Technology Snippets by Jörn Franke

Demonstrate the Potential of Existing Technologies

Hive Optimizations with Indexes, Bloom-Filters and Statistics

This blog post describes how Storage Indexes, Bitmap Indexes, Compact Indexes, Aggregate Indexes, Covering Indexes/Materialized Views, Bloom-Filters and statistics can increase performance with Apache Hive to enable a real-time datawarehouse. Furthermore, I will address how index-paradigms change due to big data volumes. Generally it is recommended to use less traditional indexes, but focus on storage indexes and bloom filters. Especially bloom filters are a recommendation for any Big Data warehouse. Additionally, I will explain how you can verify that your query benefits from the indexes described here. I expect that you have some experiences with standard warehousing technologies and know what Hadoop and Hive are. Most of the optimizations are only available in newer versions of Hive (at least 1.2.0) and it is strongly recommended to upgrade to this version, because it enables interactive queries and performance experiences similar to specialized data warehouse technologies.

The Use Case

We assume a simple use case here which is simply one customer table with the following fields:

Name	**Type**
customerId	int
gender	tinyint
age	tinyint
revenue	decimal(10,2)
name	varchar(100)
customerCategory	int

What Changes with Big Data with respect to Indexes and Statistics

Firstly, you need to know that even if you have a Big Data technology, such as Hadoop/Hive, you still need to process your data intelligently to execute efficient queries. While Hadoop/Hive certainly scale to nearly any amount of data, you need to do optimizations in order to process it quickly. Hence, similar approaches, but also new ones exists compared to traditional warehousing technologies. The main difference to a warehouse is that block sizes are usually significantly larger than in standard warehousing. For example, the current default block size in Hive when using the ORC-format for database blocks, which is highly recommended for Big Data Enterprise Warehousing formats, is 256 MB compared to around 4 MB in traditional databases. This means you should not even bother to optimize for tables which do fit in one block, because most of the optimization technologies aim at avoiding to read all data blocks in a table. If you just have one block for a table then you need to read it anyway if you access the table and optimization techniques would make no sense.

Storage-Index

The storage index is only available if your table is in ORC (https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC)-Format in Hive. Similarly the Parquet (https://cwiki.apache.org/confluence/display/Hive/Parquet)-format has also a storage index (min/max index), but many applications including Hive do not leverage it currently for Parquet. The storage index let you skip blocks where a value is not contained without needing to read the block itself. It is most suitable for numeric values (double, float, int, decimal etc.). It kicks in if you have a query using where together with the <,>,= operator or joins. For example:

SELECT * **FROM** table **WHERE** customerId=1000

or

SELECT * **FROM** table **WHERE** age>20 **AND** age < 40.

Keep in mind that for an effective storage index you need to insert the data into the table sorted on the columns for which you want to leverage the storage index. It is much less effective on unsorted tables, because it contains the min-max values of the column. If you have a badly sorted table the min-max value for all blocks will be overlapping or even the same and then the storage index is of no use. This is how you can create a table with storage index in Hive (setting tblproperties 'orc.create.index'='true'):

CREATE TABLE CUSTOMER (

customerId int,

gender tinyint,

age tinyint,

revenue decimal(10,2),

name varchar(100),

customerCategory int)

STORED AS ORC

TBLPROPERTIES ('orc.compress'='SNAPPY',

'orc.create.index'='true',

'orc.bloom.filter.columns'='',

'orc.bloom.filter.fpp'='0.05',

'orc.stripe.size'='268435456',

'orc.row.index.stride'='10000');

You may want to insert the data sorted on the columns for which you want to use the storage index. It is recommended to do this only for 1-2 columns, depending on the data you may want to include more. It is recommended to set the setting 'hive.enforce.sorting' to 'true' and describe in the create table statement which columns should be sorted (e.g. using clustered by (userid) sorted by (age)). You can also insert them sorted by using order by or sort by in the insert/select statement. Alternatively you may want to insert the data clustered by certain columns that are correlated. This clustering has to be done before inserting data into Hive.

Keep in mind that depending on your data you may also tune the index by changing orc.row.index.stride.

Bloom-Filter

Bloom filters (https://en.wikipedia.org/wiki/Bloom_filter)are relatively new feature in Hive (1.2.0) and should be leveraged for any high-performance applications. Bloom filter are suitable for queries using where together with the = operator:

SELECT AVG(revenue) **WHERE** gender=0

A bloom filter can apply to numeric, but also non-numeric (categorical) data, which is an advantage over the storage index. Internally, a bloom filter is a hash value for the data in a column in a given block. This means you can „ask" a bloom filter if it contains a certain value, such as gender=male, without you needing to read the block at all. This obviously increases performance significantly, because most of the time a database is busy with reading non-relevant blocks for a query.

