

New Kylin Streaming in eBay

[Why new Kylin Streaming](#)

[Architecture](#)

[Components](#)

[How Streaming Cube Engine Works](#)

[How Streaming Query Engine Works](#)

[Detail Design](#)

[Real-time Segment State](#)

[Real-time Segment Store](#)

[Directory Structure on Disk](#)

[Columnar Format](#)

[Compression](#)

[High Availability](#)

[Failure Recovery](#)

[Performance](#)

Why new Kylin Streaming

1. Milliseconds Data Preparation Delay

Kylin provide sub-second query latency for extremely large dataset, the underly magic is precalculation cube. But the cube building often take long time(usually hours for large data sets), in some case, the analyst needs real-time data to do analysis, so we want to provide a real-time OLAP, when data is produced to system, it can be queried.

2. Support Lambda Architecture

Real-time data often not reliable, that may caused by many reasons, for example, the upstream processing system has a bug, or the data need to be changed after some time, etc. So we need to support lambda architecture, which means the cube can be built from the streaming source(like Kafka), and the historical cube data can be refreshed from batch source(like Hive).

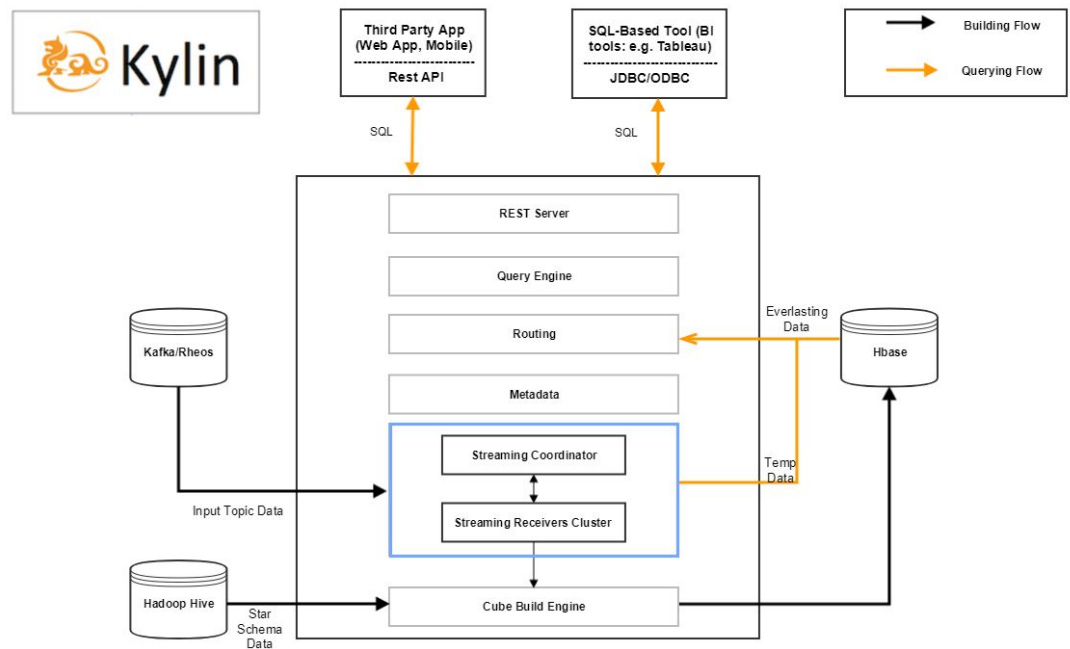
3. Less MR jobs and HBase Tables

From Kylin 1.6, community has provided a streaming solution, it uses MR to consume Kafka data and then do batch cube building, it can provide minute-level data preparation latency, but to ensure the data latency, you need to schedule the MR very shortly(5 minutes or even less), that will cause too many hadoop jobs and small hbase tables in the system, and dramatically increase the Hadoop system's load.

