Performance Benchmarking of Apache Pinot Indexing methods

FA 2023 CS 511 Project Report

Krut Patel
ksp8@illinois.edu

Vidya Kamath Pailodi
vidyak2@illinois.edu

Vivek Adrakatti
vga2@illinois.edu

ABSTRACT

This research presents a thorough benchmarking analysis of Apache Pinot's star-tree indexing techniques, crucial for optimizing real-time analytics performance. Apache Pinot[8], a leading open-source distributed data store, is renowned for its low latency and high throughput in handling large-scale data analytics. The effectiveness of Apache Pinot is significantly influenced by its indexing mechanisms, which are central to enhancing data retrieval speeds and overall system efficiency.

Our study conducts an examination of these indexing techniques, employing a series of benchmark tests across various configurations and environments. A key area of focus is the star-tree index, and its impact on query performance was rigorously evaluated. The research was conducted under varied data scales and query types, providing insights into the optimal use of this unique indexing strategy in different scenarios.

The findings of this study offer a nuanced understanding of the performance trends associated with Apache Pinot's indexing techniques. The results reveal insights into the trade-offs and benefits of different indexing configurations, providing valuable guidelines for practitioners in the field of data analytics.

This benchmarking project contributes to the domain of real-time data processing, delivering empirical evidence and practical knowledge that can aid in the optimization of real-time analytics systems. Our findings can assist professionals in the usage of the powerful star-tree index which is a complex feature whose intricacies and optimal utilization remain not fully understood within the user community.

1 INTRODUCTION

Background. Database indexing is essential in optimizing data retrieval efficiency. In the context of real-time analytics systems, indexing plays an important role in enabling fast and precise data access.

Apache Pinot [2] is a real-time distributed online analytical processing (OLAP) datastore designed to deliver scalable real-time analytics with low latency. It features ingestion from both streaming sources (Apache Kafka [1], and batch data sources (Apache Spark [3], Apache Hadoop [4], etc.) Pinot promises to ingest data in real-time and make it available for querying within seconds. It has been successfully used at LinkedIn to power 50+ user-facing products including "Who Viewed Profile", "Company Analytics", etc.

Problem Summary. The inspiration for this project lies in the growing importance of real-time analytics in various industry sectors, where timely data insights are crucial for decision-making. Apache Pinot is uniquely designed to cater to these requirements, by being able to execute OLAP queries with low latency. However, the effectiveness of Apache Pinot is heavily reliant on its indexing mechanisms, which optimize data retrieval by creating indexes that occupy disk space.

Apache Pinot makes use of various indexing techniques, such as i) Forward Index, ii) Inverted Index, iii) Bloom Filter, iv) Star Tree Index, and many others. Such indexes can be created either during ingestion or dynamically at any point.

Our research delves into an in-depth exploration of these indexing techniques. By conducting a series of benchmark tests under varied settings and scenarios, we aim to understand how different indexing configurations influence query speed and data throughput. Additionally, we investigate the trends in performance across different data scales and query complexities.

Since there exist multiple choices of indexing strategies, it might not be immediately clear to the programmer which index will serve them the best. Each index comes with its own set of configuration knobs that need to be tuned for optimal behavior. Indexes also inherently incur storage overhead, in terms of Pinot segments, that needs to be weighed against the potential benefits.

Difference from Existing Work. Apache Pinot provides a distributed columnar datastore with smart indexing and pre-aggregation techniques driving the low-latency and high performance. However, there is a need for a comprehensive study and benchmarking of the various indexing techniques provided by Apache Pinot. The lack of such a comprehensive study may lead users of the system to naively choose an unsuitable indexing method for the given data, and this may lead to increased query latency. The original Apache Pinot [8] paper only conducts a very brief comparison of query latencies between no indexes, Inverted Index, and Star Tree Index when used in the Apache Pinot System. This study focuses mainly on the less-known Star Tree Index and we aim to understand the various scenarios in which it would be advantageous to use this index.

Proposed Approach Summary. In this work, we measure the throughput and latency across various query types: group by queries, filter queries, group by queries with filtering, and distinct queries. Through the benchmark experiments, we aim to answer the following questions:

- What are the ideal usage scenarios for the star tree index?
- What is the impact of threshold (number of leaf records) on star tree index performance?
- What is the impact of threshold (number of leaf records) on star tree index size?

1.1 Apache Pinot

Apache Pinot is a real-time distributed OLAP datastore purpose-built for low-latency, high-throughput analytics, and suitable for user-facing analytical workloads [8].

