Pinot Documentation

Release 0.016

Pinot development team

Apr 26, 2019

Contents

CHAPTER 1

About Pinot

Pinot is a realtime distributed OLAP datastore, which is used at LinkedIn to deliver scalable real time analytics with low latency. It can ingest data from offline data sources (such as Hadoop and flat files) as well as streaming events (such as Kafka). Pinot is designed to scale horizontally, so that it can scale to larger data sets and higher query rates as needed.

1.1 What is it for (and not)?

Pinot is well suited for analytical use cases on immutable append-only data that require low latency between an event being ingested and it being available to be queried.

1.2 Key Features

- A column-oriented database with various compression schemes such as Run Length, Fixed Bit Length
- Pluggable indexing technologies - Sorted Index, Bitmap Index, Inverted Index
- Ability to optimize query/execution plan based on query and segment metadata .
- Near real time ingestion from streams and batch ingestion from Hadoop
- SQL like language that supports selection, aggregation, filtering, group by, order by, distinct queries on data.
- Support for multivalued fields
- Horizontally scalable and fault tolerant

Because of the design choices we made to achieve these goals, there are certain limitations in Pinot:

- Pinot is not a replacement for database i.e it cannot be used as source of truth store, cannot mutate data
- Not a replacement for search engine i.e Full text search, relevance not supported
- Query cannot span across multiple tables.

Pinot works very well for querying time series data with lots of Dimensions and Metrics. For example:

```
SELECT sum(clicks), sum(impressions) FROM AdAnalyticsTable
  WHERE ((daysSinceEpoch >= 17849 AND daysSinceEpoch <= 17856)) AND accountId IN␣
↪(123456789)
  GROUP BY daysSinceEpoch TOP 100
```

```
SELECT sum(impressions) FROM AdAnalyticsTable
  WHERE (daysSinceEpoch >= 17824 and daysSinceEpoch <= 17854) AND adveriserId =
↪'1234356789'
  GROUP BY daysSinceEpoch,advertiserId TOP 100
```

```
SELECT sum(cost) FROM AdAnalyticsTable GROUP BY advertiserId TOP 50
```

CHAPTER 2

Architecture

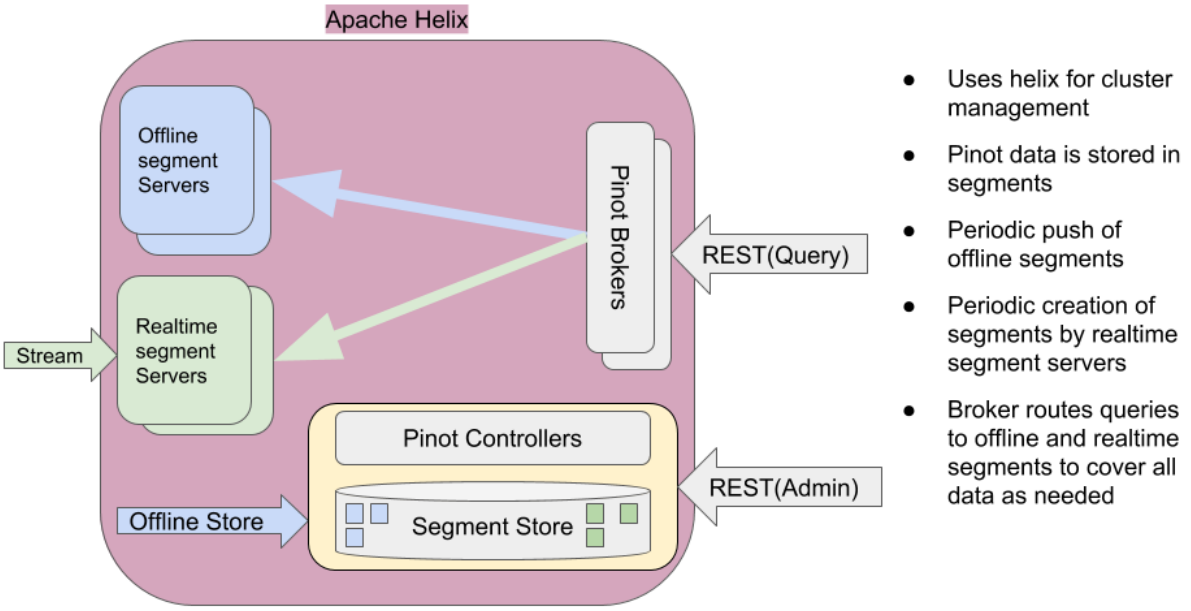

Fig. 1: Pinot Architecture Overview

2.1 Terminology

Table A table is a logical abstraction to refer to a collection of related data. It consists of columns and rows (documents).

Segment Data in table is divided into (horizontal) shards referred to as segments.

2.2 Pinot Components

Pinot Controller Manages other pinot components (brokers, servers) as well as controls assignment of tables/segments to servers.

Pinot Server Hosts one or more segments and serves queries from those segments

Pinot Broker Accepts queries from clients and routes them to one or more servers, and returns consolidated response to the client.

Pinot leverages Apache Helix for cluster management. Helix is a cluster management framework to manage replicated, partitioned resources in a distributed system. Helix uses Zookeeper to store cluster state and metadata.

Briefly, Helix divides nodes into three logical components based on their responsibilities:

Participant The nodes that host distributed, partitioned resources

Spectator The nodes that observe the current state of each Participant and use that information to access the resources. Spectators are notified of state changes in the cluster (state of a participant, or that of a partition in a participant).

