Multi-Stream Consumption in Pinot

Sajjad Moradi

1. Introduction

Currently one realtime Pinot table only consumes from one stream. For use cases in which multiple existing streams are the input source for one Pinot table, the workaround is to use stream processors to read from different streams, perform necessary transformations, and write the output of all transformed events into another stream. Then one realtime table can be set up to consume from that output stream (Figure 1). This requires maintaining two systems: the stream processor jobs and also pinot realtime tables. Also an extra stream needs to be created for storing the output of stream processor jobs.

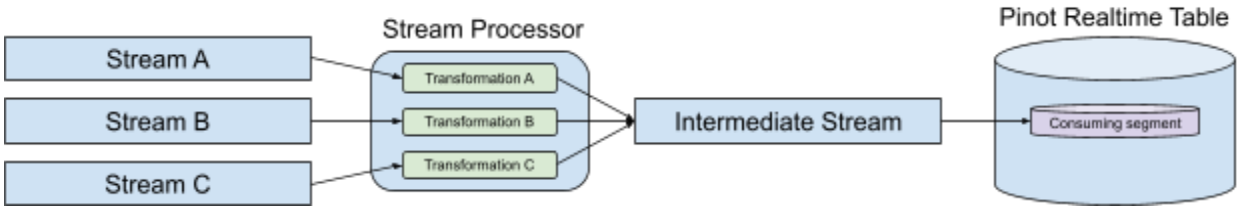


Figure 1. Existing solution for consuming data from multiple streams in a Pinot realtime table.

With the help of ingestion transformation and filtering, which has been added to Pinot realtime recently, we can support consuming from multiple streams natively in Pinot and eliminate the need of maintaining separate stream processor jobs. It also eliminates the creation of an extra stream to store the transformed events. This document describes different parts needed to natively support consuming from multiple streams.

It's worth mentioning here at the beginning of the document that the design proposed in this document doesn't handle the following complex transformations. The workaround mentioned above still needs to be used for these scenarios:

- *Join* - when two or more topics need to be joined on a shared field.
- *Re-partitioning* - when the partitioning scheme - partition column, number of partitions, and partitioning function - of the table is different than that of the input topics[1].

[1] If all input topics use the same partitioning scheme, then the proposed solution can be used. The segment assignment strategy may need some adjustment to utilize the benefits of partition aware routing.

2. How to Consume from Multiple Streams

We definitely need to spin off different consumer threads for different streams. The challenge, however, is how to store the ingested events. One approach is to use shared segments for different streams. There are a lot of complications with this approach like concurrent updates on a mutable segment from different streams' consuming threads or how we make sure the replicas have the same events as mutable segments read from multiple streams. This will require fundamental changes to the commit protocol for realtime segments. A simpler and much cleaner approach is to use separate mutable segments for different streams.

2.1 Separate Mutable Segments

The proposed solution here is simply presented in Figure 2. For each stream, there will be a separate consuming segment which has its own specific transformation/filtering.

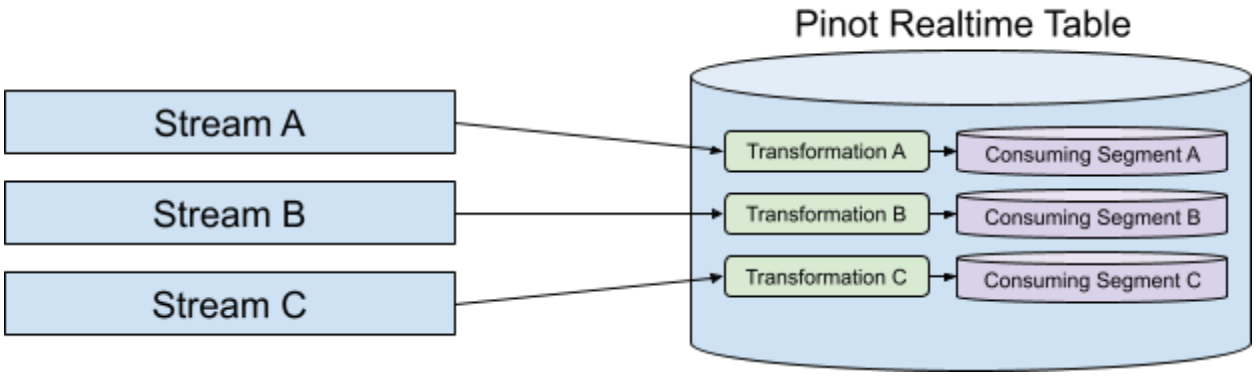


Figure 2. Proposed solution for multi-stream consumption.

