OPTIMIZING DATABRICKS

Shwetank Singh
GritSetGrow - GSGLearn.com

Optimizing Databricks

Optimize & Z-order for Delta Tables

To compact multiple files together

OPTIMIZE compacts the files to get a file size of up to **1GB**, which is configurable.

Like Defragmentation in Databases

OPTIMIZE table_name [WHERE predicate]
[ZORDER BY (col_name1 [, ...])]

Like Indexing in Databases

Z-Ordering is a technique to **co-locate** related information in the same set of files.

Given a column that you want to perform ZORDER on, say OrderColumn, Delta

- Takes existing parquet files within a partition.
- Maps the rows within the parquet files according to OrderColumn using the Z-order curve algorithm.
- In the case of only one column, the mapping above becomes a linear sort
- Rewrites the sorted data into new parquet files.

Shwetank Singh
GritSetGrow - GSGLearn.com

Optimizing Databricks

Auto Optimize in Delta Table

Automatically compacts small files during individual writes to a Delta table.

Optimize Write

Use when not using OPTIMIZE manually

It dynamically optimizes Apache Spark partition sizes based on the actual data, and attempts to write out 128MB files for each table partition. It's done inside the same Spark job.

Auto Compact

Following the completion of the Spark job, Auto Compact launches a new job to see if it can further compress files to attain a 128MB file size.

-- In Spark session conf for all new tables set
spark.databricks.delta.properties.defaults.autoOptimize.optimizeWrite = true
set spark.databricks.delta.properties.defaults.autoOptimize.autoCompact = true

-- Table properties
delta.autoOptimize.optimizeWrite = true
delta.autoOptimize.autoCompact = true

Shwetank Singh
GritSetGrow - GSGLearn.com

Optimizing Databricks

Partitioning

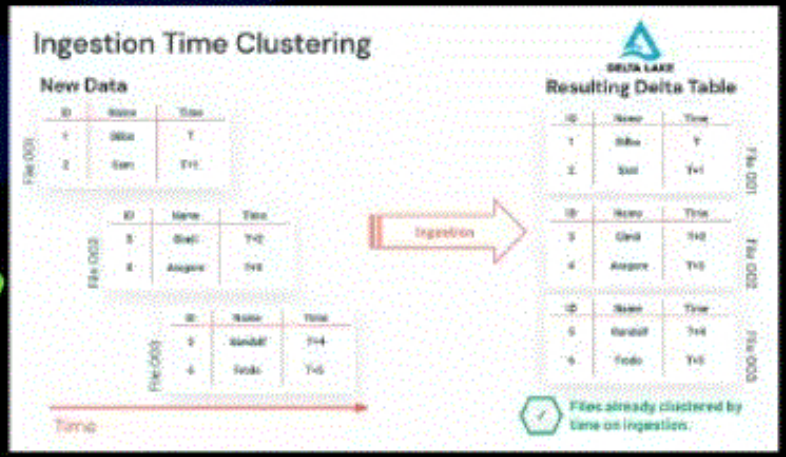
Partitioning can speed up your queries if you provide the partition column(s) as filters or join on partition column(s) or aggregate on partition column(s) or merge on partition column(s), as it will help Spark to skip a lot of unnecessary data partition (i.e., subfolders) during scan time.

PARTITIONED BY ({ partition_column [column_type] } [, ...])

useful when table size is 1 TB+.

For smaller than 1 TB tables let Ingestion Time Clustering do it's work.

The data as coming in the batch is stored in its own partition.



Shwetank Singh
GritSetGrow - GSGLearn.com

Optimizing Databricks

Broadcast hash join

To entirely avoid data shuffling, broadcast one of the two tables or DataFrames (*the smaller one*) that are being joined together. The table is broadcast by the driver, who copies it to all worker nodes.

set spark.sql.autoBroadcastJoinThreshold = <size in bytes>

default is 10 MB

In Spark 3.0 the sort-merge is change to broadcast join by AQE, if stats of any table is less than 30 MB

set spark.databricks.adaptive.autoBroadcastJoinThreshold = <size in bytes>

default can be changed

Shwetank Singh
GritSetGrow - GSGLearn.com

Optimizing Databricks

Leverage cost-based optimizer

Spark SQL can use a Cost-based optimizer (CBO) to improve query plans. For it to work, it is critical to collect table and column statistics and keep them up to date. Based on the stats, CBO chooses the most economical join strategy.

ANALYZE TABLE table_name COMPUTE STATISTICS FROM COLUMNS col1, col2, ...;

Adapative Query Execution (AQE) also leverages it

ANALYZE TABLE command needs to be executed regularly (preferably once per day or when data has mutated by more than 10%, whichever happens first)

Shwetank Singh
GritSetGrow - GSGLearn.com

Optimizing Databricks

Shuffle hash join over sort-merge join

In most cases Spark chooses sort-merge join (SMJ) when it can't broadcast tables. Sort-merge joins are the most expensive ones. Shuffle-hash join (SHJ) has been found to be faster in some circumstances (but not all) than sort-merge since it does not require an extra sorting step like SMJ.

set spark.sql.join.preferSortMergeJoin = false

Spark will try to use SHJ instead of SMJ wherever possible

Shwetank Singh
GritSetGrow - GSGLearn.com

Optimizing Databricks

AQE auto-tuning

Spark AQE has a feature called autoOptimizeShuffle (AOS), which can automatically find the right number of shuffle partitions

set spark.sql.shuffle.partitions=auto

The default setting for the number of Spark SQL shuffle partitions (i.e., the number of CPU cores used to perform wide transformations such as joins, aggregations and so on) is **200**, which isn't always the best value.