Data Model. The data in Pinot consists of records in tables. These tables have a fixed schema, and the following data types are supported in Pinot: integers, floating point numbers, strings, and booleans. Tables are broken down into segments, which are collections of records. These segments are replicated to ensure data availability and fault tolerance. Pinot uses a columnar Data orientation in every segment. To reduce the size of the segments, various encoding strategies are used, including dictionary encoding and bit packing of values. Querying in Pinot is done through SQL.

Pinot Components. Pinot comprises four key components: controllers for data management, brokers for query handling, servers for data storage and processing, and minions for auxiliary tasks. It also relies on two external services: Zookeeper and a persistent object store. For cluster management, Pinot utilizes Apache Helix [5], a framework that handles partitions and replicas in distributed systems. Figure 1 shows the interaction between different components in a pinot cluster.

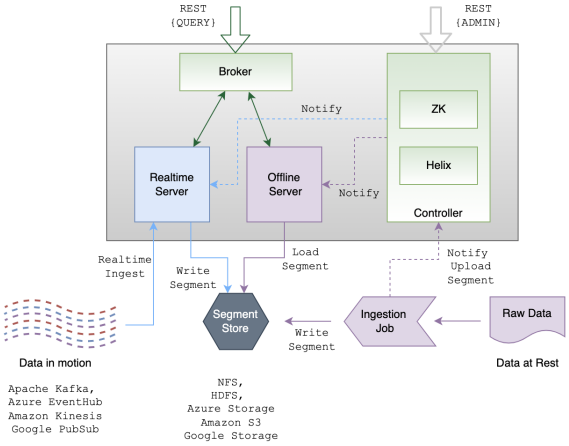

Figure 1: Architecture of Apache Pinot

1.2 Indexing Techniques

An index is a data organization set up to speed up the retrieval of data from tables.[9] An index in a database is a separate structure that occupies its own disk space. It doesn't alter the original table data but simply adds a new structure that references the table.

Apache Pinot currently supports 16 individual indexing techniques such as forward index, inverted index, sorted inverted index, range index, star-tree index, and a native text index. Each of these Indexing techniques can be applied depending on the data distribution and the intended query types. The wide variety of offered indexes is the reason why Pinot is capable of low latency and high query performance. By default, Pinot creates a dictionary-encoded forward index for each column. Here, we discuss the details of the indexing methods we tested in our benchmarking efforts.

Forward Index. The forward index in Apache Pinot, which is enabled by default, serves as a map linking document IDs, or row indices, to the respective column values of each row.
Apache Pinot offers three variations of the Forward Index:

- **Dictionary-encoded forward index with bit compression** In this method, every distinct value in a column is given an identifier, and a dictionary is created to link these identifiers to their respective values.
- **Sorted forward index with run-length encoding** If Columns are already sorted, this method stores pairs of start and end document IDs for each unique value instead of individual dictionary IDs for each document ID.
- **Raw value forward index** The raw value forward index stores actual values, not IDs, eliminating dictionary lookups and potentially improving query performance.

Inverted Index. We can define the forward index as a mapping from document IDs (also known as rows) to values. Similarly, an inverted index establishes a mapping from values to a set of document IDs, making it the "inverted" version of the forward index. It offers great query performance benefits when column-filtering operations are frequently used.
Apache Pinot offers three variations of the Inverted Index:

- **Bitmap inverted index** In this method, Pinot maintains a mapping from each value to a bitmap of rows.
- **Sorted inverted index** If Columns are already sorted, and the dictionary is enabled for that column, the sorted Forward Index behaves as an Inverted Index.

Range Index. The Range index in Apache Pinot is implemented as a variant of the inverted index. Under this indexing technique, a mapping is created from a range of values to a set of document IDs. It is useful to improve performance on queries that involve filtering over a range.

Native Text Index. Pinot uses a native text index that uses a customer text-indexing engine along with Pinot's inverted index to provide high performance while consuming less disk space. It also supports real-time text search.

Star-Tree Index. Star Tree Index [10] is a specialized tree structure that can store partial aggregates on a set of dimensions. It provides an intelligent way for users to control the amount of materialized computations. During index creation, the user can specify an upper bound on the number of records that can be scanned for any query that computes the covered aggregates. As we will see in Section 4.2, this gives the user a single knob to tune the space-time tradeoff of the index.