Controller The node that observes and controls the Participant nodes. It is responsible for coordinating all transitions in the cluster and ensuring that state constraints are satisfied while maintaining cluster stability

Pinot Controller hosts Helix Controller, in addition to hosting REST APIs for Pinot cluster administration and data ingestion. There can be multiple instances of Pinot controller for redundancy. If there are multiple controllers, Pinot expects that all of them are configured with the same back-end storage system so that they have a common view of the segments (*e.g.* NFS). Pinot can use other storage systems such as HDFS or ADLS.

Pinot Servers are modeled as Helix Participants, hosting Pinot tables (referred to as *resources* in helix terminology). Segments of a table are modeled as Helix partitions (of a resource). Thus, a Pinot server hosts one or more helix partitions of one or more helix resources (*i.e.* one or more segments of one or more tables).

Pinot Brokers are modeled as Spectators. They need to know the location of each segment of a table (and each replica of the segments) and route requests to the appropriate server that hosts the segments of the table being queried. The broker ensures that all the rows of the table are queried exactly once so as to return correct, consistent results for a query. The brokers (or servers) may optimize to prune some of the segments as long as accuracy is not satisfied. In case of hybrid tables, the brokers ensure that the overlap between realtime and offline segment data is queried exactly once. Helix provides the framework by which spectators can learn the location (*i.e.* participant) in which each partition of a resource resides. The brokers use this mechanism to learn the servers that host specific segments of a table.

2.3 Pinot Tables

Pinot supports realtime, or offline, or hybrid tables. Data in Pinot tables is contained in the segments belonging to that table. A Pinot table is modeled as a Helix resource. Each segment of a table is modeled as a Helix Partition,

Table Schema defines column names and their metadata. Table configuration and schema is stored in zookeeper.

Offline tables ingest pre-built pinot-segments from external data stores, whereas Reatime tables ingest data from streams (such as Kafka) and build segments.

A hybrid Pinot table essentially has both realtime as well as offline tables. In such a table, offline segments may be pushed periodically (say, once a day). The retention on the offline table can be set to a high value (say, a few years) since segments are coming in on a periodic basis, whereas the retention on the realtime part can be small (say, a few days). Once an offline segment is pushed to cover a recent time period, the brokers automatically switch to using the

offline table for segments in _that_ time period, and use realtime table only to cover later segments for which offline data may not be available yet.

Note that the query does not know the existence of offline or realtime tables. It only specifies the table name in the query.

2.3.1 Ingesting Offline data

Segments for offline tables are constructed outside of Pinot, typically in Hadoop via map-reduce jobs and ingested into Pinot via REST API provided by the Controller. Pinot provides libraries to create Pinot segments out of input files in AVRO, JSON or CSV formats in a hadoop job, and push the constructed segments to the controlers via REST APIs.

When an Offline segment is ingested, the controller looks up the table's configuration and assigns the segment to the servers that host the table. It may assign multiple servers for each servers depending on the number of replicas configured for that table.

Pinot supports different segment assignment strategies that are optimized for various use cases.

Once segments are assigned, Pinot servers get notified via Helix to "host" the segment. The servers download the segments (as a cached local copy to serve queries) and load them into local memory. All segment data is maintained in memory as long as the server hosts that segment.

Once the server has loaded the segment, Helix notifies brokers of the availability of these segments. The brokers start include the new segments for queries. Brokers support different routing strategies depending on the type of table, the segment assignment strategy and the use case.

Data in offline segments are immmutable (Rows cannot be added, deleted, or modified). However, segments may be replaced modified data.

2.3.2 Ingesting Realtime Data

Segments for realtime tables are constructed by Pinot servers. The servers ingest rows from realtime streams (such as Kafka) until some completion threshold (such as number of rows, or a time threshold) and build a segment out of those rows. Depending on the type of ingestion mechanism used (stream or partition level), segments may be locally stored in the servers or in the controller's segment store.

Multiple servers may ingest the same data to increase availability and share query load.

Once a realtime segment is built and loaded the servers continue to consume from where they left off.

Realtime segments are immutable once they are completed. While realtime segments are being consumed they are mutable, in the sense that new rows can be added to them. Rows cannot be deleted from segments.

See Consuming and Indexing rows in Realtime for details.

2.4 Pinot Segments

A segment is laid out in a columnar format so that it can be directly mapped into memory for serving queries. Columns may be single or multi-valued. Column types may be STRING, INT, LONG, FLOAT, DOUBLE or BYTES. Columns may be declared to be metric or dimension (or specifically as a time dimension) in the schema.

Pinot uses dictionary encoding to store values as a dictionary ID. Columns may be configured to be "no-dictionary" column in which case raw values are stored. Dictionary IDs are encoded using minimum number of bits for efficient storage (_e.g._ a column with cardinality of 3 will use only 3 bits for each dictionary ID).

There is a forward index for each column compressed appropriately for efficient memory use. In addition, optional inverted indices can be configured for any set of columns. Inverted indices, while taking up more storage, offer better query performance.

Specialized indexes like StartTree index is also supported.

CHAPTER 3

Quick Demo

A quick way to get familiar with Pinot is to run the Pinot examples. The examples can be run either by compiling the code or by running the prepackaged Docker images.

To demonstrate Pinot, let's start a simple one node cluster, along with the required Zookeeper. This demo setup also creates a table, generates some Pinot segments, then uploads them to the cluster in order to make them queryable.

All of the setup is automated, so the only thing required at the beginning is to start the demonstration cluster.