To clarify with an example, let's say `tableT` requires consumption from three streams, `topicA`, `topicB` and `topicC` which only have one partition each. Currently the realtime segment name is in the format of `tableName_paritionId_seqId_creationTime`. So the segment name for `tableT` is `tableT__0__0__20210916T0700Z`. With the proposed solution, the segment name includes the stream name as well. So for our example, there will be three segments with the following names:

- `tableT_@_topicA__0__0__20210916T0700Z`,
- `tableT_@_topicB__0__0__20210916T0700Z`, and
- `tableT_@_topicC__0__0__20210916T0700Z`

where `_@_` is the delimiter that separates the table name from the topic name. Using this approach, each consumer thread writes into its own mutable segment. Also there's no need to change anything about the segment completion protocol because nothing has basically changed for segment completion of different streams.

The following sections define changes required for the proposed design of multi-topic consumption. For a detailed example on each part, you can refer to Appendix A which shows an end-to-end example of a working POC[2] that is done for this proposal.

2.2 Stream Configs Definition

Since the table needs to consume events from multiple streams, there will be a list of `ingestionConfig>streamIngestionConfig` defined in table config. Each stream config specifies its own topic name, flush parameters, etc.

2.3 Stream Record Transformation

Since events are consumed from multiple streams and stored in one table, there has to be some record transformations involved. These transformations could be as simple as changing a field name of the incoming stream record to the destination column name of the table. Each stream can have a set of transformations that only apply to the events coming from that stream, not other streams.

Currently with a single stream consumption, there's a list of transformations that apply to different fields of the incoming events. The current format of each transformConfig that's defined in `ingestionConfig>transformConfigs` is as follows:

```
class TransformConfig {
 String _columnName;
 String _transformFunction;
}
```

A simple approach to extend transformations to multiple streams is to add the stream name to each transformConfig. Since each consuming segment is going to consume from only one stream even in case of a multi-topic consuming table, we need to make sure that only transformations with the same stream name get applied to the incoming stream events.

This approach is backward compatible. For existing single-topic consuming tables, the stream name will be null which is fine because it means all transformations need to be applied to all incoming events and there's no need to distinguish between different transformations.

2.4 Stream Record Filtering

Filtering config is a little bit different than transformation config. Currently there's only one filtering configuration defined in table config in `ingestionConfig>filterConfig`. This makes sense for single topic consuming tables because there should be only one filtering function defined to accept or reject an incoming event. Currently FilterConfig has the following structure:

```
class FilterConfig {
```

[2] The POC is written as an integration test. The link to the code will be provided later if needed.

```
  String _filterFunction;
}
```

To be able to support filtering for multiple streams, we need to have multiple filtering configs. Also, each FilteringConfig needs to contain the name of the stream to which the filtering function is going to be applied. These changes are backward incompatible. Workaround for this is to introduce a new field in ingestionConfig with the name `filterConfigs` which is a list of FilterConfig objects. Then we can deprecate the existing filterConfig field.

2.5 Segment assignment

Current segment assignment for Realtime table uniformly sprays the partitions and replicas across the instances. If `N` is the number of instances and `R` is the number of replicas, for each partition `pId` and replica `rId`, the assigned instance is calculated by the following formula:

```
instanceId = (pId * R + rId) % N
```
[3]

Two points are worth mentioning here. First, using partitionId to assign segments makes sure that consuming segments for the same partition will always get assigned to the same machine, which is a good thing to have[4]. Second, the assignment formula is based on the assumption that all partitions of the stream are similar so spraying them uniformly across instances puts a similar load on different instances.

The assignment algorithm described above cannot be used for multi-stream consumption. First of all, partitionId's of different streams will collide. Also different streams have different characteristics like different data sizes, data structures, ingestion rates, etc. Spraying segments of different streams uniformly across the given instances will lead to unequal distribution of loads on those instances.

A new segment assignment - `MultiTopicRealtimeSegemntAssignment` - can be introduced to address the issues mentioned above by defining weights for different streams in table config. Basically there will be a map of stream names to their relative load weights. The map can be omitted, meaning that streams have similar load weight. For assigning a new segment to a proper instance (and of course assigning its replicas to proper instances as well), we iterate over all instances and for each instance, the weights of currently assigned segments are added up. The instance having the lowest sum of weights will be the target instance and the segment will be assigned to that instance.