2 PROBLEM DEFINITION

The current release of Apache Pinot (v1.0) provides 13 different kinds of indexes, ranging from the classic dictionary encoding to a custom-built specialized technique called the Star-Tree Index. Each strategy has its own ideal use cases and brings its own set of constraints that can impact the entire data ingestion and query processing pipeline.

The Dictionary-based Forward Index is the default strategy applied to all ingested tables. However, it quickly deteriorates in performance as the number of distinct values in each column increases. For such cases, Pinot provides a Raw-Valued Forward Index that can provide raw access to underlying columns. However, using this precludes the ability to create any inverted indexes on such columns.

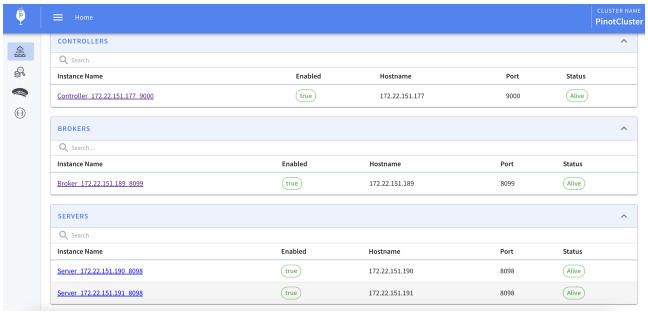

Figure 2: Screenshot of the Web Interface Displaying Configured Apache Pinot Cluster Components and Their Current Status

Pinot also has two kinds of Inverted Indexes - bitmap and sorted. Bitmap indexes can be useful for quickly filtering out unnecessary rows and sorted indexes further improve performance by improving data locality. However, the current documentation has no clear measurements or comparisons of the two.

The specialized StarTree index [7] is a way of providing materialized views based on the underlying data in an efficient manner. It can generate fast query times using preaggregated results for multiple dimensions, while still avoiding the exponential space blowup of traditional materialized views. Pinot exposes a configurable threshold between pre-aggregation and data scans to make this space-time tradeoff. This means that users need to carefully configure their indexes based on their specific use cases, which can be challenging.

3 METHODOLOGY

3.1 Setup for Apache Pinot

3.1.1 Infrastructure. The evaluation involved a setup comprising four Linux Virtual Machines (each with 8GB RAM), each allocated with specific roles: one VM was designated as the Controller, responsible for data management; another VM served as the Broker, managing communication and queries; while the remaining two VMs acted as Servers, handling data storage and processing. Figure 2 shows the cluster component details. We did not configure minions since they are mainly useful for data deletion and garbage collection.

3.2 Data generation

For our benchmarking efforts, we utilize a subset of the TPC-H Database. We run our queries on the dataset by varying the scale factor from 1, 10, and 100. With these datasets, we chose specific queries that are suited for the performance of each index and performed benchmarking to compare different indexing techniques and their effect on Pinot's performance.

However, we do not utilize the queries directly from the TPC-H benchmark itself, as most of these queries involve multi-table joins, which is not the focus of our study.

Specifically, we used the *lineitem* table from the TPC-H database. This is the table that contains the most number of rows and was thus chosen to allow us to experiment with a wider range of indexing methods.

3.2.1 Data Preparation. TPC-H data is generated using the dbgen tool at different scale factors, namely, 1, 10. Next, the data is processed, to convert into CSV format and is partitioned based on the `l_shipdate` column to facilitate distributed storage.

Additionally, for offline tables, Apache Pinot requires the columns to be pre-sorted prior to the ingestion for using the sorted forward index. Hence, the partitioned data is (secondary) sorted in ascending order on the `l_partkey` column.

This data is then ingested into the Apcahe Pinot tables, using various table configurations. The table configurations include various indexing properties that will be used for the creation of segments during the ingestion.

```
"tableIndexConfig": {
  "starTreeIndexConfigs": [
   {
      "dimensionsSplitOrder": [
        "l_receiptdate",
        "l_shipdate",
        "l_shipmode",
        "l_returnflag"
      ],
      "skipStarNodeCreationForDimensions": [],
      "functionColumnPairs": [
        "SUM__l_extendedprice",
        "SUM__l_discount",
        "SUM__l_quantity"
      ],
      "maxLeafRecords": 10
  }],
  ....
}
```

Figure 3: JSON file showing the configuration for star tree index creation.

