FLIP: Flink/Pinot Connector

Background and Motivation

A brief introduction to Apache Pinot and the ecosystem

Apache Pinot is a real-time distributed OLAP datastore, built to deliver scalable real time analytics with low latency. As Figure 1 shows, Pinot has an inbuilt lambda architecture as the following components:

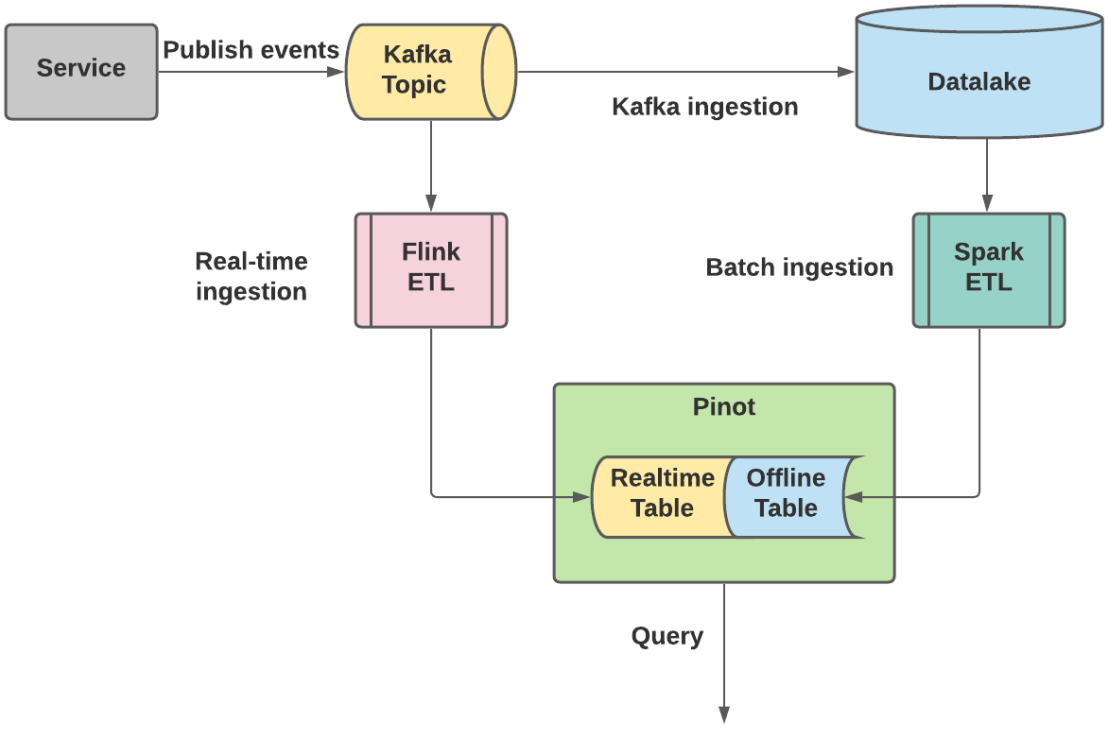

Figure 1. A typical Pinot ecosystem architecture

- Services/Applications produce the source events to the message queue (e.g. Kafka). The message queue typically buffers the messages for some retention period (e.g. a few days).
- The messages are ingested into the Data Lake (e.g. Hadoop) for historical data persistence.
- A streaming pipeline (e.g. Flink) transforms and processes the messages into another Kafka topic, which is used for ingestion by Pinot into a real-time table.

- A batch pipeline (e.g. Spark) does a similar transformation on the corresponding Hive dataset, creates Pinot segments, and pushes them to Pinot offline table.
- Both the real-time table and offline table are used for serving user queries, and Pinot creates a federated view from the real-time and offline results.

The problem

A common problem for this architecture is the duplication of the transformation logic in the streaming and batch ingestion pipelines. Because the batch data source (e.g. Hive datasets) are ingested from the Kafka topic, they typically share the same schema. Similarly, the real-time table and offline table in Pinot are different parts of the same table, and therefore share the same schema. That means users need to describe the same logic twice in Flink and Spark ETL jobs, making the jobs hard to maintain and sync. Ideally, we want to consolidate the streaming/batch ingestion logic, and use Flink for both pipelines.

Proposal

We propose a Flink Sink to Pinot on top of the TableSink interfaces (FLIP-95) for storing **BATCH** processing results in Pinot and also integrate the sink with the Unified Sink API (FLIP-143). The streaming sink is less useful to Pinot, because Pinot does not provide a record write API. Instead, Pinot ingests from the streams and buffers on the server, so that it can directly serve the latest events.

