White Paper

Accelerating BI on Hadoop: Full-Scan, Cubes or Indexes?

jethro

+1(844)384-3844 | INFO@JETHRO.IO

Overview

Organizations are storing more and more of their data in Hadoop and cloud based data lakes. Naturally, BI users start pointing their BI applications at this growing data source. Traditionally, many BI apps work by extracting data from a data source and then loading it into the BI tool's memory for fast processing. However, as dataset sizes grow into the BB's of rows, this approach is no longer practical as the data is simply too big to be processed in memory. In that case, most BI tools offer an alternative – instead of bringing the data into the BI tool, push the queries live to the data source. This is known as Live Connect (in **Tableau**), Direct Discovery (**Qlik**), DirectQuery (**PowerBI**), Direct Access (**MicroStrategy**) and so on. When switching to live DB access, size limitations no longer apply. However, BI interaction speed now depends on the performance of the data source engine.

The Problem

There are several SQL engines that can be used as the data source on Hadoop. These include Hive, Impala, Presto, SparkSQL, Drill and others. All of these tools share the same basic architecture, known as MPP full-scan. While they are perfectly suitable for ETL and data science workloads such as predictive modeling and machine learning, these tools are greatly challenged when used for BI apps. The reason for that is that BI workload has unique characteristics:

- **Concurrency**: A typical BI dashboard will issue numerous queries to render results. Multiply that by hundreds of users and the system is flooded with queries.
- **Performance**: BI users expect their dashboards to respond in no more than 5-10 seconds. Moreover, it is not acceptable that some of the charts are rendered fast – the entire dashboard needs to be completed.
- **Selectivity & Variance**: A user is typically interested in a relatively small subset of the data at any given interaction and would use several filters to identify it. Each user, however, is likely to be interested in a different subset.
- **Ad-Hoc & Agile**: With self-service BI dashboards and apps are continuously changing with new queries being created frequently.
- **Complex Data Manipulation**: Multidimensional Analysis requires joining of tables, sorting of data, large aggregations, and other expensive operations.

SQL-on-Hadoop engines are not suitable for the type and volume of BI queries as their full-scan architecture requires tremendous amount of redundant scan work. These engines will need to read the entire column (i.e. all rows) of each filter col, of each query, every single time. A few optimizations, such as partitions/projections/etc.', are also not suitable for a world of self-service BI, in which at any point of time, queries can use a different column for filtering or sorting, even if this column is not the one used as the column for partitioning. The result is not only slow, it also adds to the overall load of the system and results in a cap for the number of concurrent queries that can be served.

Workaround Solutions

In order to speed up live access of BI apps to large datasets in Hadoop, companies engage in a range of manual and complex data engineering projects to bridge the gap. A few common ones include:

- **De-normalization**: as joining data from multiple tables tends to be slow, data can be merged into one large table and the joins can be avoided. The trade of is that ow we have an even larger dataset to maintain. We are no longer able to incrementally add data – instead, we have to rebuild the full dataset frequently. This model sacrifices the application coherence in flexibility to overcome a data source's inability to perform.

- **Pre-aggregation**: create a large number of aggregations to meet every query being used by the app. This results in ongoing maintenance, trying to keep up with app changes. In addition, granular queries will still need to full-scan the table and suffer from slow performance.
- **Multi-partitioning of the data**: creating multiple replicas of the data with different partition keys in order to provide faster queries to multiple filter columns.

All of these prove to be labor intensive, expensive to maintain, and ultimately ineffective.

OLAP Cubes on Hadoop

One emerging solution is to use a commercial (AtScale, Kyvos, Dremio, kyligence) or open source (Kylin) tools to build OLAP cubes on top of Hadoop datasets. These can then be used to speed up queries that can match one of the cubes. As we know from many years of working with cubes, this approach has several limitations:

- **Manual:** cubes and aggregations need to be manually defined to match application needs. This process requires experts with deep knowledge of the data model and application queries. As the application and data changes, a team of such experts are required to constantly modify the cubes to keep up with ongoing changes.
- **Incomplete**: cubes work well for highly aggregated queries. However, for highly granular queries cubes are simply impractical. For example, let's think of a dataset that includes a high cardinality column such as "customer_id" with MM's of unique values. Some queries might need to look at an individual customer_id. Building a cube that contains this column will result in a cube so big it will not be practical. Applications where users tend to filter by multiple dimensions similarly require large and therefore impractical cubes.
- **Costly**: As we're dealing with big data, cubes can become very large, and overall, even bigger than the original data itself. Such large cubes are hard to maintain and are especially challenging when data is update daily, hourly, or even every few minutes. Cubes refresh and rebuild is a labor-intensive process.