4. High Performance and HA

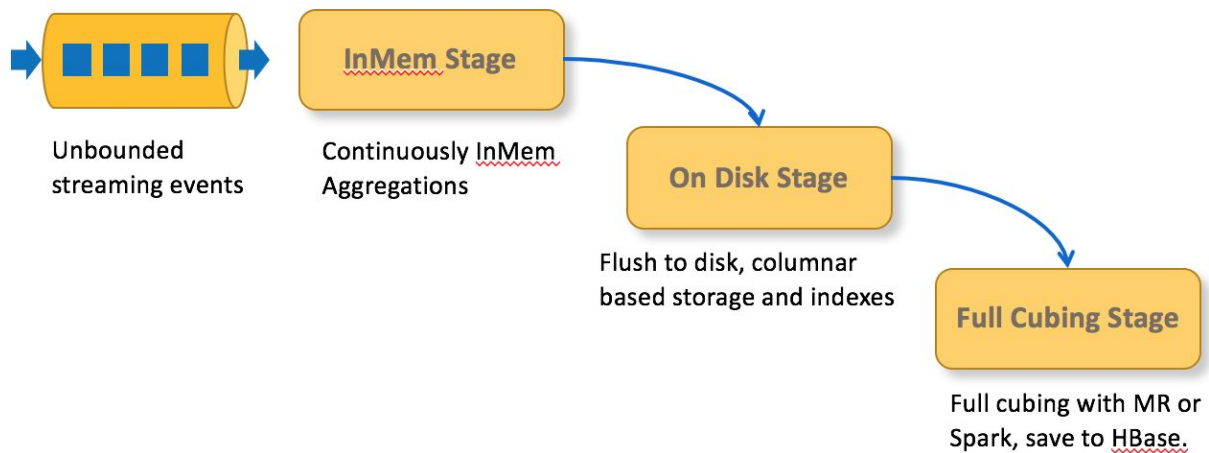
We need high query performance when there are very large volume data come from the streaming source, and the streaming node should be high available and easy to extend.

Architecture

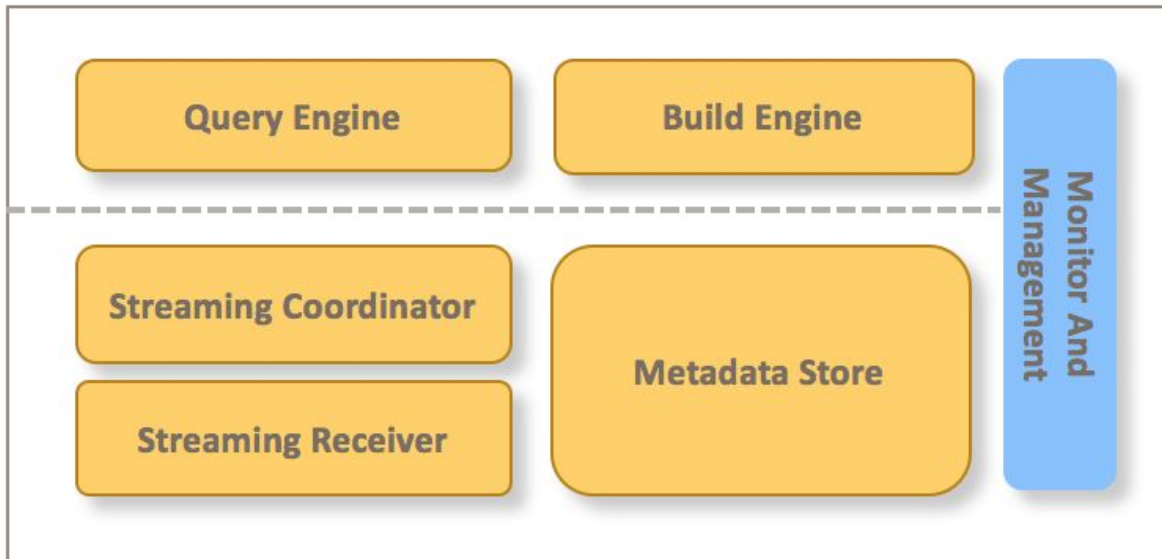


The blue rectangle is what we added in current Kylin's architecture, this is a streaming cluster which is responsible to ingest data from streaming source, and provide query for real-time data.