3.1 Compiling the code

One can also run the Pinot demonstration by checking out the code on GitHub, compiling it, and running it. Compiling Pinot requires JDK 8 or later and Apache Maven 3.

1. Check out the code from GitHub (https://github.com/apache/incubator-pinot)
2. With Maven installed, run `mvn install package -DskipTests -Pbin-dist` in the directory in which you checked out Pinot.
3. Make the generated scripts executable `cd pinot-distribution/target/apache-pinot-incubating-<version>-SNAPSHOT-bin; chmod +x bin/*.sh`

3.2 Trying out Offline quickstart demo

To run the demo with compiled code: `bin/quick-start-offline.sh`

Once the Pinot cluster is running, you can query it by going to http://localhost:9000/query/

You can also use the REST API to query Pinot, as well as the Java client. As this is outside of the scope of this introduction, the reference documentation to use the Pinot client APIs is in the *Executing queries via REST API on the Broker* section.

Pinot uses PQL, a SQL-like query language, to query data. Here are some sample queries:

```
/*Total number of documents in the table*/
SELECT count(*) FROM baseballStats LIMIT 0

/*Top 5 run scorers of all time*/
SELECT sum('runs') FROM baseballStats GROUP BY playerName TOP 5 LIMIT 0

/*Top 5 run scorers of the year 2000*/
SELECT sum('runs') FROM baseballStats WHERE yearID=2000 GROUP BY playerName TOP 5␣
↪LIMIT 0

/*Top 10 run scorers after 2000*/
SELECT sum('runs') FROM baseballStats WHERE yearID>=2000 GROUP BY playerName

/*Select playerName,runs,homeRuns for 10 records from the table and order them by␣
↪yearID*/
SELECT playerName,runs,homeRuns FROM baseballStats ORDER BY yearID LIMIT 10
```

The full reference for the PQL query language is present in the *PQL* section of the Pinot documentation.

3.3 Trying out Realtime quickstart demo

Pinot can ingest data from streaming sources such as Kafka.

To run the demo with compiled code: `bin/quick-start-realtime.sh`

Once started, the demo will start Kafka, create a Kafka topic, and create a realtime Pinot table. Once created, Pinot will start ingesting events from the Kafka topic into the table. The demo also starts a consumer that consumes events from the Meetup API and pushes them into the Kafka topic that was created, causing new events modified on Meetup to show up in Pinot.

To show new events appearing, one can run `SELECT * FROM meetupRsvp ORDER BY mtime DESC LIMIT 50` repeatedly, which shows the last events that were ingested by Pinot.

CHAPTER 4

Reference

4.1 PQL

- PQL is a derivative of SQL derivative that supports selection, projection, aggregation, grouping aggregation. There is no support for Joins.
- Specifically, for Pinot:
  - Grouping keys always appear in query results, even if not requested
  - Aggregations are computed in parallel
  - Results of aggregations with large amounts of group keys (>1M) are approximated
  - `ORDER BY` only works for selection queries, for aggregations one must use the `TOP` keyword

4.1.1 PQL Examples

The Pinot Query Language (PQL) is very similar to standard SQL:

```
SELECT COUNT(*) FROM myTable
```

4.1.2 Aggregation

```
SELECT COUNT(*), MAX(foo), SUM(bar) FROM myTable
```

4.1.3 Grouping on Aggregation

```
SELECT MIN(foo), MAX(foo), SUM(foo), AVG(foo) FROM myTable
  GROUP BY bar, baz TOP 50
```

4.1.4 Filtering

```
SELECT COUNT(*) FROM myTable
  WHERE foo = 'foo'
  AND bar BETWEEN 1 AND 20
  OR (baz < 42 AND quux IN ('hello', 'goodbye') AND quuux NOT IN (42, 69))
```

4.1.5 Selection (Projection)

```
SELECT * FROM myTable
  WHERE quux < 5
  LIMIT 50
```

4.1.6 Ordering on Selection

```
SELECT foo, bar FROM myTable
  WHERE baz > 20
  ORDER BY bar DESC
  LIMIT 100
```

4.1.7 Pagination on Selection

Note: results might not be consistent if column ordered by has same value in multiple rows.

```
SELECT foo, bar FROM myTable
  WHERE baz > 20
  ORDER BY bar DESC
  LIMIT 50, 100
```

4.1.8 Wild-card match (in WHERE clause only)

To count rows where the column `airlineName` starts with `U`

```
SELECT count(*) FROM SomeTable
  WHERE regexp_like(airlineName, '^U.*')
  GROUP BY airlineName TOP 10
```

4.1.9 Examples with UDF

As of now, functions have to be implemented within Pinot. Injecting functions is not allowed yet. The examples below demonstrate the use of UDFs

```
SELECT count(*) FROM myTable
  GROUP BY timeConvert(timeColumnName, 'SECONDS', 'DAYS')

SELECT count(*) FROM myTable
  GROUP BY div(tim
```

4.1.10 PQL Specification

SELECT

The select statement is as follows

```
SELECT <outputColumn> (, outputColumn, outputColumn,...)
  FROM <tableName>
   (WHERE ... | GROUP BY ... | ORDER BY ... | TOP ... | LIMIT ...)
```

`outputColumn` can be `*` to project all columns, columns (`foo`, `bar`, `baz`) or aggregation functions like (`MIN(foo)`, `MAX(bar)`, `AVG(baz)`).