3.2.2 Data Ingestion. To upload the data into pinot tables, batch ingestion is utilized as the method for loading data into the system. The ingestion process employs standalone local processing (alternative mechanisms include Spark or Hadoop) Specifically, DataIngestionJobs are utilized to facilitate ingestion as it is easily configurable via yaml config files. Alternative options include Minion-based Ingestion / Upload API.

3.2.3 Query types and Performance measures. For the evaluation, the following queries are taken into consideration.

- Group by query
- Aggregate query
- Filtering query
- Group by queries with filtering
- Distinct query

To evaluate the indexing techniques outlined in 1.2, we measure the performance by measuring the query response times as well as index sizes for varying queries and varying sizes of table, this is

explained in detail in 3.2.1. Specifically, for the star tree index, performance is also measured as a function of the star tree threshold.

```
Aggregate Query:

SELECT Sum(l_extendedprice),
       Sum(l_discount)
FROM   tpch_lineitem_10g

Filtering Queries:

SELECT Sum(l_extendedprice * l_discount) AS revenue
FROM   tpch_lineitem_10g
WHERE  l_discount BETWEEN .06 - 0.01 AND .06 + 0.010001
       AND l_quantity < 24

SELECT Sum(l_extendedprice)
FROM   tpch_lineitem_10g
WHERE  l_returnflag = 'R'

Group By Query:

SELECT Sum(l_extendedprice)
FROM   tpch_lineitem_10g
GROUP  BY l_shipdate

Distinct Query:

SELECT DISTINCT l_shipmode
FROM   tpch_lineitem_10g
```

Figure 4: Queries used in benchmarking study

4 EVALUATION

The Index configuration files and results are available in [6].

4.1 Star Tree Index evaluation

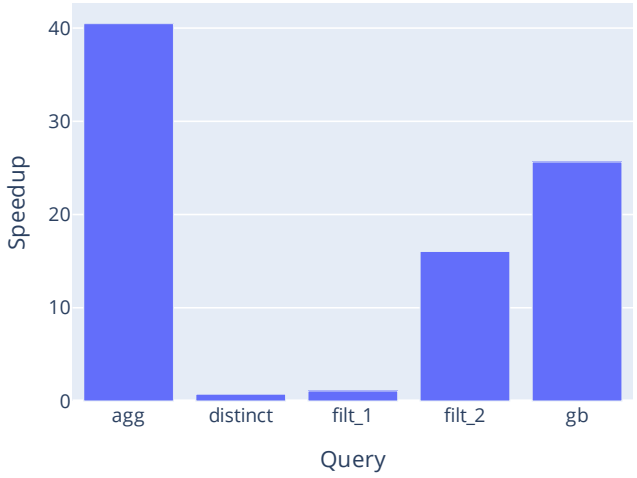

Figure 5: Speedup using StarTree Index vs non-indexed table across various queries

From Figure 5 we can see that the StarTree index can provide up to 40x speedup for many queries.

The agg query is the best case for this index - it needs to read a single record to fetch the precomputed query result, compared to a full table scan for the non-indexed case. Similarly, `filt_2` and gb queries compute results on columns with partial preaggregates already present in the index (`SUM(l_extendedprice)`). The index thus needs to perform very few record scans (based on the predicates) to compute the final results.

On the other hand, we see that `distinct` and `filt_1` queries are barely impacted by the presence of the index. In both cases, this is because the index simply does not store the partial aggregates required for the queries. `distinct` would require a large set of values to be stored, and `filt_1` would need to store a complex aggregate of two columns.

From this we can deduce the limitations of the star-tree index: i) it is best applied to queries which compute aggregates on numeric columns (to keep the index size small), and ii) all the aggregates required in the query have to be pre-specified during index creation itself.

4.2 Impact of Star Tree Threshold on Index Size

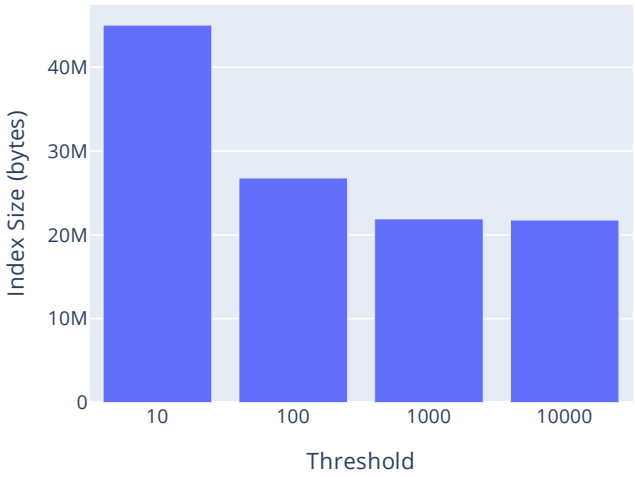

Figure 6: StarTree Index Size vs Threshold for 1GB data (600k records)

To study the impact of star tree threshold on index size, we consider four threshold values ranging from 10^1 to 10^4. Threshold values indicate the maximum number of leaf records in the star tree index. For this study, we considered the TPCH data with a scale factor of 1 and 10.

