**How to Make Your Hive Queries Run Faster on Hadoop**

**Technique #1: Use Tez**

Hive can use the [Apache Tez](https://br.hortonworks.com/hadoop/tez) execution engine instead of the venerable Map-reduce engine. I won’t go into details about the many benefits of using Tez which are mentioned here; instead, I want to make a simple recommendation: if it’s not turned on by default in your environment, use Tez by setting to ‘true’ the following in the beginning of your Hive query:

set hive.execution.engine=tez;

With the above setting, every HIVE query you execute will take advantage of Tez.

**Technique #2: Use ORCFile**

Hive supports ORCfile, a new table storage format that sports fantastic speed improvements through techniques like [predicate push-down, compression](https://br.hortonworks.com/blog/orcfile-in-hdp-2-better-compression-better-performance/) and more.

Using ORCFile for every HIVE table should really be a no-brainer and extremely beneficial to get fast response times for your HIVE queries.

As an example, consider two large tables A and B (stored as text files, with some columns not all specified here), and a simple query like:

SELECT A.customerID, A.name, A.age, A.address join

B.role, B.department, B.salary

ON A.customerID=B.customerID;

This query may take a long time to execute since tables A and B are both stored as TEXT. Converting these tables to ORCFile format will usually reduce query time significantly:

CREATE TABLE A\_ORC (

customerID int, name string, age int, address string

) STORED AS ORC tblproperties (“orc.compress" = “SNAPPY”);

INSERT INTO TABLE A\_ORC SELECT \* FROM A;

CREATE TABLE B\_ORC (

customerID int, role string, salary float, department string

) STORED AS ORC tblproperties (“orc.compress" = “SNAPPY”);

INSERT INTO TABLE B\_ORC SELECT \* FROM B;

SELECT A\_ORC.customerID, A\_ORC.name,

A\_ORC.age, A\_ORC.address join

B\_ORC.role, B\_ORC.department, B\_ORC.salary

ON A\_ORC.customerID=B\_ORC.customerID;

ORC supports compressed storage (with ZLIB or as shown above with SNAPPY) but also uncompressed storage.

Converting base tables to ORC is often the responsibility of your ingest team, and it may take them some time to change the complete ingestion process due to other priorities. The benefits of ORCFile are so tangible that I often recommend a do-it-yourself approach as demonstrated above – convert A into A\_ORC and B into B\_ORC and do the join that way, so that you benefit from faster queries immediately, with no dependencies on other teams.

**Technique #3: Use Vectorization**

Vectorized query execution improves performance of operations like scans, aggregations, filters and joins, by performing them in batches of 1024 rows at once instead of single row each time.

Introduced in Hive 0.13, this feature significantly improves query execution time, and is easily enabled with two parameters settings:

set hive.vectorized.execution.enabled = true;

set hive.vectorized.execution.reduce.enabled = true;

**Technique #4: cost based query optimization**

Hive optimizes each query’s logical and physical execution plan before submitting for final execution. These optimizations are not based on the cost of the query – that is, until now.

A recent addition to Hive, [Cost-based optimization](http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.2.0/Cost_based_SQL_Optimization_v22/index.html#Item1.1), performs further optimizations based on query cost, resulting in potentially different decisions: how to order joins, which type of join to perform, degree of parallelism and others.

To use cost-based optimization (also known as CBO), set the following parameters at the beginning of your query:

set hive.cbo.enable=true;

set hive.compute.query.using.stats=true;

set hive.stats.fetch.column.stats=true;

set hive.stats.fetch.partition.stats=true;

Then, prepare the data for CBO by running Hive’s “analyze” command to collect various statistics on the tables for which we want to use CBO.

For example, in a table tweets we want to collect statistics about the table and about 2 columns: “sender” and “topic”:

analyze table tweets compute statistics;

analyze table tweets compute statistics for columns sender, topic;

With HIVE 0.14 (on HDP 2.2) the analyze command works much faster, and you don’t need to specify each column, so you can just issue:

analyze table tweets compute statistics for columns;

That’s it. Now executing a query using this table should result in a different execution plan that is faster because of the cost calculation and different execution plan created by Hive.

**Technique #5: Write good SQL**

SQL is a powerful declarative language. Like other declarative languages, there is more than one way to write a SQL statement. Although each statement’s functionality is the same, it may have strikingly different performance characteristics.

Let’s look at an example. Consider a click-stream event table:

CREATE TABLE clicks (

timestamp date, sessionID string, url string, source\_ip string

) STORED as ORC tblproperties (“orc.compress” = “SNAPPY”);

Each record represents a click event, and we would like to find the latest URL for each sessionID.

One might consider the following approach:

SELECT clicks.\* FROM clicks inner join

(select sessionID, max(timestamp) as max\_ts from clicks

group by sessionID) latest

ON clicks.sessionID = latest.sessionID and

clicks.timestamp = latest.max\_ts;

In the above query, we build a sub-query to collect the timestamp of the latest event in each session, and then use an inner join to filter out the rest.