You can tune a bloom filter by configuring the false positive rate ('orc.bloom.filter.fpp'), but you only should deviate from the default if you have understood how bloom filters work in general.

You can specify a bloom filter when using the create or alter statement of the table by setting the TBL property 'orc.bloom.filter.columns' to the columns for which you want to create the bloom filter. It is only available if you use the ORC format:

CREATE TABLE CUSTOMER (

customerId int,

gender tinyint,

age tinyint,

revenue decimal(10,2),

name varchar(100),

customerCategory int)

STORED AS ORC

TBLPROPERTIES

('orc.compress'='SNAPPY',

'orc.create.index'='true', 'orc.bloom.filter.columns'='gender',

'orc.bloom.filter.fpp'='0.05',

'orc.stripe.size'='268435456',

'orc.row.index.stride'='10000');

By default you should use a bloom filter instead of a Bitmap Index or a Compact Index if you have select queries using where together with the = operator or when (equi-)joining large tables. You should increase effectiveness of the bloom filter by inserting data only sorted on the columns for which you define the bloom filter to avoid that all blocks of a table contain all distinct values of the column. It is recommended to set the setting 'hive.enforce.sorting' to 'true' and describe in the create table statement which columns should be sorted (e.g. using clustered by (userid) sorted by (gender)). *You can also insert them sorted by using order by or sort by in the insert/select statement.* Alternatively you may want to insert the data clustered by certain columns that are correlated. This clustering has to be done before inserting data into Hive.

Statistics

Another important aspect is to generate statistics on tables and columns by using the following commands:

ANALYZE TABLE CUSTOMER **COMPUTE STATISTICS**

ANALYZE TABLE CUSTOMER **COMPUTE STATISTICS FOR COLUMNS**

If you have a lot of columns you may only calculate statistics for selected columns. Additionally, Hive cannot currently generate statistics for all column types, e.g. timestamp. Basically, statistics enables Hive to optimize your queries using the cost-based optimizer, which is available in newer versions of Hive. This means usually significant performance improvements. However, you need to recompute statistics every time you change the content of a table. You can also decide only to recompute statistics for selected partitions, which make sense if you only change a partition.

The corresponding commands for tables including partitions (https://cwiki.apache.org/confluence/display/Hive/StatsDev) are the following:

ANALYZE TABLE CUSTOMER **PARTITION**(ds, hr) **COMPUTE STATISTICS**

ANALYZE TABLE CUSTOMER **PARTITION**(ds, hr) **COMPUTE STATISTICS FOR COLUMNS**

Never Hive versions with Hbase as a meta data store allow caching of statistics:

ANALYZE TABLE CUSTOMER **CACHE METADATA**

Aggregate Index

The Aggregate Index has no counterpart in the traditional database world. Even within Hive it is one of the least documented and probably one of the least used ones. Most of the documentation can be derived from analyzing the source code (https://github.com/apache/hive/blob/master/ql/src/java/org/apache/hadoop/hive/ql/index/AggregateIndexHandler.java).

An Aggregate Index is basically a table with predefined aggregations (e.g. count,sum,average etc.) of a certain column, e.g. gender, which are grouped by the same column. For example, the following query leverages an Aggregate Index:

SELECT gender, count(gender) **FROM** CUSTOMER **GROUP BY** gender;

The following query does NOT leverage an Aggregate Index:

SELECT customerCategory , count(gender) **FROM** CUSTOMER **GROUP BY** customerCategory;

Hence, the Aggregate Index is quite limited and only suitable in niche scenarios. However, you can create one as follows in Hive:

CREATE INDEX idx_GENDER

ON TABLE CUSTOMER(GENDER) **AS**
'org.apache.hadoop.hive.ql.index.AggregateIndexHandler'

WITH DEFERRED REBUILD

IDXPROPERTIES('AGGREGATES'='count(gender)')

STORED AS ORC TBLPROPERTIES

('**orc.compress'='SNAPPY',**

'orc.create.index'='true');

After the index is created you need to update them once you changed the data in the source table as follows:

ALTER INDEX idx_GENDER **ON** CUSTOMER **REBUILD**;

You can choose to auto-update the index, however in a data warehouse environment it makes usually more sense if you do it manually. Keep in mind that you always should as little indexes as possible, because having too many indexes on a table affects seriously performance.

