Apache Spark & Delta Lake Tips

- Lingeshwaran Kanniappan

1. Using `DRY RUN` prior to running `VACUUM` command to view the list of files

```
-- Suppose you are going to delete the table using VACUUM
-- We can view the files that are going to be vacuumed by running the command
VACUUM sampleTable RETAIN 192 HOURS DRY RUN -- view the files for the past 8 days

-- After you are satisfied with the result that was generated

-- vacuum the files
VACUUM sampleTable RETAIN 192 HOURS
```

One more point to note is that if you are trying to run for any period less than **167.5 hours**, with the default configuration the VACUUM will fail. For example, We get the following error if we run `VACUUM` with retain period of **167.4 hours**:

requirement failed: Are you sure you would like to vacuum files with such a low retention period? If you have writers that are currently writing to this table, there is a risk that you may corrupt the state of your Delta table.

To overcome this you can set the configuration (⚠️ It is NOT recommended to use VACUUM for less than 7 days unless absolutely necessary):
`SET spark.databricks.delta.retentionDurationCheck.enabled = false;`

and this configuration to enable logging:

`SET spark.databricks.delta.vacuum.logging.enabled = true;`

2. Avoid Using `VACUUM` with a horizon of less than one hour Data inconsistency can be a problem even for small tables

This is for similar reasons as mentioned in the previous point. We must be wary about the versions that will be deleted once we run a dangling VACUUM command. It is better first to clone the data if needed and check the data that is going to be removed.

3. Try to use `ANALYZE TABLE` command when you are running joins on multiple tables in a single query.

This can help the **Cost based optimizer** to kick in the query plan and optimize the complex joins and improve performance.

```
EXPLAIN <query here>
-- this should give you the statistics of the rowCount
-- if this is not displayed for a complex query there may be optimization inefficiencies
```

4. Try using the `text.`path`` format for reading raw files that are prone to corruption

```
SELECT * FROM text.`path/to/raw/file/with/corruption/sampleTable/`
```

This can help in first loading and getting a view of the rows before processing them further using some standard clean up techniques and then loading it into next table.

5. Enable Query Watchdog to watch over Ad-Hoc Analytics

Data analysts, Data Scientists and Data Engineers work on many complicated queries and can forget the impact of hidden long-running tasks. For instance, if there is an equi-join on two large tables (~ 1 million rows each) which has duplicate keys and it bypasses our attention, then there is going to be a cross join and will result in a long-running task.

Only once the task finishes - we will be able to identify this issue given we have not checked our input before. To avoid this we can use the following configuration that is provided by **Query Watchdog**.

```
-- Default values - DBR 7.3 LTS
-- increase or decrease the values based on the need/use case
SET spark.databricks.queryWatchdog.outputRatioThreshold = 10000L
SET spark.databricks.queryWatchdog.minTimeSecs = 300
SET spark.databricks.queryWatchdog.minOutputRows = 10000000L
```

Enable the `queryWatchdog` by setting the following configuration:

```
-- By default this is false
SET spark.databricks.queryWatchdog.enabled = true
```

`outputRatioThreshold`: It is the ratio of the output rows to the input rows until which the single task will be allowed to run.

`minTimeSecs`: It is the minimum amount of time until when a single task will be allowed to run

`minOutputRows`: It is the minimum number of output rows a single task will be allowed to generate.

Similarly, a cluster intensive table scan can be watched over by the following threshold settings in Spark SQL configurations

```
-- Default values
spark.databricks.queryWatchdog.maxHivePartitions = 10000
spark.databricks.queryWatchdog.maxQueryTasks = 10000
```

Increase or decrease these values based on the analysis performed on the data that is going to be used.

6. Use `GENERATED` columns instead of CTAS statements for custom schema tables

`CREATE TABLE AS SELECT` (CTAS) does not support manual schema specification. But, it is a very useful clause to construct tables from already existing tables.

In case of scenarios where we need columns which will always be calculated using a certain logic applied on another column, we can use `GENERATED` columns instead.