When queries are unable to use a cube, they are forced to run with one of the native SQL-on-Hadoop engine and go through a full-scan process. Unfortunately, a partial solution that make some queries perform fast but others don't is often unacceptable for BI users.

Indexes on Hadoop

Another emerging solution is to fully index datasets in Hadoop using commercial tools (Jethro) or open source (Druid). In that case, a DB-like index is built for each column. When queries are sent from the BI app, the indexes are used to narrow down the rows needed for the query, instead of performing a slow full-scan. With full-indexing, we

can be prepared for any existing or future queries as regardless of which columns the user will choose to filter by there will be an index to accelerate it.

And like with any other technology, indexes have their own trade-offs:

- **Costly:** Indexes require additional storage and index maintenance can slow down new data ingestion.
- **Incomplete:** Not all queries can use an index. Some queries don't use a filter at all or use a filter with such a low cardinality that most of the rows still need to be accessed.

When a query cannot use an index, it is forced to do a full-scan which is typically slow and resource consuming.

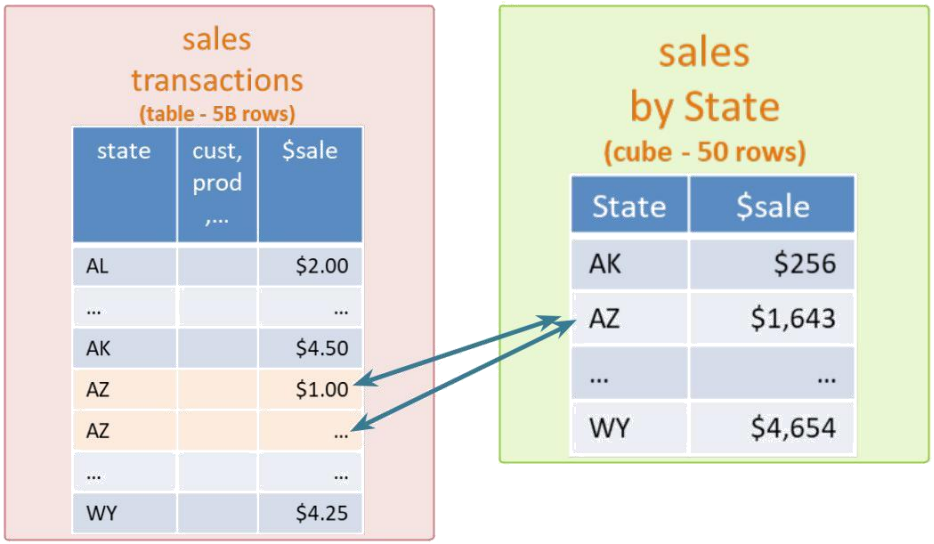
How about Cubes and Indexes?

As it turns out, cubes and indexes and perfectly complimentary approaches with very little overlap. The scenarios where cubes will be ineffective are the exact scenarios where indexes would shine, and vice versa. Cubes work great when queries are highly aggregated, and the cubes are small in size. This is usually the case when few, low cardinality dimensions are used. For example, a cube including dimensions such as gender, age group, state, or market category will be small and highly effective. At the same time, using indexes to filter on such dimension columns will not be highly effective as a significant portion of the data will still need to be scanned. Conversely, Indexes work great when queries are highly granular and looking for a small subset of the data. For example, a query that filters by item_id (MM's of rows of values) and by customer zip code (1,000's of values) or a query that drills down to a customer or transaction level data. For such granular queries cubes will be a poor solution as they will need be very large because of the many hi cardinality dimensions. Such large cubes will be impractical and slow.