Design

The record writes in the Flink work in the following way, as shown in Figure 2:

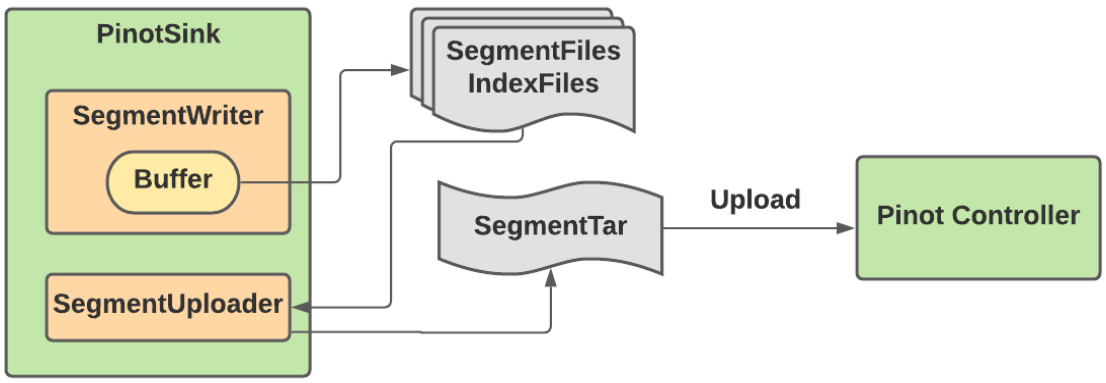

Figure 2. Pinot Sink workflow

1. PinotSink contains (1) a SegmentWriter, which can receive the records to write and generate the segment files, (2) a segmentUploader for uploading segments. The Sink converts the Flink data type to Pinot data type to feed the writer.

2. The sink initiates a segment write request to the PinotController, which will
    a. Initiate a transaction if it has not started, and send the Flink job id to indicate the transaction id.
    b. Send the segment write configs like “overwrite” or “append” to the time range of the segments.
3. The SegmentWriter has an internal buffer with a configured threshold for flushing
4. Upon flushing, SegmentWriter creates a segment file including the metadata files and index files.
5. The SegmentUploader then packages the files into a tar
6. The SegmentUploader uploads the segment tar to the Pinot controller
7. After all segments are uploaded, the sink sends a signal to Pinot controller to commit the transaction.

In order to support the at-least-once semantics, the segments encode the identifier in the segment name and therefore can be replaced if the job reruns. Also, because we aim to support the batch mode only in this proposal, we don’t plan to support the intermediate checkpoints.

Note that there is an ongoing effort on the SegmentWriter in the Pinot community, which implements step #2 - #5. To avoid duplicated efforts and simplify the connector implementation, the Pinot connector in this proposal will depend on the SegmentWriter, which is expected to release in 0.8.0. In particular, the SegmentWriter has the following methods:

```
interface SegmentWriter {
  init(Configuration configs);
  write(PinotRow row);
  write(PinotRow[] rows);
  flush();
  close();
}
```

And the Pinot sink will be implemented like the following:

```
public class PinotSinkFunction<T> extends RichSinkFunction<T> {
  public void open(Configuration parameters) throws Exception {
     SegmentWriter writer = new ...
  }

  public void invoke(T value, Context context) throws Exception {
     GenericRow row = pinotRowConverter.concert(value);
     writer.write(row) ;
     if(checkThreshod()) {
       writer.flush();
     }
  }

  public void close() throws Exception {
     writer.flush();
     writer.close();
  }
```

```
}
```

Connector Options

Option	Required	Default	Type	Description
connector	Y	none	string	The connector to use, here shall be ‘pinot’
table-name	Y	none	string	name of the pinot table
url	Y	none	string	URL of the Pinot controller
sink.buffer-flush.max-size	N	5mb	string	maximum size in memory of buffered rows for creating a segment.
sink.buffer-flush.max-rows	N	1000	int	maximum number of rows to buffer for each segment creation
sink.parallelism	N	none	int	Defines the parallelism of the Pinot sink operator. By default, the parallelism is determined by the framework using the same parallelism of the upstream chained operator.
segment-name.type	N	simple	string	the type of name generator to use. Following values are supported - ● simple - this is the default spec. ● normalizedDate - use this type when the time column in your data is in the String format instead of epoch time.
segment.name.postfix	N	none	string	For simple SegmentNameGenerator. Postfix will be appended to all the segment names.
segment.name.prefix	N	none	string	For normalizedDate SegmentNameGenerator. The Prefix will be prepended to all the segment names.

Schema and Data Type Mapping

The Sink connector can fetch the schema as well as table configurations via the Pinot controller API. The schema and table configs are used for index creation.

The data type needs to be converted during the write operation.

Note by default, Pinot transforms null values coming from the data source to a default value determined by the type of the corresponding column (or as specified in the schema), per the Pinot guide.

Flink SQL type	**Pinot type**	**Default value for null**
TINYINT	Integer	0
SMALLINT	Integer	0
INT	Integer	0
BIGINT	Long	0
DECIMAL	Not supported	Not supported
FLOAT	Float	0.0
BOOLEAN	Integer	0
DATE	Stores the number of days since epoch as an Integer value	0
TIME	Stores the milliseconds since epoch as Long value.	0
Timestamp	Stores the milliseconds since epoch as Long value.	0
STRING	String	“null”
BYTES	Bytes	byte[0]
ARRAY	Array	default value of array type

Rejected Alternatives

An alternative to the Pinot sink could be to use Hive as a data source and Kafka batch as a sink, and then have Pinot ingests from the Kafka topic. However, this does not work for the following reasons:

- The segments are managed differently in Pinot's realtime table and offline table. The realtime segments are grouped using Kafka offsets, whereas the offline segments are split based on the time range. As a result, The realtime segments cannot be replaced if the job reruns.
- Also, it's less efficient to use Kafka the intermediate storage for the batch processing, comparing to the direct segment creation and uploads.