The above algorithm is for the cases where the given segment is for a new partition. This happens when a new table is created. It can also happen for streams like Kinesis where new partitions can appear at any moment of time. On the other hand, for the cases where the given segment for assignment is for an existing partition, we can find the assigned instances for the existing partition and simply assign the new segment and its replica to the same instances.

---

[3] Similar situation exists for replica-group based assignments. Within a replica-group, partitions get sprayed uniformly across the instances: `instanceId = pId % N`

[4] This is a requirement for the Upsert use case. It's also beneficial for no-consumption alerts on consuming segments.

Using the weight based assignment will give us the opportunity to adjust the stream weights down the road based on the observed characteristics of the actual data from different streams. Ingestion rate for different streams can be inferred by looking at startTime and endTime and also startOffset and endOffset in segment ZK metadata. The memory footprint of the segments of different streams is available when consuming segments are converted to completed ones. These characteristics can be utilized to adjust the load weight associated with each stream. The adjustment can happen on-demand through rebalance requests or it can be set up to be applied periodically[5].

[5] The weight adjustment is a nice-to-have feature and it will not be included in the first roll out of the feature.

Appendix A - Example POC

- Avro schema of input Kafka topics

```
org.apache.avro.Schema _flightsSchema =
   SchemaBuilder.record(FLIGHTS).fields()
        .requiredInt(FLIGHT_NUM)
        .requiredString(SOURCE_CITY)
        .requiredString(DEST_CITY)
        .requiredLong(DEPARTURE_TIME)
        .requiredLong(ARRIVAL_TIME)
        .requiredString(AIRLINE)
        .requiredInt(CREATION_DATE)
        .endRecord();
```

```
org.apache.avro.Schema _trainScheduleSchema =
SchemaBuilder.record(TRAIN_SCHEDULES).fields()
       .requiredInt(TRAIN_NO)
       .requiredString(SRC)
       .requiredString(DST)
       .requiredLong(DEP_TIME)
       .requiredLong(ARR_TIME)
       .requiredString(OPERATOR)
       .requiredInt(CREATE_DATE)
       .endRecord();
```

- Pinot schema of the table

```
{
 "schemaName": "transportSchedule",
 "dimensionFieldSpecs": [
   {
     "name": "scheduleNo",
     "dataType": "INT"
   },
   {
     "name": "type",
     "dataType": "STRING"
   },
   {
     "name": "source",
     "dataType": "STRING"
   },
   {
     "name": "destination",
     "dataType": "STRING"
   },
   {
     "name": "departureTime",
     "dataType": "LONG"
   },
   {
     "name": "arrivalTime",
     "dataType": "LONG"
   },
   {
     "name": "operator",
     "dataType": "STRING"
   }
 ],
 "dateTimeFieldSpecs": [
   {
     "name": "createDate",
     "dataType": "INT",
     "format": "1:DAYS:EPOCH",
     "granularity": "1:DAYS"
   }
 ]
}
```

- Transformations

```
"transformConfigs": [
 {
   "columnName": "scheduleNo",
   "transformFunction": "Groovy({flightNum}, flightNum)",
   "streamName": "Flights"
 },
 {
   "columnName": "scheduleNo",
   "transformFunction": "Groovy({trainNo}, trainNo)",
   "streamName": "TrainSchedules"
 },
 {
   "columnName": "type",
   "transformFunction": "Groovy({\"flight\"}, flightNum)",
   "streamName": "Flights"
 },
 {
   "columnName": "type",
   "transformFunction": "Groovy({\"train\"}, trainNo)",
   "streamName": "TrainSchedules"
 },
 {
   "columnName": "source",
   "transformFunction": "Groovy({sourceCity}, sourceCity)",
   "streamName": "Flights"
 },
 {
   "columnName": "source",
   "transformFunction": "Groovy({src}, src)",
   "streamName": "TrainSchedules"
 },
 {
   "columnName": "destination",
   "transformFunction": "Groovy({destCity}, destCity)",
   "streamName": "Flights"
 },
 {
   "columnName": "destination",
   "transformFunction": "Groovy({dst}, dst)",
   "streamName": "TrainSchedules"
 },
 {
   "columnName": "departureTime",
   "transformFunction": "Groovy({depTime}, depTime)",
   "streamName": "TrainSchedules"
 },
 {
   "columnName": "arrivalTime",
   "transformFunction": "Groovy({arrTime}, arrTime)",
   "streamName": "TrainSchedules"
 },
 {
   "columnName": "operator",
   "transformFunction": "Groovy({airline}, airline)",
   "streamName": "Flights"
 },
 {
   "columnName": "createDate",
   "transformFunction": "Groovy({creationDate}, creationDate)",
   "streamName": "Flights"
 }
]
```