Using both cubes and indexes is the only way to provide fast performance for the full-range of BI queries – aggregated and granular – and avoid slow full-scans altogether.

Cubes in Jethro

Jethro AutoCubes are exactly as they sound – cubes that are created automatically by Jethro instead of manually by an expert. Jethro monitors ongoing queries sent by BI tools and examines each of them to see if it could be optimally served by a cube. If so, Jethro will create such a cube using a background process and store it in HDFS. Future permutations of this query – different filter values, any subset combination of the dimensions / measure – will be served from the cube instead of the full table. Jethro cubes are completely transparent to the BI app which is only aware of the underlying tables.



Key features of Jethro AutoCubes

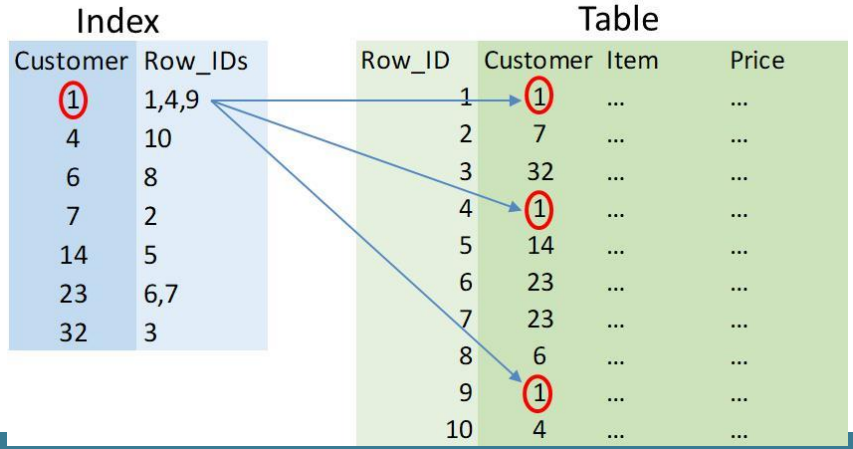
- - Cubes over table JOIN. Jethro builds two types of cubes for queries that include joined tables. The first is a “dimension” cube which is built based on the results of the query, after the JOIN is performed and can be used with both star and snowflake schemas. The second is “key” cube which is built around the JOIN keys of the underlying tables. Such key cube can be used for a large number of future queries as it is generic in nature.
- Support for COUNT DISTINCT aggregations. When the measure used for a query is COUNTD Jethro will store the list of distinct values for each cube entry instead of the overall number of such unique values. This enables Jethro to use the cube to answer not only how many distinct values are in a specific entry (e.g. STATE=’AZ”) but also queries accessing multiple entries (e.g. state IN (‘CA’, ‘NY’)).

Cube maintenance is also automated. When new data arrives, Jethro will incrementally update all the cubes it built. As apps are modified or new one added, or as new columns are added to the data model, Jethro automatically adapts and creates new cubes to accommodate them.

Because Jethro cubes are designed to work in conjunction with Jethro indexes, Jethro will only create small cubes that are optimal for aggregated queries. Large cubes that would be needed for granular queries are avoided as such queries are optimized using indexes instead.

Indexes in Jethro

Jethro indexes all columns automatically so no expert is required to decide which cols are most important. This approach is critical for self-service BI and agile app development as users can freely choose how to access their data and never be concerned if a specific column is indexed or not and what will be the performance implications of it.



Jethro indexes are typical DB indexes. They are technically Inverted Lists – each col value has an entry with the list of rows that have that specific value. When a query arrives, Jethro first evaluates all filters and uses the index to create a working-set – a list of all rows that are needed after applying all filters. Only then Jethro will fetch the data, perform JOINs and apply the relevant aggregations. This means that if a query is granular – accessing a small number of rows, it will take little time to perform regardless of the size of the dataset.