We divide the unbounded incoming streaming data into 3 stages, the data come into different stages are all queryable.



Components



Streaming Receiver: Responsible to ingest data from stream data source, and provide real-time data query.

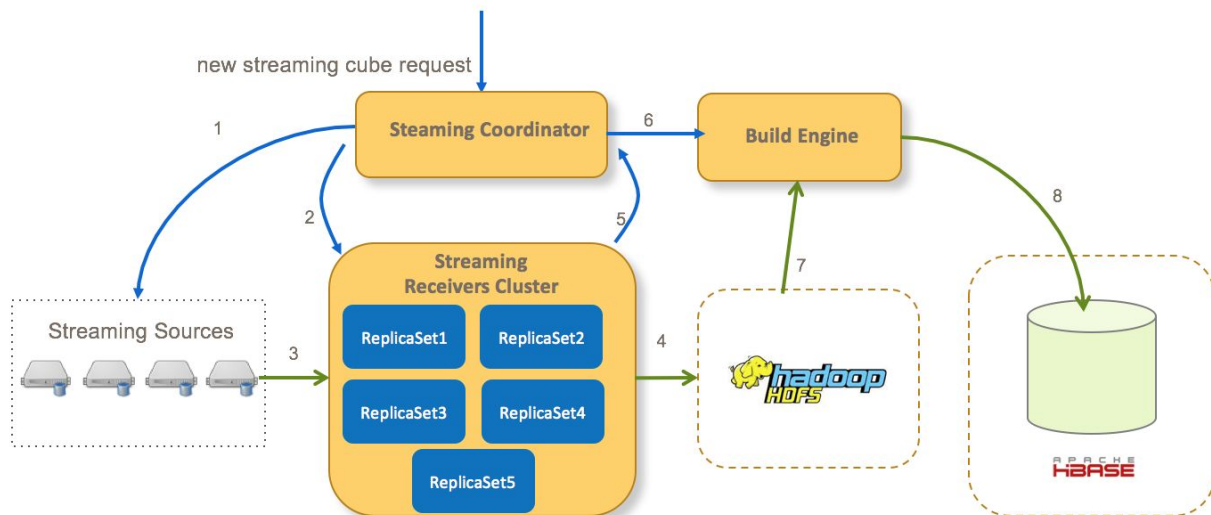
Streaming Coordinator: Responsible to do coordination works, for example, when new streaming cube is onboard, the coordinator need to decide which streaming receivers can be assigned.

Metadata Store: Used to store streaming related metadata, for example, the cube assignments information, cube build state information.

Query Engine: Extend the existing query engine, support to query real-time data from streaming receiver

Build Engine: Extend the existing build engine, support to build full cube from the real-time data