- Stream configs


```
"streamIngestionConfig": {
 "streamConfigMaps": [
   {
     "streamType": "kafka",
     "stream.kafka.consumer.type": "LOWLEVEL",
     "stream.kafka.broker.list": "localhost:19092",
     "Stream.kafka.consumer.factory.class.name":
       "org.apache.pinot.plugin.stream.kafka20.KafkaConsumerFactory",
     "stream.kafka.topic.name": "Flights",
     "Stream.kafka.decoder.class.name":
       "org.apache.pinot.integration.tests.MultiStreamConsumptionIntegrationTest$FlightsMessageDecoder",
     "realtime.segment.flush.threshold.rows": "6",
     "stream.kafka.consumer.prop.auto.offset.reset": "smallest"
   },
   {
     "streamType": "kafka",
     "stream.kafka.consumer.type": "LOWLEVEL",
     "stream.kafka.broker.list": "localhost:19093",
     "Stream.kafka.consumer.factory.class.name":
      "org.apache.pinot.plugin.stream.kafka20.KafkaConsumerFactory",
     "stream.kafka.topic.name": "TrainSchedules",
     "Stream.kafka.decoder.class.name":
      "org.apache.pinot.integration.tests.MultiStreamConsumptionIntegrationTest$TrainSchedulesMessageDecoder",
     "realtime.segment.flush.threshold.rows": "6",
     "stream.kafka.consumer.prop.auto.offset.reset": "smallest"
   }
 ]
},
```