Supported aggregations on single-value columns

- `COUNT`
- `MIN`
- `MAX`
- `SUM`
- `AVG`
- `MINMAXRANGE`
- `DISTINCTCOUNT`
- `DISTINCTCOUNTHLL`
- `FASTHLL`
- `PERCENTILE[0-100]`: e.g. `PERCENTILE5`, `PERCENTILE50`, `PERCENTILE99`, etc.
- `PERCENTILEEST[0-100]`: e.g. `PERCENTILEEST5`, `PERCENTILEEST50`, `PERCENTILEEST99`, etc.

Supported aggregations on multi-value columns

- `COUNTMV`
- `MINMV`
- `MAXMV`
- `SUMMV`
- `AVGMV`
- `MINMAXRANGEMV`
- `DISTINCTCOUNTMV`
- `DISTINCTCOUNTHLLMV`
- `FASTHLLMV`
- `PERCENTILE[0-100]MV`: e.g. `PERCENTILE5MV`, `PERCENTILE50MV`, `PERCENTILE99MV`, etc.
- `PERCENTILEEST[0-100]MV`: e.g. `PERCENTILEEST5MV`, `PERCENTILEEST50MV`, `PERCENTILEEST99MV`, etc.

WHERE

Supported predicates are comparisons with a constant using the standard SQL operators (=, <, <=, >, >=, <>, '!=') , range comparisons using `BETWEEN` (`foo BETWEEN 42 AND 69`), set membership (`foo IN (1, 2, 4, 8)`) and exclusion (`foo NOT IN (1, 2, 4, 8)`). For `BETWEEN`, the range is inclusive.

Comparison with a regular expression is supported using the regexp_like function, as in `WHERE regexp_like(columnName, 'regular expression')`

GROUP BY

The `GROUP BY` clause groups aggregation results by a list of columns, or transform functions on columns (see below)

ORDER BY

The `ORDER BY` clause orders selection results by a list of columns. PQL supports ordering `DESC` or `ASC`.

TOP

The `TOP n` clause causes the 'n' largest group results to be returned. If not specified, the top 10 groups are returned.

LIMIT

The `LIMIT n` clause causes the selection results to contain at most 'n' results. The `LIMIT a, b` clause paginate the selection results from the 'a' th results and return at most 'b' results.

Transform Function in Aggregation and Grouping

In aggregation and grouping, each column can be transformed from one or multiple columns. For example, the following query will calculate the maximum value of column `foo` divided by column `bar` grouping on the column `time` converted form time unit `MILLISECONDS` to `SECONDS`:

```
SELECT MAX(DIV(foo, bar) FROM myTable
  GROUP BY TIMECONVERT(time, 'MILLISECONDS', 'SECONDS')
```

Supported transform functions

ADD Sum of at least two values

SUB Difference between two values

MULT Product of at least two values

DIV Quotient of two values

TIMECONVERT Takes 3 arguments, converts the value into another time unit. E.g. `TIMECONVERT(time, 'MILLISECONDS', 'SECONDS')`

DATETIMECONVERT Takes 4 arguments, converts the value into another date time format, and buckets time based on the given time granularity. *e.g.* `DATETIMECONVERT(date, '1:MILLISECONDS:EPOCH', '1:SECONDS:EPOCH', '15:MINUTES')`

VALUEIN Takes at least 2 arguments, where the first argument is a multi-valued column, and the following arguments are constant values. The transform function will filter the value from the multi-valued column with the given constant values. The VALUEIN transform function is especially useful when the same multi-valued column is both filtering column and grouping column. *e.g.* VALUEIN(mvColumn, 3, 5, 15)

4.1.11 Differences with SQL

- JOIN is not supported
- Use TOP instead of LIMIT for truncation
- LIMIT n has no effect in grouping queries, should use TOP n instead. If no TOP n defined, PQL will use TOP 10 as default truncation setting.
- No need to select the columns to group with.

The following two queries are both supported in PQL, where the non-aggregation columns are ignored.

```
SELECT MIN(foo), MAX(foo), SUM(foo), AVG(foo) FROM mytable
  GROUP BY bar, baz
  TOP 50

SELECT bar, baz, MIN(foo), MAX(foo), SUM(foo), AVG(foo) FROM mytable
  GROUP BY bar, baz
  TOP 50
```

- The results will always order by the aggregated value (descending).

The results for query:

```
SELECT MIN(foo), MAX(foo) FROM myTable
  GROUP BY bar
  TOP 50
```

will be the same as the combining results from the following queries:

```
SELECT MIN(foo) FROM myTable
  GROUP BY bar
  TOP 50
SELECT MAX(foo) FROM myTable
  GROUP BY bar
  TOP 50
```

where we don't put the results for the same group together.

4.2 Index Techniques

Pinot currently supports the following index techniques, where each of them have their own advantages in different query scenarios.

4.2.1 Forward Index

Dictionary-Encoded Forward Index with Bit Compression

For each unique value from a column, we assign an id to it, and build a dictionary from the id to the value. Then in the forward index, we only store the bit-compressed ids instead of the values.

With few number of unique values, dictionary-encoding can significantly improve the space efficiency of the storage.

Raw Value Forward Index

In contrast to the dictionary-encoded forward index, raw value forward index directly stores values instead of ids.

Without the dictionary, the dictionary lookup step can be skipped for each value fetch. Also, the index can take advantage of the good locality of the values, thus improve the performance of scanning large number of values.

Sorted Forward Index with Run-Length Encoding

On top of the dictionary-encoding, all the values are sorted, so sorted forward index has the advantages of both good compression and data locality.

Sorted forward index can also be used as inverted index.

4.2.2 Inverted Index (only available with dictionary-encoded indexes)

Bitmap Inverted Index

Pinot maintains a map from each value to a bitmap, which makes value lookup to be constant time.