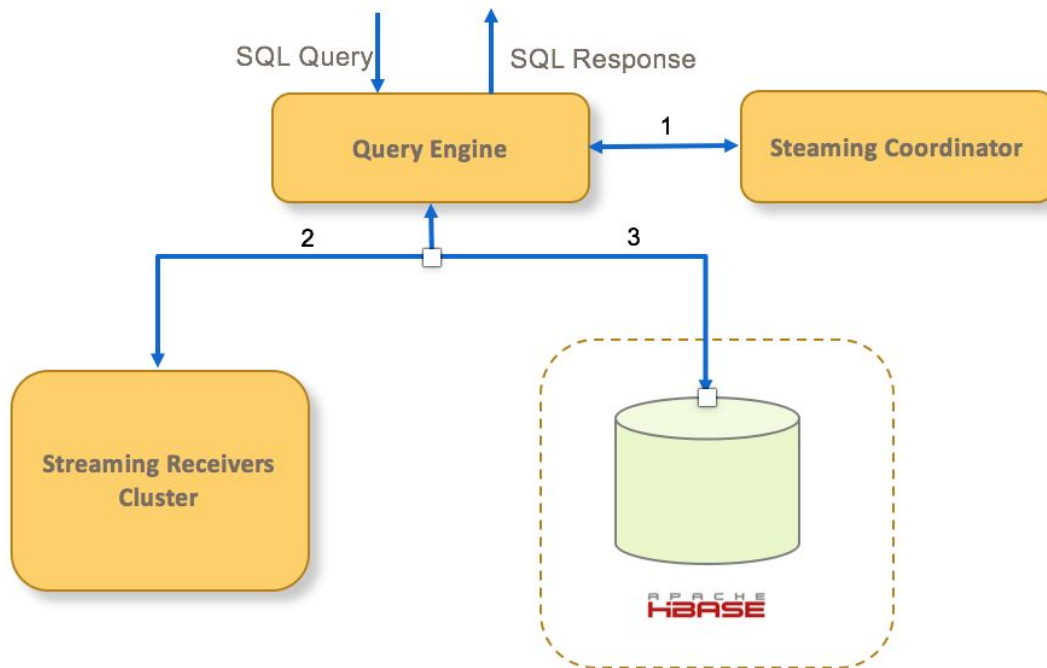
How Streaming Cube Engine Works



When new streaming cube request comes, the following steps will happen:

1. Coordinator ask streaming source for all partitions of the cube
2. Coordinator decide which streaming receivers to assign to consume streaming data, and ask streaming receivers to start consuming data.
3. Streaming receiver start to consume and index streaming events
4. After sometime, streaming receiver copy the immutable segments from local files to remote HDFS files
5. Streaming receiver notify the coordinator that a segment has been persisted to HDFS
6. Coordinator submit a cube build job to Build Engine to trigger cube full building after all receivers have submitted their segments
7. Build Engine build all cuboids from the streaming HDFS files
8. Build Engine store cuboid data to Hbase, and then the coordinator will ask the streaming receivers to remove the related local real-time data.

How Streaming Query Engine Works



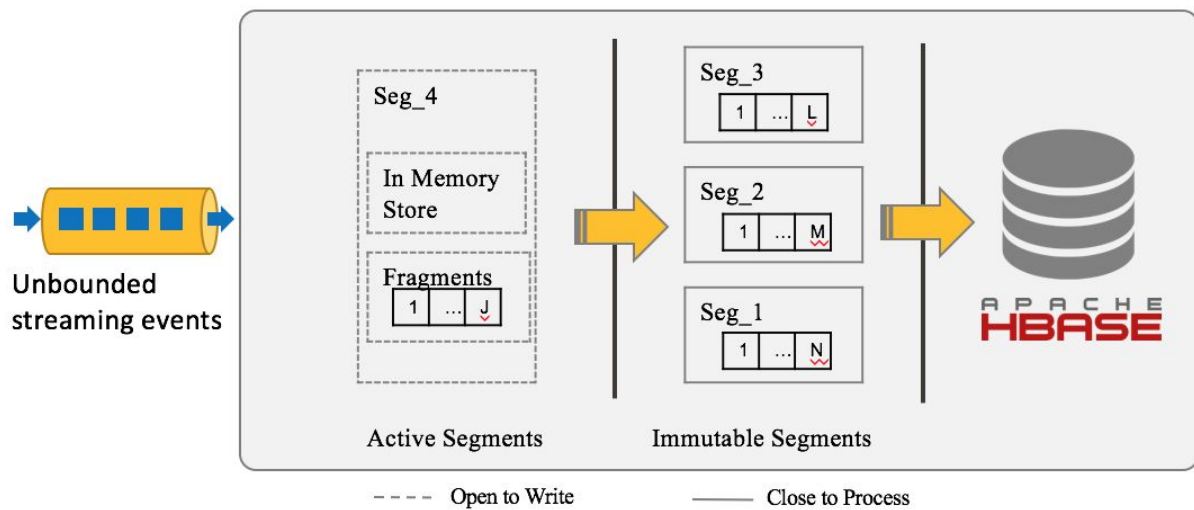
1. Query Engine ask Streaming Coordinator what streaming receivers are assigned for the cube
2. Query Engine send query request to related streaming receivers to query realtime segments
3. Query Engine send query request to Hbase to query historical segments
4. Query Engine aggregate the query results, and send response back to client

Detail Design

Real-time Segment State

Real-time segments are divided by event time, when new event comes, it will be calculated which segment it will be located, if the segment doesn't exist, create a new one.