The Aggregate Index is certainly an interesting one, but the community need to put some more thinking into it to make it more effective, e.g. learning from past aggregate queries to have all the required aggregates stored appropriately. This should be not a dump caching index, but make it smart. For instance, if users often calculate the aggregates for customer numbers on a certain continent then the individual countries should be added automatically to the index. Finally, the Aggregate Index could be augmented with a counting bloom filter (https://en.wikipedia.org/wiki/Bloom_filter#Counting_filters), but this one is not implemented in Hive.

Covering Index/Materialized Views

Hive has currently no command to create and maintain automatically a covering index or materialized view. Nevertheless, these are simply subsets of the table they are referring to. Hence, you can imitate the behavior by using the following commands:

CREATE TABLE CUSTOMER2040

STORED AS ORC

TBLPROPERTIES (

'orc.compress'='SNAPPY',

'orc.create.index'='true', 'orc.bloom.filter.columns'='customerid',

'orc.bloom.filter.fpp'='0.05',

'orc.stripe.size'='268435456',

'orc.row.index.stride'='10000')

AS SELECT * **FROM** CUSTOMER **WHERE** (age>=20) **AND** (age<=40)

This creates a “materialized view” on the customers aged between 20-40. Of course you have to update them manually, which can be a little bit tricky. You may have a flag in the original table to indicate if a customer has been upgraded from the last time you created a „materialized view“ from it. You may partition the alternative table differently to gain performance improvements.

Bitmap Index

A Bitmap Index in hive can be compared to the Bitmap Index in traditional databases (https://en.wikipedia.org/wiki/Bitmap_index). However, one should evaluate if it can be replaced by a Bloom-Filter, which performs significantly better. The Bitmap-index is suitable for queries using where together with the = operator.

It is suitable if the column on which you put the Bitmap-index has only few distinct values, e.g. it is a gender column or a customer category.

Similar to the storage index, it is most effective if you sort the data in this column, because otherwise every block contains values in the bitmap index and hence every block is read anyway. It is recommended to set the setting ‘hive.enforce.sorting’ to ‘true’ and describe in the create table statement which columns should be sorted. Alternatively, you can create reasonably smaller block sizes (by setting orc.stripe.size) for the table to which the index applies to reduce the likelihood that one block contains all types of values.

Internally, a bitmap index in Hive is simply another table, which contains the unique values of the column together with the block where they are stored. Of course, you can also define column-combinations. For example, you can put an index covering gender and customer category. This is most effective if the where part of your query discriminates often against both at the same time.

Since the bitmap index is „just“ another table you need to also select the right format as for normal tables. It is recommended to use the same as for normal tables, such as ORC or Parquet. You may also want to decide to compress the index, but only if it is really large. If your table is already partitioned the index is also partitioned. You may change this partition structure for the index to increase performance.

You can create a bitmap index as follows in Hive (ORC format, compressed with Snappy):

CREATE INDEX idx_GENDER

ON TABLE CUSTOMER (GENDER) **AS**
‘org.apache.hadoop.hive.ql.index.bitmap.BitmapIndexHandler’

WITH DEFERRED REBUILD

STORED AS ORC

TBLPROPERTIES (

'orc.compress'='SNAPPY',

'orc.create.index'='true');

After the index is created you need to update them once you changed the data in the source table as follows:

ALTER INDEX idx_GENDER **ON** CUSTOMER **REBUILD**;

You can choose to auto-update the index, however in a data warehouse environment it makes usually more sense if you do it manually.

Compact Index

The Compact Index can be compared to a normal index in a traditional database. Similarly to a Bitmap Index one should evaluate if it makes sense to use a bloom filter and storage indexes instead. In most of the cases the Compact Index is not needed anymore.

A Compact Index is suitable for columns containing a lot of distinct values, e.g. customer identifier. It is recommended to use a Compact Index for numeric values. It is also suitable for queries using where with the <,>,= operators or joins. Internally, a Compact Index is represented in Hive as a table sorted table containing all the values of the column to be indexed and the blocks where they occur. Of course you can define an index on several columns, if you use them very often together in the where part of a query or in the join-part.

Since the index is a table, you should store it in ORC format and optionally compress it. If your table is already partitioned the index is also partitioned. You may change this partition structure for the index to increase performance. You can define that the index can be used for compact binary search by setting the IDXPROPERTIES 'hive.index.compact.binary.search'='true'. This optimizes search within the index. You do not need to sort the data, because this index is sorted.