Sorted Inverted Index

Because the values are sorted, the sorted forward index can directly be used as inverted index, with constant time lookup and good data locality.

4.2.3 Advanced Index

Star-Tree Index

Unlike other index techniques which work on single column, Star-Tree index is built on multiple columns, and utilize the pre-aggregated results to significantly reduce the number of values to be processed, thus improve the query performance.

4.3 Executing queries via REST API on the Broker

The Pinot REST API can be accessed by invoking `POST` operation witha a JSON body containing the parameter `pql` to the `/query` URI endpoint on a broker. Depending on the type of query, the results can take different shapes. The examples below use curl.

4.3.1 Aggregation

```
curl -X POST -d '{"pql":"select count(*) from flights"}' http://localhost:8099/query


{
 "traceInfo":{},
 "numDocsScanned":17,
 "aggregationResults":[
    {
       "function":"count_star",
       "value":"17"
    }
 ],
 "timeUsedMs":27,
 "segmentStatistics":[],
 "exceptions":[],
 "totalDocs":17
}
```

4.3.2 Aggregation with grouping

```
curl -X POST -d '{"pql":"select count(*) from flights group by Carrier"}' http://
↪localhost:8099/query


{
 "traceInfo":{},
 "numDocsScanned":23,
 "aggregationResults":[
    {
       "groupByResult":[
          {
             "value":"10",
             "group":["AA"]
          },
          {
             "value":"9",
             "group":["VX"]
          },
          {
             "value":"4",
             "group":["WN"]
          }
       ],
       "function":"count_star",
       "groupByColumns":["Carrier"]
    }
 ],
 "timeUsedMs":47,
 "segmentStatistics":[],
 "exceptions":[],
 "totalDocs":23
}
```

4.3.3 Selection

```
curl -X POST -d '{"pql":"select * from flights limit 3"}' http://localhost:8099/query


{
 "selectionResults":{
    "columns":[
       "Cancelled",
       "Carrier",
       "DaysSinceEpoch",
       "Delayed",
       "Dest",
       "DivAirports",
       "Diverted",
       "Month",
       "Origin",
       "Year"
    ],
    "results":[
       [
          "0",
          "AA",
          "16130",
          "0",
          "SFO",
          [],
          "0",
          "3",
          "LAX",
          "2014"
       ],
       [
          "0",
          "AA",
          "16130",
          "0",
          "LAX",
          [],
          "0",
          "3",
          "SFO",
          "2014"
       ],
       [
          "0",
          "AA",
          "16130",
          "0",
          "SFO",
          [],
          "0",
          "3",
          "LAX",
          "2014"
       ]
    ]
```

(continues on next page)

(continued from previous page)

```
 },
 "traceInfo":{},
 "numDocsScanned":3,
 "aggregationResults":[],
 "timeUsedMs":10,
 "segmentStatistics":[],
 "exceptions":[],
 "totalDocs":102
}
```

4.3.4 Java

The Pinot client API is similar to JDBC, although there are some differences, due to how Pinot behaves. For example, a query with multiple aggregation function will return one result set per aggregation function, as they are computed in parallel.

Connections to Pinot are created using the ConnectionFactory class' utility methods to create connections to a Pinot cluster given a Zookeeper URL, a Java Properties object or a list of broker addresses to connect to.

```
Connection connection = ConnectionFactory.fromZookeeper
  ("some-zookeeper-server:2191/zookeeperPath");

Connection connection = ConnectionFactory.fromProperties("demo.properties");

Connection connection = ConnectionFactory.fromHostList
  ("some-server:1234", "some-other-server:1234", ...);
```

Queries can be sent directly to the Pinot cluster using the Connection.execute(java.lang.String) and Connection.executeAsync(java.lang.String) methods of Connection.

```
ResultSetGroup resultSetGroup = connection.execute("select * from foo...");
Future<ResultSetGroup> futureResultSetGroup = connection.executeAsync
  ("select * from foo...");
```

Queries can also use a PreparedStatement to escape query parameters:

```
PreparedStatement statement = connection.prepareStatement
  ("select * from foo where a = ?");
statement.setString(1, "bar");

ResultSetGroup resultSetGroup = statement.execute();
Future<ResultSetGroup> futureResultSetGroup = statement.executeAsync();
```

In the case of a selection query, results can be obtained with the various get methods in the first ResultSet, obtained through the getResultSet(int) method:

```
ResultSet resultSet = connection.execute
  ("select foo, bar from baz where quux = 'quuux'").getResultSet(0);

for (int i = 0; i < resultSet.getRowCount(); ++i) {
  System.out.println("foo: " + resultSet.getString(i, 0));
  System.out.println("bar: " + resultSet.getInt(i, 1));
}

resultSet.close();
```

In the case of aggregation, each aggregation function is within its own ResultSet:

```
ResultSetGroup resultSetGroup = connection.execute("select count(*) from foo");

ResultSet resultSet = resultSetGroup.getResultSet(0);
System.out.println("Number of records: " + resultSet.getInt(0));
resultSet.close();
```

There can be more than one ResultSet, each of which can contain multiple results grouped by a group key.

```
ResultSetGroup resultSetGroup = connection.execute
    ("select min(foo), max(foo) from bar group by baz");

System.out.println("Number of result groups:" +
    resultSetGroup.getResultSetCount(); // 2, min(foo) and max(foo)

ResultSet minResultSet = resultSetGroup.getResultSet(0);
for(int i = 0; i < minResultSet.length(); ++i) {
    System.out.println("Minimum foo for " + minResultSet.getGroupKeyString(i, 1) +
        ": " + minResultSet.getInt(i));
}

ResultSet maxResultSet = resultSetGroup.getResultSet(1);
for(int i = 0; i < maxResultSet.length(); ++i) {
    System.out.println("Maximum foo for " + maxResultSet.getGroupKeyString(i, 1) +
        ": " + maxResultSet.getInt(i));
}

resultSet.close();
```

4.4 Managing Pinot via REST API on the Controller

TODO : Remove this section altogether and find a place somewhere for a pointer to the management API. Maybe in the 'Running pinot in production' section?