The new created segment is in 'Active' state first, if no further events coming into the segment after some preconfigured period, the segment state will be changed to 'Immutable', and then write to remote HDFS.



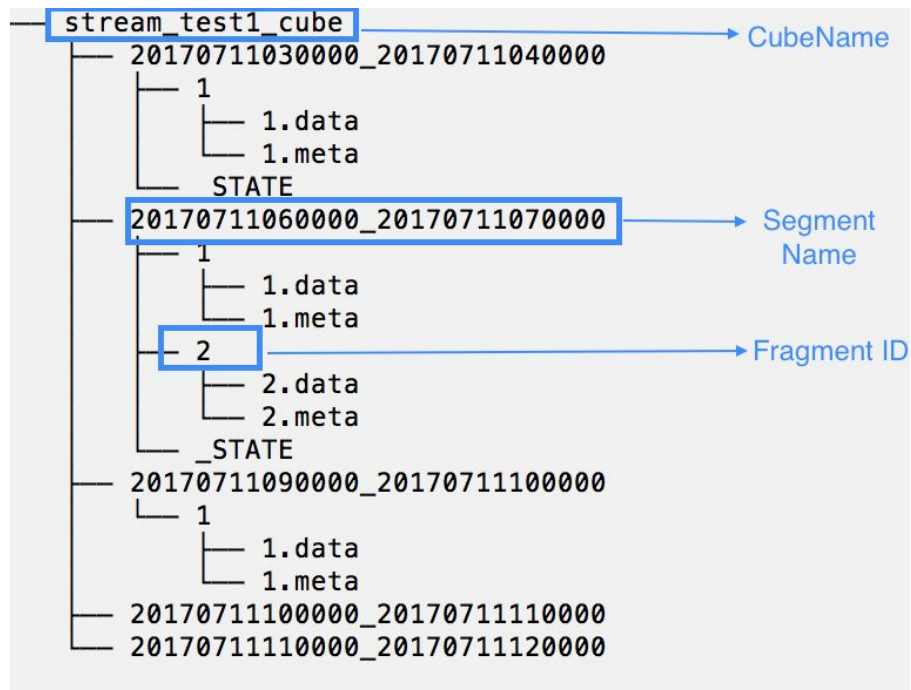
Real-time Segment Store

Each real-time segment has a memory store, new event will first goes into the memory store to do aggregation, when the memory store size reaches the configured threshold, it will be then be flushed to local disk as a fragment file.

Not all cuboids are built in the receiver side, only basic cuboid and some specified cuboids are built.

The data is stored as columnar format on disk, and when there are too many fragments on disk, the fragment files will be merged by a background thread automatically.

