Apache Hive Optimization Techniques — 2

Ankit Prakash Gupta Aug 13, 2019 · 10 min read

In the earlier article, we covered how appropriate data modeling using partitioning and bucketing, choosing Tez as execution engine as well as compression could prove to be very big cost-saving factors. Let us now delve into the other factors which can be very fruitful optimizations.

Using ORC Format

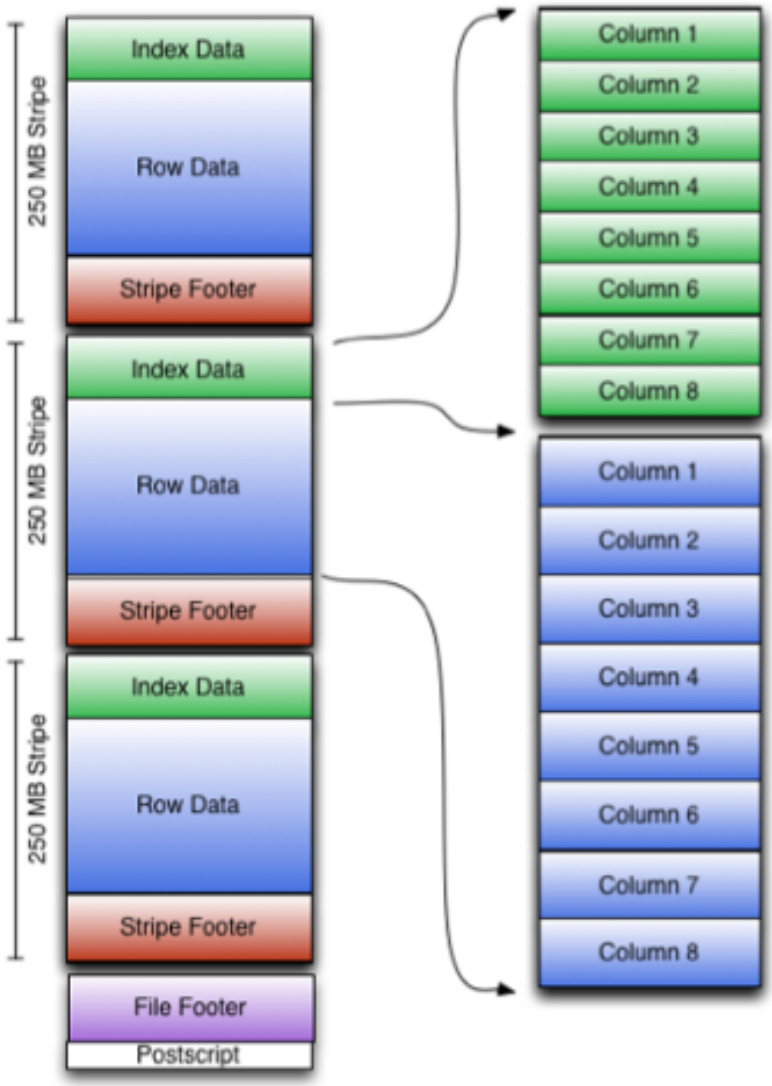
Choosing the correct file format can be another thing which can optimize the processing job to a great extent. Hive supports a big family of file-formats:

- Text File
- SequenceFile
- RCFiles
- Avro Files
- ORC Files
- Parquet
- Custom INPUTFORMAT and OUTPUTFORMAT

Before, any of these columnar-format most of the data was stored in the form of delimited text files. In these files, even if we have to read only one column we have to read and parse the whole record for each row. ORC files enable numerous optimizations in Hive like the following:

- Achieves higher level of compression, the compression algorithm can be changed using **orc.compress** setting in the hive. By default, it uses Zlib Compression.
- It has the ability to skip scanning an entire range of rows within a block, if irrelevant to the query, using the light-weight indexes stored within the file.
- It has the ability to skip decompression of rows within a block, if irrelevant to the query.
- Single file as the output of each task, reducing the load on the name node.
- It supports multiple streams to read the file simultaneously.
- It keeps the metadata stored in the file using Protocol Buffers, used for serializing structured data.

single mapper (256 MB and 512 MB are common sizes.) Stripes may or may not be processed by separate tasks, and a single ORC file can contain many stripes, each of which is independent. In stripes, the columns are compressed separately, only the columns that are in the projection for a query need be decompressed.



Within a stripe the data is divided into 3 Groups:

1. The **stripe footer** contains a directory of stream locations.
2. **Row data** is used in table scans, by default contains 10,000 rows.
3. **Index data** include min and max values for each column and row positions within each column. **Row index** entries provide offsets that enable seeking to the right

queries.