There is a REST API which allows management of tables, tenants, segments and schemas. It can be accessed by going to `http://[controller host]/help` which offers a web UI to do these tasks, as well as document the REST API.

It can be used instead of the `pinot-admin.sh` commands to automate the creation of tables and tenants.

4.5 Creating Pinot segments

Pinot segments can be created offline on Hadoop, or via command line from data files. Controller REST endpoint can then be used to add the segment to the table to which the segment belongs.

4.5.1 Creating segments using hadoop

To create Pinot segments on Hadoop, a workflow can be created to complete the following steps:

1. Pre-aggregate, clean up and prepare the data, writing it as Avro format files in a single HDFS directory
2. Create segments

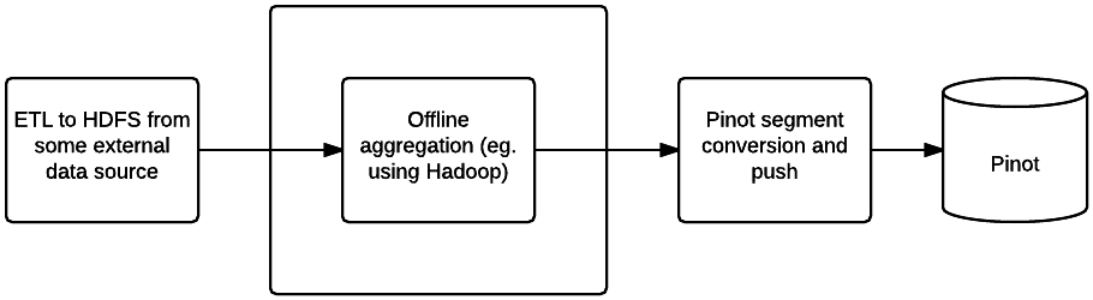

Fig. 1: Offline Pinot workflow

3. Upload segments to the Pinot cluster

Step one can be done using your favorite tool (such as Pig, Hive or Spark), Pinot provides two MapReduce jobs to do step two and three.

Configuring the job

Create a job properties configuration file, such as one below:

```
# === Index segment creation job config ===

# path.to.input: Input directory containing Avro files
path.to.input=/user/pinot/input/data

# path.to.output: Output directory containing Pinot segments
path.to.output=/user/pinot/output

# path.to.schema: Schema file for the table, stored locally
path.to.schema=flights-schema.json

# segment.table.name: Name of the table for which to generate segments
segment.table.name=flights

# === Segment tar push job config ===

# push.to.hosts: Comma separated list of controllers host names to which to push
push.to.hosts=controller_host_0,controller_host_1

# push.to.port: The port on which the controller runs
push.to.port=8888
```

Executing the job

The Pinot Hadoop module contains a job that you can incorporate into your workflow to generate Pinot segments.

```
mvn clean install -DskipTests -Pbuild-shaded-jar
hadoop jar pinot-hadoop-0.016-shaded.jar SegmentCreation job.properties
```

You can then use the SegmentTarPush job to push segments via the controller REST API.

```
hadoop jar pinot-hadoop-0.016-shaded.jar SegmentTarPush job.properties
```

4.5.2 Creating Pinot segments outside of Hadoop

Here is how you can create Pinot segments from standard formats like CSV/JSON.

1. Follow the steps described in the section on *Compiling the code* to build pinot. Locate `pinot-admin.sh` in `pinot-tools/target/pinot-tools-pkg/bin/pinot-admin.sh`.
2. Create a top level directory containing all the CSV/JSON files that need to be converted into segments.
3. The file name extensions are expected to be the same as the format name (*i.e* `.csv`, or `.json`), and are case insensitive. Note that the converter expects the `.csv` extension even if the data is delimited using tabs or spaces instead.
4. Prepare a schema file describing the schema of the input data. The schema needs to be in JSON format. See example later in this section.
5. **Specifically for CSV format, an optional csv config file can be provided (also in JSON format). This is used to configure pa**
   A detailed description of this follows below.

Run the pinot-admin command to generate the segments. The command can be invoked as follows. Options within "[ ]" are optional. For -format, the default value is AVRO.

```
bin/pinot-admin.sh CreateSegment -dataDir <input_data_dir> [-format [CSV/JSON/AVRO]]␣
↪[-readerConfigFile <csv_config_file>] [-generatorConfigFile <generator_config_file>
↪] -segmentName <segment_name> -schemaFile <input_schema_file> -tableName <table_
↪name> -outDir <output_data_dir> [-overwrite]
```

To configure various parameters for CSV a config file in JSON format can be provided. This file is optional, as are each of its parameters. When not provided, default values used for these parameters are described below:

1. fileFormat: Specify one of the following. Default is EXCEL.