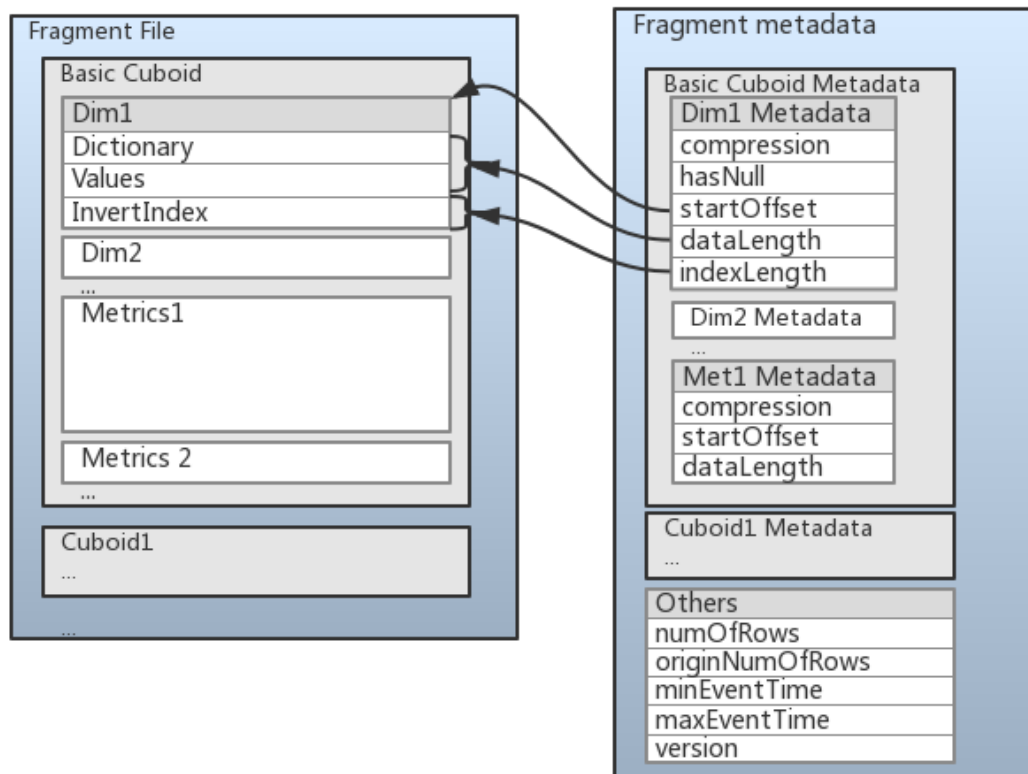
Directory Structure on Disk



The fragment id is increased automatically.

Columnar Format

To improve the query performance, the data is stored in columnar format, the data format is like:



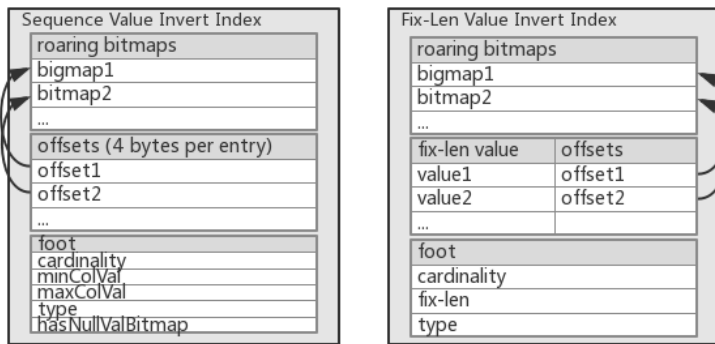
Each cuboid data is stored together, and in each cuboid the data is stored column by column, the metadata is stored in json format.

Each dimension data is divided into 3 parts:

The first part is Dictionary part, the part exists when the dimension encoding is set to 'Dict' in cube design, by default we use tri-tree dictionary, just like how dictionary used in Kylin: <https://kylin.apache.org/blog/2015/08/13/kylin-dictionary/> to minimize the memory footprints and preserve the original order.

The second part is dictionary encoded values, additional compression mechanism can be applied to these values, since the values for the same column are usually similar, so the compression rate will be very good.

The third part is invert-index data, use Roaring Bitmap to store the invert-index info, the following picture shows how invert-index data is stored, there are two types of format, the first one is dictionary encoding dimension's index data format, the second is other encoding dimension's index data format.



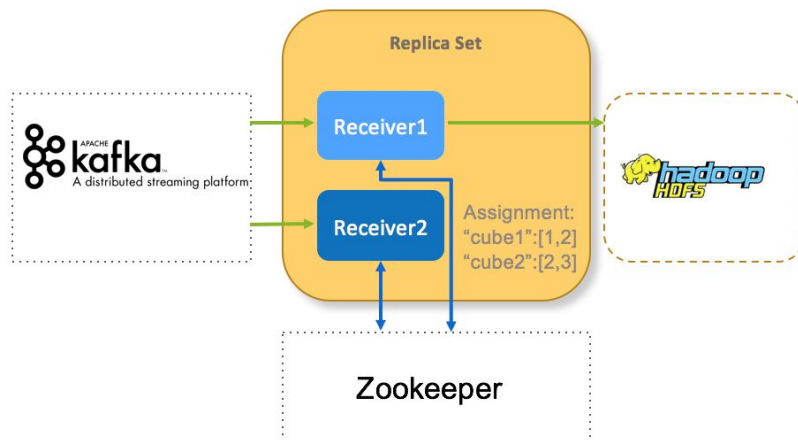
Compression

- Support Run Length Encoding and LZ4 Compression
- Use RLE compression for time-related dim and first dim
- Use LZ4 for other dimensions by default
- Use LZ4 Compression for simple-type measure(long, double)
- No compression for complex measure(count distinct, topn, etc.)