You can create a Compact Index as follows in Hive (ORC format, compressed with Snappy, binary search enabled):

CREATE INDEX idx_CUSTOMERID

ON TABLE CUSTOMER (CUSTOMERID) **AS**
'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler'

WITH DEFERRED REBUILD

IDXPROPERTIES ('hive.index.compact.binary.search'='true')

STORED AS ORC

TBLPROPERTIES (

'orc.compress'='SNAPPY',

'orc.create.index'='true');

After the index is created you need to update them once you changed the data in the source table as follows:

ALTER INDEX idx_CUSTOMERID **ON** CUSTOMER **REBUILD**;

You can choose to auto-update the index, however in a data warehouse environment it makes usually more sense if you do it manually.

Verifying Index-and Bloom-Filter Usage in Queries

You can verify that your bloom filters and storage indexes work by setting the following property before executing your query in beeline (assuming you use TEZ as an execution engine):

set hive.tez.exec.print.summary=true;

Afterwards you execute the query in beeline and get at the end some summary statistics about how many bytes have been read, written and how many rows have been processed. This will help you to identify further optimizations of your storage index or bloom filters, such as inserting data sorted on a selected relevant subset of columns. Alternatively, you can use TEZ-UI to see a summary of these statistics.

Similarly you can verify the usage of traditional indexes, such as the Bitmap-Index or the Compact-Index. Additionally, you can get information using EXPLAIN or EXPLAIN DEPENDENCY (https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Explain) on your query.

Conclusions

We presented here different approaches to optimize your Big Data data warehouse Hive. Especially the new version of Hive (1.2.0) offers performance comparable to traditional data warehouses or appliances, but suitable for Big Data. It also came clear that traditional warehouse approaches for optimizations are supported, but are deprecated in the Big Data analytics world:

- Bitmap Indexes can be replaced by a bloom filter
- Compact Indexes can be replaced by a bloom filter and storage indexes

In some rare cases Compact Indexes can still make sense in addition to bloom filters and storage indexes. Examples for these cases are OLTP scenarios, where one looks up one specific row by a primary key or similar. Alternatively, you may duplicate tables with different sort order to improve performance. Niche methods, such as Aggregate Indexes, are promising, but need to mature more. However, if you have a bad data model consisting only of Strings or varchars then you will face decreased performance anyway. Always use the right data type for your data, particularly the numeric data. You may also want to replace dates with ints. I have not addressed here several other optimization methods that are available in Hive:

- Think about pre-joining tables into a flat table. Storage indexes and bloom filters work efficiently for flat tables instead of joining every time when needed. This is a paradigm shift from traditional databases.
- Understand how the Yarn Scheduler (Fair (https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html) or Capacity (https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html)) works for optimal concurrency and high performance queries
- hive.execution.engine: You should set it to TEZ for most of your scenarios. Alternatively you may set Spark, but this will mean some optimizations do not work.
- Prewarm TEZ containers.
- Use TEZ > 0.8 for interactive subsecond queries (TEZ Service) together with LLAP (http://de.slideshare.net/Hadoop_Summit/llap-longlived-execution-in-hive)
- Do not use single insert, updates, deletes for analytics tables, but bulk operations, such as CREATE TABLE AS SELECT … (CTAS).
- Use in-database analytics and machine learning algorithms provided by HiveMall (https://github.com/myui/hivemall)
- Use in-memory techniques for often used tables/partitions, e.g. with Apache Ignite (https://ignite.apache.org/)
- Store lookup tables in Hbase: You can create external Tables in Hive pointing to tables in Hbase (https://cwiki.apache.org/confluence/display/Hive/HBaseIntegration). These show a high performance for lookup operations of reference data, e.g. in case you want to validate incoming data on a row by row base in staging tables avoiding costly joins.
- Use the Mapside join hint (https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Joins)to load (small) dimension tables in-memory
- Increase replication factor of selected tables,partitions,buckets,indexes: The more copies you have on different nodes the more performance you gain with the disadvantage of wasting more space. You can use the hadoop dfs -setrep -R -w X /path/to/hive/tableinhdfs (X is the replication count, 3 is the default)
- Use partitions: (https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL) Partitions increase significantly your performance, because only a selected subset of the data needs to be read. Be aware that partitions are only enabled when you use it in the where clause and in case of joins ***additionally*** in the on-clause (otherwise you will do anyway a full table scan for older Hive versions). Example for a transaction table partitioned by transaction date:

