Optimizing Impala Performance

Intro

- In traditional database systems, indices are crucial for query speed.
- For data warehouses, it is sometimes not ideal because of maintenance overhead. Sometimes it is simply faster to do full table scan.
- Impala ignores this problem by not having indices at all!
  - Index maintenance would be very expensive due to the volume of data.
- Biggest I/O savings in Impala come from two sources:
  - Appropriate file format.
  - Partitioning.

Query Performance

- The most complex and resource-intensive queries usually involve join oerations.
- A critical factor is to collect statistics for all tables involved in the join.

Statistics

- Collecting statistics gives Impala information about the size of the table, number of distinct values, etc that can be used to find the cheapest execution plan (less resource-intensive).
- No updated statistics usually turns into suboptimal execution plans.
- `COMPUTE STATS` statements should be issued whenever table changes by a 30% or more.
- For large partitioned tables, `COMPUTE INCREMENTAL STATS` helps limiting the scanning to only new or changed partitions.

Memory usage

- Efficient use of memory is important for overall performance and scalability in a highly concurrent production setup.
- Some queries are not too memory-intensive:
  - `SELECT columns FROM table;`
  - `SELECT columns FROM table WHERE condition;`
  - Data is sent back to the coordinator node, instead of being stored in memory.
- Some queries are more expensive:
  - `ORDER BY, GROUP BY, DISTINCT, UNION`
  - These type of queries require intermediate results to be held in memory.
- This does not mean that you should not write these types of queries, just to be careful.

Optimizing memory usage: `GROUP BY`

- Calls to aggregation functions like `MAX(), AVG(),SUM()` are efficient, *provided that* the `GROUP BY` column has few unique values.
- A `GROUP BY` clause involving a column of type `string` is much less efficient than if we normalize (replace long strings with numeric IDs).

Optimizing memory usage: `UNION`

- `UNION` operator does more work than `UNION ALL` because `UNION` eliminates duplicates.
- If possible, replace `UNION` with a `WITH` subquery using `UNION ALL` and then use `DISTINCT` on the result set.

Optimizing memory usage: `INT`

- For tiny values (1-12 months, 1-31 for days for instance), use `TINYINT` instead of `INT` type.
  - `TINYINT`: -128 to 127.
  - `INT:` -2147483648 to 2147483647.
  - `SMALLINT:` -32768 to 32767.
  - `BIGINT:` -9223372036854775808 to 9223372036854775807.
- In this case, might be better to have a single `timestamp` column and use `EXTRACT()` when needed.

Optimizing memory usage during data load

- Inserting data into a Parquet table might be intensive, especially a partitioned table.
- To help this computation be effective, `COMPUTE STATS` for the source table can help correctly estimate the volume and distribution of data being inserted.
- If statistics are not available or are inaccurate, the `SHUFFLE` hint can be included before `SELECT` in a `INSERT ... SELECT` statement.
- Run separate `INSERT` statements for each partition, including constant values.

Partitioned Tables

- Partitioning acts like indices, leveraging fast bulk I/O capabilities of HDFS.
- Columns like `YEAR`, `COUNTRY` make good candidates for partition keys.
- Recommended block size of partition is 5 GB and higher.
    - Having too many blocks in a partition puts burden into Metastore and Hadoop NameNode who need to track all the small pieces of your table.