The `GENERATED` columns have the following characteristics:

- They don't have values provided during time of definition. If the value is provided there is `CHECK` constraint that must be satisfied, else it will fail with error.
- They only have the generation expression that is required to calculate them.
- It is available from **DBR 8.3+**.
- `MERGE` is supported since **DBR 9.1**.

```
-- Simple example, DBR 8.3+
CREATE TABLE default.sales_delta (
  orgName STRING,
  orgAddress STRING,
  sellerId STRING,
  salestimeStamp STRING,
  salesDate DATE GENERATED ALWAYS AS
               (CAST(CAST salestimeStamp/1e6 AS TIMESTAMP) AS DATE)),
  orgId STRING,
  salePrice INT
)
USING DELTA
PARTITIONED BY (orgName)
```

Here, the column `salesDate` is calculated from the `salestimeStamp` column of type `STRING` during the instance of data insert into the `sales_delta` table.

7. Use `VALIDATE` mode in `COPY INTO` commands for incremental loads with data that needs to be checked

From DBR 10.3, we have the option to use `VALIDATE` mode in `COPY INTO` commands. This gives the user to run a check while copying data from source to the delta table.

```
COPY INTO sales_for_2022
FROM dbfs:/user/sampleTables/sales_for_2022
FILEFORMAT = CSV
VALIDATE [ALL | <num_rows> ROWS]
```

The validation is done on three levels:

- Check if the data can be parsed.
- Check if the schema matches that of the table or if the schema needs to be evolved.
- Check if nullability and check constraints are met for all columns.

8. Avoid Spark Caching when you are using Delta Caching - don't mix them both

- Data skipping can be lost if there are additional filters added on top of the cached DataFrame.
- Spark cache uses the native `.cache()` method (**in-memory storage**) and it is a **lazy operation** which is compiled and executed later.
- Delta cache uses the (**disk storage**) automatically maintains the file consistency and detects any changes to the underlying data files and manages its cache.
- `"spark.databricks.io.cache.enabled"` is disabled by default, you can enable it and additionally use a delta accelerated worker nodes (Standard L Series on Azure and i3 instance on AWS) to get the pre-configured SSDs to cache effectively.
- Activate the delta cache if needed using the `CACHE SELECT` clause.
- Delta cache is **10 times faster** than the Spark cache.

```
-- How to delta cache a delta table?
CACHE SELECT * FROM frequentlyQueriedTable;
CACHE SELECT colOne, colTwo FROM frequentlyQueriedTable WHERE colOne > 0;
```

9. Managing the number of Execution Contexts & setting auto-eviction process

Execution context is a dedicated REPL environment for each supported programming language: Python, R, Scala, and SQL. When we run a cell in a notebook, the command is dispatched to the appropriate language REPL environment and run.

A cluster has a maximum number of execution contexts (**145**). Once the number of execution contexts has reached this threshold, you cannot attach a notebook to the cluster or create a new execution context. We see the following error

Can't attach this notebook because the cluster has reached the attached notebook limit. Detach a notebook and retry.

To prevent this from happening - **auto-eviction process** is enabled for us by default.

If it is disabled for some reason, you can enable it by setting this configuration property (`spark.databricks.chauffeur.enableIdleContextTracking=true`)

10. Run `DELETE` then `VACUUM` if you want to drop a large managed delta table

If you want to delete a large managed table with minimal impact on the time of execution - you can run the commands in the following order:

`DELETE FROM` → `VACUUM` (⚠️ the retain period has to be set properly) → `DROP TABLE`

The first two steps reduce the amount of metadata and number of uncommitted files that would otherwise increase the data deletion time.

11. Reduce the number of memory partitions on data which applies filters a large number of rows

When we apply a filter operation on a largely partitioned data which results in very few rows, it is always recommended to repartition the output data post filter. This will reduce the inefficiency that will be caused by leaving the data un-partitioned.