    - **SELECT** * **FROM** CUSTOMER T1 **LEFT JOIN** TRANSACTIONS T2 **ON** (T1.CUSTOMERID=T2.CUSTOMERID AND T2.DATE=20140101) **WHERE** T2.DATE=20140101
- Use compression: We used here in the example Snappy compression, because it is fast. However, it does not compress as much as zlib. Hence, for older data in different partitions you may use zlib instead of Snappy, because they are accessed less often.
- Use Bucketing (https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#LanguageManualDDL-BucketedSortedTables): Bucketing is suitable for optimizing map-side joins or if you want to sample data . For instance, in an extremely large table you may only select a sample of data and compute the average for this sample, because the result will be similar to the one obtained if you calculate the average over the full table:
    - **SELECT** AVG (AGE) **FROM** CUSTOMER **TABLESAMPLE**(**BUCKET** 3 **OUT OF** 100)

Posted on July 25, 2015May 28, 2016 by jornfranke This entry was posted in big data, data warehouse, hive and tagged aggregateindex, analytics, big data, bitmapindex, bloom filter, compactindex, data warehouse, hadoop, high performance, hive, index, llap, realtime, statistics, storage, tez. Bookmark the permalink.

13 thoughts on “Hive Optimizations with Indexes, Bloom-Filters and Statistics”

PATCHAREE SAYS:
OCTOBER 13, 2015 AT 7:45 AM
Hi. Thanks for very interesting post. However what hive version did you use? I tried to follow your test, but the create table statement (with sorted by) failed (ParseException) on hive 1.2.1

REPLY

JORNFRANKE SAYS:
OCTOBER 24, 2015 AT 5:39 PM
Hive 1.2.0, can you please post the statement and the error. Thank you.

REPLY

JORNFRANKE SAYS:
OCTOBER 24, 2015 AT 7:20 PM
sorry there was a small error in the statements, you need to put sorted by before stored as…
you can find the fix in the updated text

REPLY

Pingback: Batch-processing & Interactive Analytics for Big Data – the Role of in-Memory | Blog by Jörn Franke
Pingback: Big Data News: HUG Ireland’s 1st 2016 Big Data Event, Airbnb’s Predictive Model using NPS and Hive Optimization | Sonra. Unleash the Value of your Data.
ARMAAN SAYS:
MARCH 15, 2016 AT 7:36 AM
This syntax is not working without using clustered by .Also how can we see this index is utilised or not.

REPLY

JORNFRANKE SAYS:
MARCH 15, 2016 AT 5:29 PM
Thank you for your comment. The syntax has been corrected. It was already described in the section “Verifying Index-and Bloom-Filter Usage in Queries” how to check if indexes are used.

REPLY

Pingback: Hive & Bitcoin: Analytics on Blockchain data with SQL | Technology Snippets by Jörn Franke
DENNIS SAYS:
MAY 27, 2016 AT 7:43 PM
“Similarly the Parquet-format has also a storage index (min/max index), but many applications including Hive do not leverage it currently for Parquet.”, that was in version 1.2.0? Do you know if that has changed? We use Spark with Parquet and

query external Hive tables based on that. I can clearly see that no records are being read from files that don't match the statistics metadata but I'm finding it hard to figure out why that is.

REPLY

JORNFRANKE SAYS:
MAY 27, 2016 AT 8:17 PM
I am not sure if I understood your example here. Do you think it is not leveraged in your case or it is? Do you use HiveContext or access the files directly from Spark? There has been issues with Hive and Parquet, also in 1.2.0, but still in 1.2.0 the predicate pushdown for Parquet should work (maybe it could be more optimized). You could check if it works in Hive, if you have TEZ. You may want to activate the option hive.optimize.index.filter and hive.optimize.ppd in Hive. If you just use Spark without going in fact through Hive, you have to activate the option "spark.sql.parquet.filterPushdown". Do not forget to sort the data on the column which you use in the where clause – otherwise you will have to read the whole table independent of min/max indexes. You can get metrics in Spark using its monitoring framework (http://spark.apache.org/docs/latest/monitoring.html#metrics) or the log files. Similarly you can check the Hive log files.

REPLY

DENNISHUNZIKER SAYS:
JUNE 1, 2016 AT 9:00 PM
We use HiveContext and I can see that the filtering based on meta data (min/max) is working fine. I saw that you mentioned it wasn't supported so I was expecting it not to work, hence the question whether you're aware if anything has changed since writing this post.
We're in the process of upgrading Spark, so I'll definitely have a look at some of these options and sorting the data.

JORNFRANKE SAYS:
JUNE 2, 2016 AT 6:08 PM
It works, but there was in one release a bug where it did not work for parquet and scanned the whole table. Sorting is essential for leveraging the min/max
indexes.

Pingback: #onenote# hive – Think Note

__PRESENT