High Availability

Streaming receivers are group into replica-sets, all receivers in the same replica-set share the same assignments, so that when one receiver is down, the query and event consuming will not be impacted.

In each replica-set, there is a lead responsible to upload real-time segments to HDFS, and zookeeper is used to do leader election

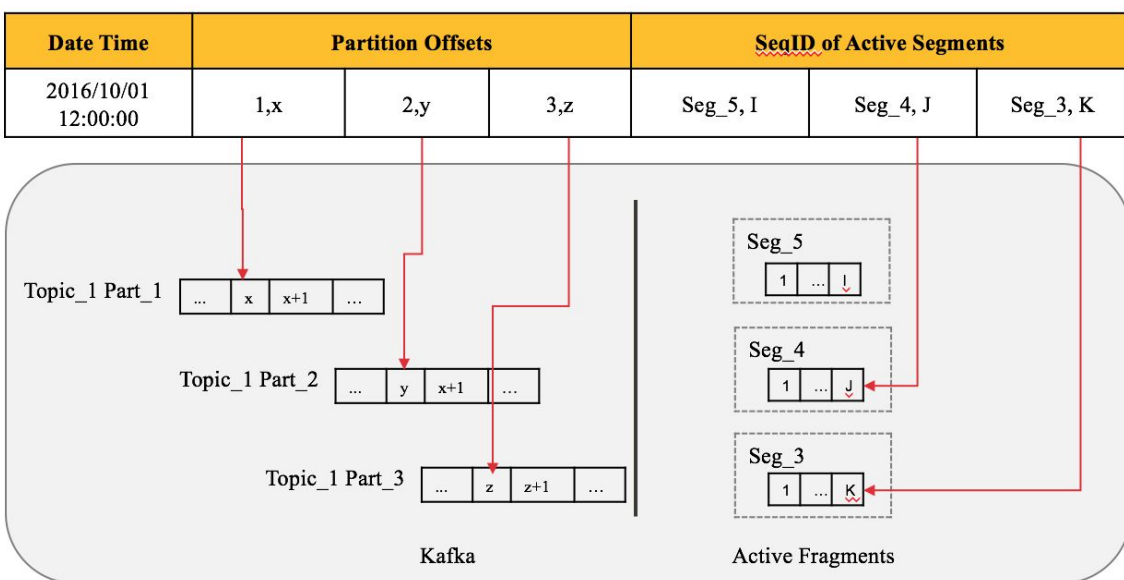


Failure Recovery

We do checkpoint periodically in receiver side, so that when the receiver is restarted, the data can be restored correctly.

There are two parts in the checkpoint: the first part is the streaming source consume info, for Kafka it is {partition:offset} pairs, the second part is disk states {segment:fragmentID} pairs, which means when do the checkpoint what's the max fragmentID for each segment.

When receiver is restarted, it will check the latest checkpoint, set the Kafka consumer to start to consume data from specified partition offsets, and remove the fragment files that the fragmentID is larger than the checkpointed fragmentID on the disk.



Besides the local checkpoint, we also have remote checkpoint, to restore the state when the disk is crashed, the remote checkpoint is saved to Cube Segment metadata after HBase segment build, like:

```
"segments": [{ ...,  
  "stream_source_checkpoint": {"0": 8946898241, "1": 8193859535, ...}  
},  
]
```

The checkpoint info is the smallest partition offsets on the streaming receiver when real-time segment is sent to full build.

Performance

Detail performance data for real-time query and ingest can be viewed in the doc:

<https://drive.google.com/file/d/1GSBMpRuVQRmr8Ev2BWvssfMd-Rck9vsH/view?ths=true>