns or modifying data types, without breaking existing data pipelines.

adding new columns - .option("mergeSchema", "true") \

data type widening - 'delta.enableTypeWidening' = 'true'

Rename a Column or Delete a Column

in parquet file you have data also it contains the column names

it created a new log file (metadata file)

column mapping

physical file qty
logical name quantity

azure databricks supports column mapping for delta lake tables, which enables metadata-only changes to mark columns as deleted or renamed without rewriting data files.