Just using ORC format, gives rise to the following optimizations:

- Push-down Predicates
- Bloom Filters
- ORC Compression
- Vectorization

Push-down Predicates

The ORC reader uses the information present in the Index data of the stripes, to make sure whether the data in the stripe is relevant to the query or not. So, essentially the where conditions of the query are passed to the ORC Reader, before bothering to unpack the relevant columns from the row data block. For instance, if the value from the where condition is not within the min/max range for a block, the entire block can be skipped.

Bloom Filters

Bloom Filters is a probabilistic data structure that tells us whether an element is present in a set or not by using a minimal amount of memory. A catchy thing about bloom filters is that they will occasionally incorrectly answer that an element is present when it is not. Their saving grace, however, is that they will never tell you that an element is not present if it is.

Bloom Filters again helps in the push-down predicates for ORC File formats. If a Bloom filter is specified for a column, even if the min/max values in a row-group's index say that a given column value is within the range for the row group, the Bloom filter can answer specifically whether the value is actually present. If there is a significant probability that values in a where clause will not be present, this can save a lot of pointless data manipulation.

We can set the bloom filter columns and bloom filter's false positive probability using the following table properties:

- **orc.bloom.filter.fpp:** false positive probability for bloom filter.

ORC Compression

ORC format not only compresses the values of the columns but also compresses the stripes as a whole. It also provides choices for the overall compression algorithm. It allows the following compression algorithm to be used :

- **Zlib:** Higher Compression. Uses more CPU and saves space.
- **Snappy :** Compresses less. Uses less CPU and more space.
- **None:** No compression at all.

If the CPU is tight, you might want to use Snappy. Rarely will you want to use none, which is used primarily for diagnostic purposes. Also, the Hive's writer is intelligent if it does not foresee any marginal gain, it will not compress the data.

Vectorization

Query Operations like scanning, filtering, aggregations, and joins can be optimized to a great extent using the vectorized query execution. A standard query execution processes one row at a time and goes through a long code path as well as significant metadata interpretation.

Vectorization optimizes the query execution by processing a block of 1024 rows at a time, inside which each row is saved as a vector. On vectors, simple arithmetic and comparison operations are done very quickly. Execution is speeded up because within a large block of data of the same type, the compiler can generate code to execute identical functions in a tight loop without going through the long code path that would be required for independent function calls. This SIMD-like behavior gives rise to a host of low-level efficiencies: fewer instructions executed, superior caching behavior, improved pipelining, more favorable TLB behavior, etc.

Not every data type and operation can benefit from vectorization, but most simple operations do, and the net is usually a significant speedup.

> ***set hive.vectorized.execution.enabled = true;***
>
> ***set hive.vectorized.execution.reduce.enabled=true***
>
> ***set hive.vectorized.execution.reduce.groupby.enabled=true***

We can check if the query is being executed in a vectorized manner using the Explain Command. In the output of the Explain Command, you will probably see the following in different stages:

Execution mode: vectorized

Join Optimizations

So, before delving into the join optimizations, let's understand how a common join query is actually executed using MapReduce.

- Mappers do the parallel sort of the tables on the join keys, which are then passed on to reducers.
- All of the tuples with the same key is given to the same reducer. A reducer may get tuples for more than one key. Key for tuple will also include table id, thus sorted output from two different tables with the same key can be recognized. Reducers will merge the sorted stream to get join output.

One thing to keep in mind while writing the join query is the table which is mentioned last is streamed through the reducers, rest all are buffered in the memory in the reducers. So, we should always keep in mind that the large table is mentioned in the last which will lessen the memory needed by the reducer, or we can use the STREAMTABLE hint.

```
SELECT /*+ STREAMTABLE(a) */ a.val, b.val, c.val
FROM a JOIN b ON (a.key = b.key1) JOIN c ON (c.key =
```

Another thing to pay attention to when where clause is used with a join, then join part is executed first and then the results are filtered using the where clauses. Like, in the below query:

```
SELECT a.val, b.val FROM a LEFT OUTER JOIN b ON (a.key=b.key) WHERE a.ds='2009-07-07' AND b.ds='2009-07-07'
```

While the same filtering of the records could have been executed along when joining the tables. This can be done by including the filtering conditions along with the join conditions.

```
SELECT a.val, b.val FROM a LEFT OUTER JOIN b ON (a.key=b.key AND b.ds='2009-07-07' AND a.ds='2009-07-07')
```

This join includes a shuffle phase in which outputs of the mappers is sorted and shuffled with the join keys, which takes a big cost to be executed in terms of processing, I/O and data transfer cost. So, to reduce the above-mentioned costs, a lot of join algorithms have been worked on, like:

- Multi-way Join
- Map Join
- Bucket Map Join
- SMB Join
- Skew Join

Multi-way Join

If multiple joins share the same driving side join key then all of those joins can be done in a single task.

join key x.
Hence, Reducing the number of reducers by doing all reducing task in just one reducer. Hive does this optimization itself and manages it internally. Like in the below query:

```
SELECT a.val, b.val, c.val FROM a JOIN b ON (a.key = b.key1) JOIN c ON (c.key = b.key1)
```

Map Join (Broadcast Join or Broadcast-Hash Join)

Useful for star schema joins, this joining algorithm keeps all of the small tables (dimension tables) in memory in all of the mappers and big table (fact table) is streamed over it in the mapper. This avoids shuffling cost that is inherent in Common-Join. For each of the small table (dimension table), a hash table would be created using join key as the hash table key and when merging the data in the Mapper Function, data will be matched with the mapping hash value.