Low Threshold Values: As seen in figure 6, keeping a low threshold size expanded space requirements, allowing fewer record scans when computing final results.

Higher Threshold Values: Conversely, higher threshold values resulted in a reduction in index size. The coarser granularity minimized the number of pre-aggregated data points stored, leading to a smaller index footprint.

Further we see that after 1000 rows, increasing the threshold does not provide any storage benefits. This suggests there is avenue for future work here- the system should be able to automatically find the best threshold for a given latency.

4.3 Impact of star tree threshold on query performance

Unfortunately, we did not observe any impact of the row threshold on the runtime performance of any of the queries discussed previously. We suspect that this is due to the low scale of data that we were using, where the system overhead is enough to hide the query read times.

4.4 Other indexes

4.4.1 Inverted Bitmap index. We saw no significant difference in the performance of queries when using bitmap indexes vs the default. This is likely because Pinot already generates an inverted index on the partition column (`l_shipdate` in our case) by default. This is enough to answer most queries quickly and there is no added benefit of generating bitmap indexes.

4.4.2 Text index. Text index enables Regex type queries with the function, `TEXT_CONTAINS`, which supports text search on native text indices. We enabled this index for the `l_comment` column and were able to successfully execute a variety of text-based queries. However it is a limitation of Pinot that it cannot answer such queries in the absence of a native text index. As a result, we were unable to perform any comparison benchmarks for the text index in Pinot.

5 DISCUSSION & LIMITATIONS

This study highlights the potential advantages of using StarTree index in pinot. It shines when answering known aggregation queries on numeric columns, especially when latency is a hard constraint.

We studied the impact of threshold on star tree index size and performance. An interesting future study would be understanding the performance of indices as a function of the number of records selected by a filter. We expect that the star tree index performance remains independent of the selectivity because of the constant row threshold value.

Pinot has recently introduced a multi-stage query engine that allows running queries spanning multiple tables via JOINs. Our study can be extended to evaluate the impact of indices on such queries.

Further, due to limitations in space and compute in the VMs, we were unable to extend our study to a scale factor of 100 or larger. This would simulate real-world workloads and highlight the benefits of startree.

REFERENCES

[1] [n.d.]. *Apache Kafka™: An open-source distributed event streaming platform.* https://kafka.apache.org/

[2] [n.d.]. *Apache Pinot™: Realtime distributed OLAP datastore.* https://pinot.apache.org/

[3] [n.d.]. *Apache Spark™: Unified engine for large-scale data analytics.* https://spark.apache.org/

[4] [n.d.]. *The Apache™ Hadoop®: An open-source software for reliable, scalable, distributed computing.* https://hadoop.apache.org/

[5] [n.d.]. *A cluster management framework for partitioned and replicated distributed resources.* https://helix.apache.org/

[6] [n.d.]. *pinot-index-benchmarking.* https://github.com/VidyaKamath/pinot-index-benchmarking.git

[7] [n.d.]. *Star-Tree Index.* https://docs.pinot.apache.org/basics/indexing/star-tree-index

[8] Jean-François Im, Kishore Gopalakrishna, Subbu Subramaniam, Mayank Shrivastava, Adwait Tumbde, Xiaotian Jiang, Jennifer Dai, Seunghyun Lee, Neha Pawar, Jialiang Li, et al. 2018. Pinot: Realtime olap for 530 million users. In *Proceedings of the 2018 International Conference on Management of Data.* 583–594.

[9] Sam S Lightstone, Toby J Teorey, and Tom Nadeau. 2010. *Physical Database Design: the database professional's guide to exploiting indexes, views, storage, and more.* Morgan Kaufmann.

[10] Dong Xin, Jiawei Han, Xiaolei Li, and Benjamin W. Wah. 2003. Star-Cubing: Computing Iceberg Cubes by Top-Down and Bottom-Up Integration. In *Very Large Data Bases Conference.* https://api.semanticscholar.org/CorpusID:9273878