While the query is a reasonable solution—from a functional point of view—it turns out there’s a better way to re-write this query as follows:

SELECT \* FROM

(SELECT \*, RANK() over (partition by sessionID,

order by timestamp desc) as rank

FROM clicks) ranked\_clicks

WHERE ranked\_clicks.rank=1;

Here we use Hive’s OLAP functionality (OVER and RANK) to achieve the same thing, but without a Join.

Clearly, removing an unnecessary join will almost always result in better performance, and when using big data this is more important than ever. I find many cases where queries are not optimal — so look carefully at every query and consider if a rewrite can make it better and faster.

### Analyze Your Table When You Make Changes To It

Analyzing a table (also known as computing statistics) is a built-in Hive operation that you can execute to collect metadata on your table. This can vastly improve query times on the table because it collects the row count, file count, and file size (bytes) that make up the data in the table and gives that to the query planner before execution. By running this query, you collect that information and store it in the Hive Metastore (metadata store), which will make future queries on this table more optimal. Every time the contents of the table change, stats should be recollected.

**Analyze a Table**

|  |
| --- |
|  |
| ANALYZE TABLE my\_database.my\_table compute statistics; | |

[view raw](https://gist.github.com/hadoopsters/94c9757cdb0868bb08d5554c04c316cc/raw/64bb616f0f3fe6d4cbe769f8184473f28e5f6a1b/analyzing_hive_tables.hql) [analyzing\_hive\_tables.hql](https://gist.github.com/hadoopsters/94c9757cdb0868bb08d5554c04c316cc#file-analyzing_hive_tables-hql) hosted with by [GitHub](https://github.com)

You can (and really should) analyze partitioned tables as well. In fact, it’s particularly helpful for tables with partitions, as, again, it assists in query planning and optimization when querying the table.

**Analyze a Partitioned Table**

|  |
| --- |
|  |
| ANALYZE TABLE my\_database.my\_table PARTITION (YEAR=2017, MONTH=11, DAY=30) compute statistics; | |

[view raw](https://gist.github.com/hadoopsters/67c83e113b000d607314bc7d0280d89b/raw/64e487b82367ac2996eb0092afb4c31fa112a055/analyzing_partitioned_hive_tables.hql) [analyzing\_partitioned\_hive\_tables.hql](https://gist.github.com/hadoopsters/67c83e113b000d607314bc7d0280d89b#file-analyzing_partitioned_hive_tables-hql) hosted with by [GitHub](https://github.com)

You can also analyze the columns of your table and/or partitions. This is a more intense stat-collecting function that collects metadata on columns you specify, and stores that information in the Hive Metastore for query optimization. That information includes, per column, The number of distinct values, The number of NULL values, Minimum or maximum K values where K could be given by a user, Histogram: frequency and height balanced, Average size of the column, Average or sum of all values in the column if their type is numerical, and Percentiles of the value.

**Analyzing Table Columns**

|  |
| --- |
|  |
| ANALYZE TABLE my\_database.my\_table compute statistics for column1, column2, column3; -- column stats for non-partitioned table | |
|  | |

|  |
| --- |
| ANALYZE TABLE my\_database.my\_table PARTITION (YEAR=2017, MONTH=11, DAY=30, HOUR=0) compute statistics for column1, column2, column3; -- column stats for single hour of partitioned table |
|  |

|  |
| --- |
| ANALYZE TABLE my\_database.my\_table PARTITION (YEAR=2017, MONTH=11, DAY=30, HOUR) compute statistics for column1, column2, column3; -- column stats for a single day of partitioned table |

[view raw](https://gist.github.com/hadoopsters/216fb0716f2e1d8f52bca0bda722f354/raw/a1af8f7d4a01c57c3bc680854a2543b501b76a01/analyzing_columns_in_hive_tables.hql) [analyzing\_columns\_in\_hive\_tables.hql](https://gist.github.com/hadoopsters/216fb0716f2e1d8f52bca0bda722f354#file-analyzing_columns_in_hive_tables-hql) hosted with by [GitHub](https://github.com)

Make sure (when you’re ready to query your table) you have these Hive settings enabled to ensure you make the most use out of your calculated stats!

|  |
| --- |
|  |
| set hive.compute.query.using.stats=true; | |
|  | |



|  |
| --- |
| set hive.stats.fetch.column.stats=true; |

[view raw](https://gist.github.com/hadoopsters/24dc975233f8e6b87d435bd7647e3b78/raw/a2d1c30a2e335a1759928e74660460e427445ef1/hive_settings.hql) [hive\_settings.hql](https://gist.github.com/hadoopsters/24dc975233f8e6b87d435bd7647e3b78#file-hive_settings-hql) hosted with by [GitHub](https://github.com)

###  Use ORC, Partitioning and Analyzing for a Powerful Combo

1. Build your table with partitions, ORC format, and SNAPPY compression.
2. Analyze your table when you make changes or add a partition, and analyze the partition.
3. Analyze the columns you use most often (or all of them) at the partition level when you add a partition.