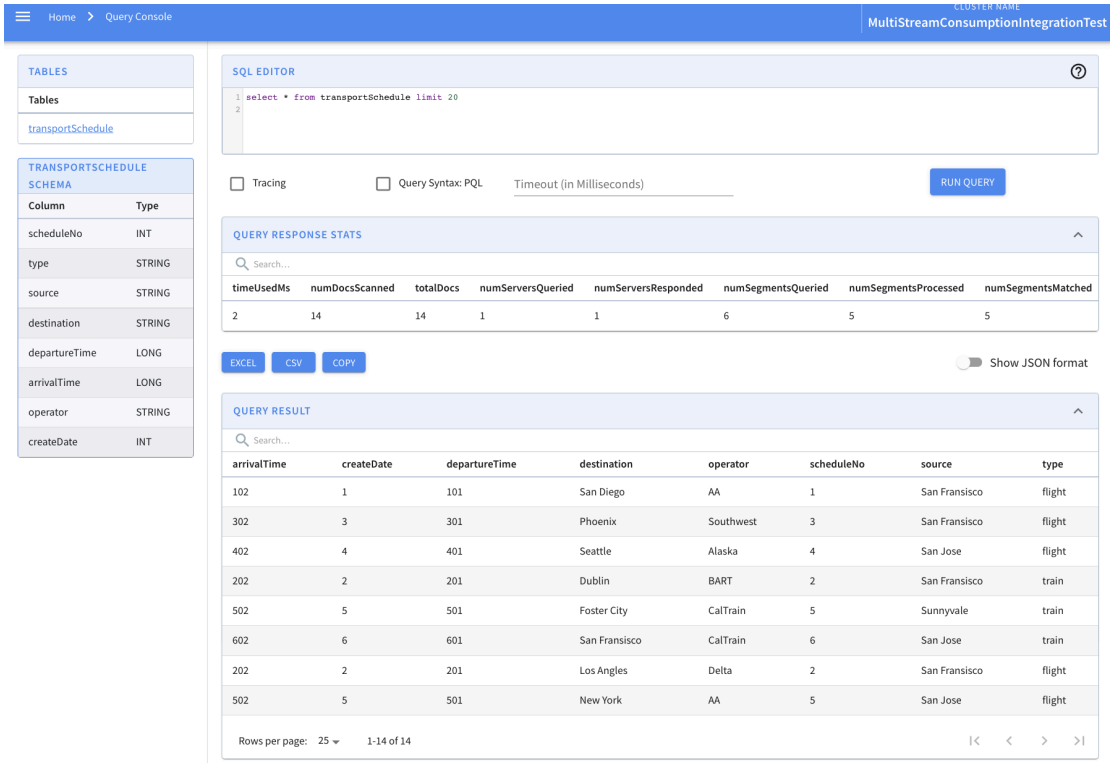

- External view - Each input topic has two partitions

- Data pushed into the input topics

```
// flights
File flightsAvroFile = new File(FLIGHTS + ".avro");
DataFileWriter<GenericRecord> flightsAvroWriter = new DataFileWriter<>(new GenericDatumWriter<>(_flightsSchema));
flightsAvroWriter.create(_flightsSchema, flightsAvroFile);
addData(FLIGHTS, flightsAvroWriter, 1, "San Fransisco", "San Diego",  101L, 102L, 1, "AA");
addData(FLIGHTS, flightsAvroWriter, 2, "San Fransisco", "Los Angles", 201L, 202L, 2, "Delta");
addData(FLIGHTS, flightsAvroWriter, 3, "San Fransisco", "Phoenix",    301L, 302L, 3, "Southwest");
addData(FLIGHTS, flightsAvroWriter, 4, "San Jose",      "Seattle",    401L, 402L, 4, "Alaska");
addData(FLIGHTS, flightsAvroWriter, 5, "San Jose",      "New York",   501L, 502L, 5, "AA");
addData(FLIGHTS, flightsAvroWriter, 6, "San Jose",      "Dallas",     601L, 602L, 6, "Southwest");
addData(FLIGHTS, flightsAvroWriter, 7, "Oakland",       "San Diego",  701L, 702L, 7, "AA");
addData(FLIGHTS, flightsAvroWriter, 8, "Oakland",       "Los Angles", 801L, 802L, 8, "AA");
flightsAvroWriter.close();

// trains
File trainAvroFile = new File(TRAIN_SCHEDULES + ".avro");
DataFileWriter<GenericRecord> trainAvroWriter =
    new DataFileWriter<>(new GenericDatumWriter<>(_trainScheduleSchema));
trainAvroWriter.create(_trainScheduleSchema, trainAvroFile);
addData(TRAIN_SCHEDULES, trainAvroWriter, 1, "San Fransisco", "Fremont",       101L, 102L, 1, "BART");
addData(TRAIN_SCHEDULES, trainAvroWriter, 2, "San Fransisco", "Dublin",        201L, 202L, 2, "BART");
addData(TRAIN_SCHEDULES, trainAvroWriter, 3, "San Fransisco", "Mountain View", 301L, 302L, 3, "BART");
addData(TRAIN_SCHEDULES, trainAvroWriter, 4, "San Jose",      "Oakland",       401L, 402L, 4, "BART");
addData(TRAIN_SCHEDULES, trainAvroWriter, 5, "Sunnyvale",     "Foster City",   501L, 502L, 5, "CalTrain");
addData(TRAIN_SCHEDULES, trainAvroWriter, 6, "San Jose",      "San Fransisco", 601L, 602L, 6, "CalTrain");
addData(TRAIN_SCHEDULES, trainAvroWriter, 7, "San Jose",      "Livermore",     701L, 702L, 7, "BART");
addData(TRAIN_SCHEDULES, trainAvroWriter, 8, "Dublin",        "Oakland",       801L, 802L, 8, "BART");
trainAvroWriter.close();
```

- Sample query result



Appendix B - Q&A on potential operational issues

- What happens if one stream stops emitting events and the other streams continue with no problem?

  We don't join topics here. We just consume from each topic independently. Let's look at how we handle this issue currently with single-topic consumption. If there's some problem with the stream topic and realtime server can't consume from it, we get alerted that there's no consumption on that topic. During the period when the problem shows up till the time it gets fixed and consumption resumes, the corresponding data is missing. That means queries that have data for the problematic period will have incorrect results and that gets communicated with the table owners.

  Same thing happens for multi-topic consumption. If there's an issue with one or some of the topics and consumption stops for them, we get alerted for those topics. Until the problem is fixed, the queries will have incorrect results, like single-topic consumption. The topics with no problem continue consumption. Once data is available on the problematic topics, the consumption resumes for these topics from the point that event consumption previously stopped. Basically since there's no join involved between these topics, they consume independent of each other.

- How does the new design address high QPS low latency use cases that rely on partitioning to route the queries?

  As described at the beginning of the document, the proposed solution doesn't work with use cases requiring re-partitioning. If there's no need for repartitioning and all the input topics use the same partitioning scheme - partition function, number of partitions, and partition column - then one can use partition based segment assignment which assigns segments of the same partitions - no matter to which topic they belong - to the same servers. This way, a query with filter on partition column will be directed to only one server which makes it performant for high QPS low latency use cases.