Key features of Jethro indexes

- Indexes are implemented as hierarchical compressed bitmaps and are stored in HDFS. An index-of-index is used to provide direct access to each index entry, so indexes do not need to be stored in memory.
- JOIN Indexes are used to create an index on a fact table using the values of a col from a dimension table. This way, when the dimension table col is used as a filter, no JOIN is needed, and an index can be used directly
- RANGE Indexes are used when an index col has a large number of values and queries typically filter by a range of values
- DATE Indexes are created automatically for TIMESTAMP columns and create entries for Years, Months, Days, HRs, etc. They are a similar RANGE indexes in the way they can cover a date-range using a single index entry to cover a full year instead of 365 individual daily entries.

Indexes in Jethro are Incrementally updated. When incremental data arrives, Jethro will append the current indexes. With this approach indexes are never locked and therefore even frequent loading of new data has no impact on ongoing query performance.

Query optimization in Jethro

The best way to speed up a query is to minimize the amount of work actually required to process it. Jethro uses a combination of optimization techniques ranging from query results, cubes and indexes. When a query arrives, it is first broken to individual subqueries and each is evaluated separately. The optimization sequence:

- Query Result Cache
  a. Results of past queries are saved in permanent storage. When a new query arrives, Jethro first checks if the same exact query already ran and its results were saved. If they were, the results are returned without any additional processing.
  b. Past results are automatically updated (incrementally or full replacement) when new data arrives.
- AutoCubes
  a. Jethro next checks if a cube aggregation including the query's dimensions and measures already exists. If not, Jethro checks if a key cube can be used to serve this query.
  b. The matching process uses the metadata of existing cubes. They are sorted by size so if a cube match is found it will use the smallest, and most efficient cube.

- Indexes

  a. If no cached results and no cubes are available, Jethro will use indexes to process the query. First, indexes are used to apply all filters, resulting in a working set – list of all needed rowids.

  b. Next, Jethro fetches the relevant column data for those rows and executes the SQL logic on that subset of the data.

  c. Many queries can be served directly from the indexes and don't even require accessing of the underlying data. A common example is BI tools sends a "SELECT DISTINCT col" query to get the list of distinct values for a filter. In Jethro such a query is a simple access to the list f values from the index.

- Execution optimization

  a. JOIN optimization – Jethro uses a variety of techniques to remove JOINs. These include Star Transformation, use of JOIN Indexes, conversion of OUTER to INNER JOINs and switching join table order after applying filters.

  b. Partition pruning – if one of the filters is also a partition key, Jethro will skip any processing (eg index search) of the unneeded partitions. This feature is not as impactful in Jethro as it is with Full-Scan as index filtering already minimizes the amount of data that is accessed.

  c. Multi-threading and pipelining – the execution is broken into small chunks of data (ie TupleSets) which are all processed in parallel across many threads. This approach optimizes utilization of system resources and also enables returning of initial results before the entire query is complete.

  d. GROUP BY optimization – using the information from the indexes, Jethro can predict the size of the resulting GROUP BY and choose the most memory-optimal aggregation algorithm for it.

  e. Subquery parallelism – Jethro treats each subquery independently and process them (when logic permits) in parallel.

- BI Tool specific optimizations

  a. Each BI tool has a certain pattern to the SQL queries it generates. While mostly optimized, often times such queries are complex and are hard to optimize, even by mature SQL engines.

  b. Jethro has incorporated many optimization rules that identifies specific query patterns and transforms them into a more optimal form. For example, Jethro will "push" filters from outer query into internal one or will be able to "extract" filters from a CASE statement and push them into a WHERE clause.

Summary

Big Data platforms are successfully used to store a growing amount of valuable enterprise data. An important way to utilize this data is to enable BI users to access this data in a self-service way. A key to the success of such BI on Big Data is to make it at interactive speed – as we know, slow BI is no BI. There are multiple approaches and tools used to improve BI performance on Big Data. They include Cubes, Indexes and Query Caching. The right solution must combine all 3 techniques as neither one (or two) by themselves will be able to address the full range of granular and aggregated BI queries. The winning solution must also automate the process of applying the acceleration strategies as relying on IT to manually define cubes or build indexes will requires tremendous resources and slow-down the agile nature of self-service BI. Jethro's solution is unique in its ability to provide fully automated Interactive BI on Big Data.

Thanks for reading!

Let's chat and discuss how you can accelerate your BI to the speed of thought.

+1 (844) 384-3844
info@jethro.io