But the constraint is, all but one of the tables being joined are small, the join can be performed as a map only job.

Hive can optimize join into the Map-Side join, if we allow it to optimize the joins by doing the following setting:

```
set hive.auto.convert.join=true;

set hive.auto.convert.join.noconditionaltask = true;

set hive.auto.convert.join.noconditionaltask.size = 10000000;
```

The first two settings will allow hive to optimize the joins and third setting will give hive an idea about the memory available in the mapper function to keep the hash table of the small tables.

Or else, we can also use MAPJOIN hint in the query, such as:

```
SELECT /*+ MAPJOIN(b) */ a.key, a.value
```

But the restriction is, full outer join AND right outer join cannot be performed for the table b.

Bucket Map Join

Let's assume that the size of the tables bigger to fit in the memory of the Mapper. But when chunked into buckets can fit in the memory, the tables being joined are bucketized on the join columns, and the number of buckets in one table is a multiple of the number of buckets in the other table, the buckets can be joined with each other by creating a hash table of the smaller table's bucket in the memory and streaming the other table's content in the mapper. Assuming, If table A and B have 4 buckets each, the following join can be done on the mapper only.

```
SELECT /*+ MAPJOIN(b) */ a.key, a.value
FROM a JOIN b ON a.key = b.key
```

Instead of fetching B completely for each mapper of A, only the required buckets are fetched. For the query above, the mapper processing bucket 1 for A will only fetch bucket 1 of B. It is not the default behavior, and is governed by the following parameter.

```
set hive.optimize.bucketmapjoin = true
```

Sort-Merge-Bucket Join

This is an optimization on Bucket Map Join; if data to be joined is already sorted on joining keys then hash table creation is avoided and instead a sort-merge join algorithm is used. This join can be used using the following settings:

```
set hive.input.format= org.apache.hadoop.hive.ql.io.BucketizedHiveInputFormat;
set hive.optimize.bucketmapjoin = true;
set hive.optimize.bucketmapjoin.sortedmerge = true;
```

The query would be the same as the above query, and the hive would form its execution strategy.

FROM a JOIN b ON a.key = b.key

Skew Join

If the distribution of data is skewed for some specific values, then join performance may suffer since some of the instances of join operators (reducers in the map-reduce world) may get overloaded and others may get underutilized. On user hint, hive would rewrite a join query around skew value as union of joins. So, it would execute one query to implement join of the skewed key and other for the rest of the join keys, and would further merge the outputs of both the queries.

Example R1 PR1.x=R2.a — R2 with most data distributed around x=1 then this join may be rewritten as (R1 PR1.x=R2.a and PR1.x=1 — R2) union all (R1 PR1.x=R2.a and PR1.x<>1 — R2)

```
set hive.optimize.skewjoin = true;
set hive.skewjoin.key=500000;
```

Cost-based Optimizations

So, we all will agree to the saying that there could be multiple solutions to one problem and until the rise of the cost-based optimizer, hive used the hard-coded query plans to execute a single query. The CBO lets hive optimize the query plan based on the metadata gathered. CBO provides two types of optimizations: logical and physical.

Logical Optimizations:

- Projection Pruning
- Deducing Transitive Predicates
- Predicate Pushdown
- Merging of Select-Select, Filter-Filter into a single operator
- Multi-way Join

Physical Optimizations:

- Partition Pruning
- Scan pruning based on partitions and bucketing
- Scan pruning if a query is based on sampling
- Apply Group By on the map side in some cases
- Optimize Union so that union can be performed on map side only
- Decide which table to stream last, based on user hint, in a multiway join
- Remove unnecessary reduce sink operators
- For queries with limit clause, reduce the number of files that needs to be scanned for the table.

Turning on CBO will typically halve the execution time for a query, and may do much better than that. Along with these optimizations, CBO will also transform join queries into the best possible join using the statistics of the table saved in the metastore.

CBO can be turned on using the following setting in the hive-site.xml file :

> ***hive.cbo.enable=true***

The statistics of the table is auto gathered by hive, if *hive.stats.autogather* is set to True else could be calculated using the ANALYZE command.

I hope this and the earlier article might have given you a complete overview of how Hive Queries can be optimized and how hive's key features work in the background using MapReduce.

I hope you find this article informative and easy to learn if you have any queries feel free to reach me at info.ankitp@gmail.com and connect with me on LinkedIn.

Big Data Apache Hive Optimization Analytics Best Practices