   ##. EXCEL ##. MYSQL ##. RFC4180 ##. TDF

1. header: If the input CSV file does not contain a header, it can be specified using this field. Note, if this is specified, then the input file is expected to not contain the header row, or else it will result in parse error. The columns in the header must be delimited by the same delimiter character as the rest of the CSV file.
2. delimiter: Use this to specify a delimiter character. The default value is ",".
3. dateFormat: If there are columns that are in date format and need to be converted into Epoch (in milliseconds), use this to specify the format. Default is "mm-dd-yyyy".
4. dateColumns: If there are multiple date columns, use this to list those columns.

Below is a sample config file.

```
{
  "fileFormat" : "EXCEL",
  "header" : "col1,col2,col3,col4",
  "delimiter" : "\t",
  "dateFormat" : "mm-dd-yy"
  "dateColumns" : ["col1", "col2"]
}
```

Sample Schema:

```
{
  "dimensionFieldSpecs" : [
    {
      "dataType" : "STRING",
      "delimiter" : null,
      "singleValueField" : true,
      "name" : "name"
    },
    {
      "dataType" : "INT",
      "delimiter" : null,
      "singleValueField" : true,
      "name" : "age"
    }
  ],
  "timeFieldSpec" : {
    "incomingGranularitySpec" : {
      "timeType" : "DAYS",
      "dataType" : "LONG",
      "name" : "incomingName1"
    },
    "outgoingGranularitySpec" : {
      "timeType" : "DAYS",
      "dataType" : "LONG",
      "name" : "outgoingName1"
    }
  },
  "metricFieldSpecs" : [
    {
      "dataType" : "FLOAT",
      "delimiter" : null,
      "singleValueField" : true,
      "name" : "percent"
    }
   ]
  },
  "schemaName" : "mySchema",
}
```

Pushing segments to Pinot

You can use curl to push a segment to pinot:

```
curl -X POST -F segment=@<segment-tar-file-path> http://controllerHost:controllerPort/
↪segments
```

Alternatively you can use the pinot-admin.sh utility to upload one or more segments:

```
pinot-tools/target/pinot-tools-pkg/bin//pinot-admin.sh UploadSegment -controllerHost
↪<hostname> -controllerPort <port> -segmentDir <segmentDirectoryPath>
```

The command uploads all the segments found in `segmentDirectoryPath`. The segments could be either tar-compressed (in which case it is a file under `segmentDirectoryPath`) or uncompressed (in which case it is a directory under `segmentDirectoryPath`).

4.6 Running Pinot in production

4.6.1 Installing Pinot

Requirements

- Java 8+
- Several nodes with enough memory
- A working installation of Zookeeper

Recommended environment

- Shared storage infrastructure (such as NFS)
- Regular Zookeeper backups
- HTTP load balancers (such as nginx/haproxy)

4.6.2 Deploying Pinot

Direct deployment of Pinot

Deployment of Pinot on Kubernetes

4.6.3 Managing Pinot

Creating tables

Updating tables

Uploading data

Configuring realtime data ingestion

Monitoring Pinot

CHAPTER 5

Customizing Pinot

5.1 Pluggable Streams

Prior to commit ba9f2d, Pinot was only able to support reading from Kafka stream.

Pinot now enables its users to write plug-ins to read from pub-sub streams other than Kafka. (Please refer to Issue #2583)

Some of the streams for which plug-ins can be added are:

- Amazon kinesis
- Azure Event Hubs
- LogDevice
- Pravega
- Pulsar

You may encounter some limitations either in Pinot or in the stream system while developing plug-ins. Please feel free to get in touch with us when you start writing a stream plug-in, and we can help you out. We are open to receiving PRs in order to improve these abstractions if they do not work for a certain stream implementation.

Refer to Consuming and Indexing rows in Realtime for details on how Pinot consumes streaming data.

5.1.1 Requirements to support Stream Level (High Level) consumers

The stream should provide the following guarantees:

- Exactly once delivery (unless restarting from a checkpoint) for each consumer of the stream.
- (Optionally) support mechanism to split events (in some arbitrary fashion) so that each event in the stream is delivered exactly to one host out of set of hosts.
- Provide ways to save a checkpoint for the data consumed so far. If the stream is partitioned, then this checkpoint is a vector of checkpoints for events consumed from individual partitions.

- The checkpoints should be recorded only when Pinot makes a call to do so.
- The consumer should be able to start consumption from one of:
  - latest avaialble data
  - earliest available data
  - last saved checkpoint

5.1.2 Requirements to support Partition Level (Low Level) consumers

While consuming rows at a partition level, the stream should support the following properties:

- Stream should provide a mechanism to get the current number of partitions.
- Each event in a partition should have a unique offset that is not more than 64 bits long.
- Refer to a partition as a number not exceeding 32 bits long.
- Stream should provide the following mechanisms to get an offset for a given partition of the stream:
  - get the offset of the oldest event available (assuming events are aged out periodically) in the partition.
  - get the offset of the most recent event published in the partition
  - (optionally) get the offset of an event that was published at a specified time
- Stream should provide a mechanism to consume a set of events from a partition starting from a specified offset.
- Events with higher offsets should be more recent (the offsets of events need not be contiguous)

In addition, we have an operational requirement that the number of partitions should not be reduced over time.

5.1.3 Stream plug-in implementation

In order to add a new type of stream (say,Foo) implement the following classes:

1. FooConsumerFactory extends StreamConsumerFactory
2. FooPartitionLevelConsumer implements PartitionLevelConsumer
3. FooStreamLevelConsumer implements StreamLevelConsumer
4. FooMetadataProvider implements StreamMetadataProvider
5. FooMessageDecoder implements StreamMessageDecoder

Depending on stream level or partition level, your implementation needs to include StreamLevelConsumer or PartitionLevelConsumer.