The correct number of partitions can be decided by following the heuristic:

Select the number of partitions to achieve 1 GB of data per memory partition.

```
import org.apache.spark.sql.functions._

val u = normalRDD(sc, 1000000L, 10)
val v = u.map(x => 1.0 + 2.0 * x).toDF("value").repartition(10000) // creating 10k partitions

// 1. Let's see the number of partitions
v.rdd.partitions.size // 10,000,000 rows sitting in 10k partitions

// 2. Applying Filter
v.filter("value > 8").count() // ~210 rows : this is 0.002% of the entire data

// 3. Checking the number of partitions on filtered data
v.filter("value > 4").rdd.partitions.size // 10k partitions [NO CHANGE!]
// there are lot of partitions that are empty
// hence it is always ideal to repartition the data
// when it is known that the filter will result in very few rows.
```

12. Using `()` in pySpark multi-line code if you want to avoid backslashes `\`

This might save you some keystrokes :)

```
-- The usual way of writing multi-line Pyspark statements
spark.read \
    .format("csv") \
    .schema("id STRING, name STRING") \
    .option("header", True) \
    .load(f"/FileStore/shared_uploads/info.csv") \
    .createOrReplaceTempView("sampleFile")
```

We can instead include the entire block of code in `()` to make use of lesser keystrokes.

```
-- The alternative to try as well
(spark.read
    .format("csv")
    .schema("id STRING, name STRING")
    .option("header", True)
    .load(f"/FileStore/shared_uploads/info.csv")
    .createOrReplaceTempView("sampleFile"))
```

13. Spark SQL also support Join Hints

We tend to use join hints to indicate the Spark engine about about the type of join we want to perform. In the Spark API we can do so, using the `broadcast` function for example. Similarly, we can provide proper hints in a vanilla SQL query too as shown below:

```
-- Join Hints for broadcast join
SELECT /*+ BROADCAST(smallTable) */ *
FROM smallTable
INNER JOIN largeTable
ON smallTable.key = largeTable.key;
```

The precedence of execution of the hint priority is as denoted below:

`BROADCAST` > `MERGE` > `SHUFFLE` > `SHUFFLE_REPLICATE_NL`

```
-- Join Hints with fallback information on join hints
-- It will try to perform BROADCAST on TableA and ignore the MERGE hint
-- due to the default precedence of execution of the hints
SELECT /*+ BROADCAST(tableA), MERGE(tableA, tableB) */ *
FROM tableA
INNER JOIN tableB
ON tableA.key = tableB.key;
```

14. Enable Spark Speculative Execution for Jobs that are struggling with long running tasks

Consider the situation where you are trying to figure out why there are tasks in a job which are lagging the overall execution time. These are “task strugglers” which can be marked for **speculation** in Spark which will in turn help in improving the runtime.

These are tasks that are born out of jobs which are long-running and where you have to kill and re-run the task to make it succeed. To avoid this hassle, we can leverage the use of Spark Speculative Execution.

You can enable this by setting the property `spark.speculation=true` which will in turn make Spark re-launch the long running tasks in a parallel thread and get the result from that if it completes faster.

We can set the following flags in addition to create a granular configuration:

Flag	Definition
`spark.speculation.multiplier`	The number of times the current task execution time is slower than the median time of task execution
`spark.speculation.interval`	How often Spark will check for the tasks on the speculation validity
`spark.speculation.quantile`	Percentage of tasks which must be completed before the speculation is enabled in a particular stage.

References:

[1] Databricks - Delta Lake
[2] Databricks - Delta Caching
[3] Spark Docs
[4] Databricks - Change Data Feed
[5] Databricks - Delta FAQ
[6] Tuning Guide

Feel free to connect with me!

Lingeshwaran Kanniappan - Consultant - Capgemini | LinkedIn

Lingesh is a talented data engineer with good technical skills. He strives to complete the work with quality and uses his initiative to introduce innovation and efficiencies. He is professional, has interests and proven capability in Python, Spark, Scala, Azure services and various cloud developer tools.

https://www.linkedin.com/in/lingeshwarankanniappan/