As a result, each Spark task (or CPU core) is given a large amount of data to process, and if the memory available to each core is insufficient to fit all of that data, some of it is spilled to disk.

Spilling to disk is a costly operation, as it involves data serialization, de-serialization, reading and writing to disk, etc.

Shwetank Singh
GritSetGrow - GSGLearn.com

Optimizing Databricks

Manually fine tune Partitions

As a rule of thumb, we need to make sure that after tuning the number of shuffle partitions, **each task should approximately be processing 128MB to 200MB of data.**

Let's assume that:
Total number of total worker cores in cluster = T
Total amount of data being shuffled in shuffle stage (in megabytes) = B
Optimal size of data to be processed per task (in megabytes) = 128
Hence the multiplication factor (M): M = ceiling(B / 128 / T)
And the number of shuffle partitions (N): N = M x T
Note that we have used the ceiling function here to ensure that all the cluster cores are fully engaged till the very last execution cycle.

OR

```
-- in SQL
set spark.sql.shuffle.partitions = 2*<number of total worker cores in cluster>
-- in PySpark
spark.conf.set("spark.sql.shuffle.partitions", 2*<number of total worker cores in cluster>)
-- or
spark.conf.set("spark.sql.shuffle.partitions", 2*sc.defaultParallelism)
```

Shwetank Singh
GritSetGrow - GSGLearn.com

Optimizing Databricks

Fixing Skewed Data

Filter Skewed data

If it's possible to filter out the values around which there is a skew, then that will easily solve the issue. If you join using a column with a lot of null values, for example, you'll have data skew. In this scenario, filtering out the null values will resolve the issue.

Skew Hints

In the case where you are able to identify the table, the column, and preferably also the values that are causing data skew, then you can explicitly tell Spark about it using skew hints so that Spark can try to resolve it for you.

SELECT /+ SKEW('table', 'column_name', (value1, value2)) */ * FROM table*

Shwetank Singh
GritSetGrow - GSGLearn.com

Optimizing Databricks

Fixing Skewed Data Continued

AQE skew optimization

By default any partition that has at least 256MB of data and is at least 5 times bigger in size than the average partition size will be considered as a skewed partition by AQE.

set spark.sql.adaptive.skewJoin.enabled = false

Salting

It's a strategy for breaking a large skewed partition into smaller partitions by appending random integers as suffixes to skewed column values.

Shwetank Singh
GritSetGrow - GSGLearn.com

Optimizing Databricks

Delta data skipping

Delta data skipping automatically collects the stats (min, max, etc.) for the first 32 columns for each underlying Parquet file when you write data into a Delta table. Databricks takes advantage of this information (minimum and maximum values) at query time to skip unnecessary files in order to speed up the queries.

delta.dataSkippingNumIndexedCols = <value>

Change the default value of 32

ALTER TABLE table_name ALTER [COLUMN] col_name col_name data_type [COMMENT col_comment] [FIRST|AFTER colA_name]

Move larger columns to after the last column for which Databricks is going to collect the stats information

Shwetank Singh
GritSetGrow - GSGLearn.com

Optimizing Databricks

Data Skipping and Pruning

Column pruning

Select only those columns that are truly part of the workload computation and are needed by downstream queries.

```
-- SQL
SELECT col1, col2, .. coln FROM table
-- PySpark
dataframe = spark.table("table").select("col1", "col2", ... "coln")
```

Predicate pushdown

Pushing down the filtering to the "bare metal" — i.e., a data source engine. Predicate pushdown is data source engine dependent. It works for data sources like Parquet, Delta, Cassandra, JDBC, etc., but it will not work for data sources like text, JSON, XML, etc.

```
-- SQL
SELECT col1, col2 .. coln FROM table WHERE col1 = <value>
-- PySpark
dataframe = spark.table("table").select("col1", "col2", ... "coln").filter(col("col1") = <value>)
```

Shwetank Singh
GritSetGrow - GSGLearn.com

Optimizing Databricks

Data Skipping and Pruning

Partition pruning

The partition elimination technique allows optimizing performance when reading folders from the corresponding file system so that the desired files only in the specified partition can be read.

To leverage partition pruning, all you have to do is provide a filter on the column(s) being used as table partition(s).

Dynamic partition pruning and Dynamic file pruning

From Spark 3.0 onwards prune the partitions/files the join reads from a fact table by identifying those partitions that result from filtering the dimension tables.

Shwetank Singh
GritSetGrow - GSGLearn.com

Optimizing Databricks

Data Caching

Delta Caching or Disk Caching

The Delta cache accelerates data reads by creating copies of remote files in nodes' local storage (SSD drives) using a fast intermediate data format.

set spark.databricks.io.cache.enabled = true

Spark cache

Using cache() and persist() methods, Spark provides an optimization mechanism to cache the intermediate computation of a Spark DataFrame so they can be reused in subsequent actions. Similarly, you can also cache a table using the CACHE TABLE command.

Shwetank Singh
GritSetGrow - GSGLearn.com

Optimizing Databricks

Disk Cache vs Spark Cache

Feature	disk cache	Apache Spark cache
Stored as	Local files on a worker node.	In-memory blocks, but it depends on storage level.
Applied to	Any Parquet table stored on ABFS and other file systems.	Any DataFrame or RDD.
Triggered	Automatically, on the first read (if cache is enabled).	Manually, requires code changes.
Evaluated	Lazily.	Lazily.
Availability	Can be enabled or disabled with configuration flags, enabled by default on certain node types.	Always available.
Evicted	Automatically in LRU fashion or on any file change, manually when restarting a cluster.	Automatically in LRU fashion, manually with unpersist.

Shwetank Singh
GritSetGrow - GSGLearn.com