The properties for the stream implementation are to be set in the table configuration, inside streamConfigs section.

Use the `streamType` property to define the stream type. For example, for the implementation of stream `foo`, set the property `"streamType" : "foo"`.

The rest of the configuration properties for your stream should be set with the prefix `"stream.foo"`. Be sure to use the same suffix for: (see examples below):

- topic
- consumer type
- stream consumer factory
- offset

- decoder class name
- decoder properties
- connnection timeout
- fetch timeout

All values should be strings. For example:

```
"streamType" : "foo",
"stream.foo.topic.name" : "SomeTopic",
"stream.foo.consumer.type": "lowlevel",
"stream.foo.consumer.factory.class.name": "fully.qualified.pkg.
↪ConsumerFactoryClassName",
"stream.foo.consumer.prop.auto.offset.reset": "largest",
"stream.foo.decoder.class.name" : "fully.qualified.pkg.DecoderClassName",
"stream.foo.decoder.prop.a.decoder.property" : "decoderPropValue",
"stream.foo.connection.timeout.millis" : "10000", // default 30_000
"stream.foo.fetch.timeout.millis" : "10000" // default 5_000
```

You can have additional properties that are specific to your stream. For example:

```
"stream.foo.some.buffer.size" : "24g"
```

In addition to these properties, you can define thresholds for the consuming segments:

- rows threshold
- time threshold

The properties for the thresholds are as follows:

```
"realtime.segment.flush.threshold.size" : "100000"
"realtime.segment.flush.threshold.time" : "6h"
```

An example of this implementation can be found in the KafkaConsumerFactory, which is an implementation for the kafka stream.

5.2 Segment Fetchers

When pinot segment files are created in external systems (hadoop/spark/etc), there are several ways to push those data to pinot Controller and Server:

1. push segment to shared NFS and let pinot pull segment files from the location of that NFS.
2. push segment to a Web server and let pinot pull segment files from the Web server with http/https link.
3. push segment to HDFS and let pinot pull segment files from HDFS with hdfs location uri.
4. push segment to other system and implement your own segment fetcher to pull data from those systems.

The first two options should be supported out of the box with pinot package. As long your remote jobs send Pinot controller with the corresponding URI to the files it will pick up the file and allocate it to proper Pinot Servers and brokers. To enable Pinot support for HDFS, you will need to provide Pinot Hadoop configuration and proper Hadoop dependencies.

5.2.1 HDFS segment fetcher configs

In your Pinot controller/server configuration, you will need to provide the following configs:

```
pinot.controller.segment.fetcher.hdfs.hadoop.conf.path=`<file path to hadoop conf␣
↪folder>
```

or

```
pinot.server.segment.fetcher.hdfs.hadoop.conf.path=`<file path to hadoop conf folder>
```

This path should point the local folder containing `core-site.xml` and `hdfs-site.xml` files from your Hadoop installation

```
pinot.controller.segment.fetcher.hdfs.hadoop.kerberos.principle=`<your kerberos␣
↪principal>
pinot.controller.segment.fetcher.hdfs.hadoop.kerberos.keytab=`<your kerberos keytab>
```

or

```
pinot.server.segment.fetcher.hdfs.hadoop.kerberos.principle=`<your kerberos principal>
pinot.server.segment.fetcher.hdfs.hadoop.kerberos.keytab=`<your kerberos keytab>
```

These two configs should be the corresponding Kerberos configuration if your Hadoop installation is secured with Kerberos. Please check Hadoop Kerberos guide on how to generate Kerberos security identification.

You will also need to provide proper Hadoop dependencies jars from your Hadoop installation to your Pinot startup scripts.

5.2.2 Push HDFS segment to Pinot Controller

To push HDFS segment files to Pinot controller, you just need to ensure you have proper Hadoop configuration as we mentioned in the previous part. Then your remote segment creation/push job can send the HDFS path of your newly created segment files to the Pinot Controller and let it download the files.

For example, the following curl requests to Controller will notify it to download segment files to the proper table:

```
curl -X POST -H "UPLOAD_TYPE:URI" -H "DOWNLOAD_URI:hdfs://nameservice1/hadoop/path/to/
↪segment/file.gz" -H "content-type:application/json" -d '' localhost:9000/segments
```

5.2.3 Implement your own segment fetcher for other systems

You can also implement your own segment fetchers for other file systems and load into Pinot system with an external jar. All you need to do is to implement a class that extends the interface of SegmentFetcher and provides config to Pinot Controller and Server as follows:

```
pinot.controller.segment.fetcher.`<protocol>`.class =`<class path to your␣
↪implementation>
```

or

```
pinot.server.segment.fetcher.`<protocol>`.class =`<class path to your implementation>
```

You can also provide other configs to your fetcher under config-root `pinot.server.segment.fetcher.<protocol>`

5.3 Pluggable Storage

Pinot enables its users to write a PinotFS abstraction layer to store data in a source of truth data layer of their choice for offline segments. We do not yet have support for realtime consumption in deep storage.

Some examples of storage backends(other than local storage) currently supported are:

- HadoopFS
- Azure Data Lake

If the above two filesystems do not meet your needs, please feel free to get in touch with us, and we can help you out.

5.3.1 New Storage Type implementation

In order to add a new type of storage backend (say, Amazon s3) implement the following class:

1. S3FS extends PinotFS

The properties for the stream implementation are to be set in your controller and server configurations, like so.