Stream Ingest with Kafka-Connect

References:

PNDA-guide/streamingest

PNDA-guide/streamingest/topic-preparation

Confluent Kafka-Connect Deep Dive/Converters and Serialization

Overview

Kafka is the "front door" of PNDA, allowing the ingest of high-velocity data streams, distributing data to all interested consumers and decoupling data sources from data processing applications and platform clients.

Kafka Connect, an open source component of Kafka, is a framework for connecting Kafka with external systems such as databases, key-value stores, search indexes, and file systems. Using Kafka Connect you can use existing connector implementations for common data sources and sinks to move data into and out of Kafka.

We want to store the events that arrives to the kafka topics in HDFS datesets. Currently PNDA uses Apache Gobblin to ingest data streams from kafka to HDFS.

However Apache Gobblin has some disadvantages:

- Apache Incubator Project.
- Configuration complexity.
- Job distribution through MapReduce mode disadvantages (from gobblin documentation):
    - The MR mode suffers from problems such as large overhead mostly due to the overhead of submitting and launching a MR job and poor cluster resource usage.
    - The MR mode is also fundamentally not appropriate for real-time data ingestion given its batch nature.

NOTE: Recently Apache Gobblin added support of Apache Yarn as orchestrator for job distribution that may solves some of the previous disadvantages.

Here we explore kafka-connect as an alternative, a component of Kafka that would let us remove Yarn dependency and simplify configuration, management and scaling of the stream ingest jobs.

Getting Started

We will setup a local deployment of PNDA to test how kafka-connect works. Follow the Testing locally with microk8s' instructions, using the next yaml profile to enable kafka-connect related components:

```
# Enabling Kafka and Kafka-connect related components:
cp-zookeeper:
  enabled: true
cp-kafka:
  enabled: true
cp-schema-registry:
  enabled: true
schema-registry-ui:
  enabled: true
cp-kafka-connect:
  enabled: true
kafka-connect-ui:
  enabled: true
kafka-manager:
  enabled: true


# Enabling HDFS related components:
hdfs:
  enabled: true

# Disabling the rest of the external components
grafana:
  enabled: false
jupyterhub:
  enabled: false
cp-kafka-rest:
  enabled: false
cp-ksql-server:
  enabled: false
hbase:
  enabled: false
opentsdb:
  enabled: false
spark-standalone:
  enabled: false
```

Kafka-connect-ui can be accessed through the pnda-kafka-connect-ui service ip:

```
kubectl get service -o wide -n pnda pnda-kafka-connect-ui
```

Accessing kafka-connect-ui service ip through your browser:

Now we will configure some HDFS Sink Connectors to ingest events in kafka topics to HDFS.

Following instructions in PNDA-guide/topic-preparation we find different kind of topics:

- Matching PNDA schema topic
- Avro configured topic
- un-configured topic

Matching schema topic

When data being streamed to the platform already happends to adhere to the PNDA schema.

Let's divide the test in two parts, depending in how the data is encoded in the kafka topic:

AvroEncoded

Reference: https://docs.confluent.io/current/connect/kafka-connect-hdfs/index.html

We use the kafka-avro-console-producer tool to produce some avro encoded data in a topic. The tool is available in the cp-schema-registry pod so we open a bash terminal inside the cp-schema-registry container of the pod:

```
kubectl exec -ti svc/pnda-cp-schema-registry -- /bin/bash


JMX_PORT=5559 kafka-avro-console-producer --broker-list $SCHEMA_REGISTRY_KAFKASTORE_BOOTSTRAP_SERVERS --topic
pnda.avro.test --property value.schema='{"namespace": "pnda.entity","type": "record","name": "event","fields":
[ {"name": "timestamp", "type": "long"}, {"name": "source", "type": "string"}, {"name": "rawdata", "type":
"bytes"}]}'
```

Then copy the next lines in the topic. If you don't respect the pnda avro schema, the kafka-avro-console-producer will exit with error.

```
{"timestamp":1567516179000,"source":"test","rawdata":"200"}
{"timestamp":1567516180000,"source":"test","rawdata":"300"}
{"timestamp":1567516181000,"source":"test","rawdata":"150"}
{"timestamp":1567516182000,"source":"test","rawdata":"100"}
{"timestamp":1567516183000,"source":"test","rawdata":"230"}
{"timestamp":1567516184000,"source":"test","rawdata":"250"}
```

CTRL+C to exit pnda-avro-consol-producer and 'exit' to close container terminal.

Go back to the kafka-connect-ui and create a new HDFS Sink connector with the following config properties:

```
name=pnda_avro
connector.class=io.confluent.connect.hdfs.HdfsSinkConnector
topics=pnda.avro.test
hdfs.url=hdfs://pnda-hdfs-namenode:8020
tasks.max=3
flush.size=2
format.class=io.confluent.connect.hdfs.avro.AvroFormat
key.converter=org.apache.kafka.connect.storage.StringConverter
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://pnda-cp-schema-registry:8081
logs.dir=/kafkaconnectlogs
topics.dir=/user/PNDA/datasets
partitioner.class=io.confluent.connect.storage.partitioner.HourlyPartitioner
path.format='year'=YYYY/'month'=MM/'day'=dd/'hour'=HH
locale=ES
timezone=CET
```

A new avro file containting $flush.size events should be writen to HDFS. dataset folder is named as the kafka topic, and can be further partitioned with partitioner classes (more info at hdfs-connector-doc).

You can check if files are correctly writen through "Utilities" > "Browse the Filesystem" in the HDFS-namenode ui:

```
kubectl get service pnda-hdfs-namenode
```

Default port is (50070).

JsonEncoded

> *WARNING kafka-connect issue! Kafka-connect logs an error when writing HDFS in Avro format from events encoded in Json in kafka. With a flush_size to 1, It will correctly write a single avro file per event (does not make to much sense).*
>
> *The issue describing this problem is written here.*

Avro configured topic

For a source that ingest events in avro format with a schema different to pnda-schema, a HDFS sink connector must be created with a custom configuration with transformations to extract/rename timestamp, source and raw fields.

Reference: https://docs.confluent.io/current/connect/transforms/index.html.

Un-configured topic

For topics without following a schema, the following connector configuration will transform the event to a record following pnda-schema before storing in HDFS:

- source field will be set to the topic name.
- timestamp field will be the kafka ingestion time.
- rawdata will be the event.

We use the kafka-avro-producer tool to produce some schemaless data in a topic. The tool is available in the cp-kafka pod so we open a bash terminal inside the cp-kafka container of the pod:

```
kubectl exec -ti svc/pnda-cp-kafka -- /bin/bash


KAFKA_OPTS='' kafka-console-producer --topic unconfigured --broker-list=localhost:9092
```

Then copy the next lines in the topic.

```
This is event one.
Second event without a schema.
```

CTRL+C to exit pnda-console-producer and 'exit' to close container terminal.

Go back to the kafka-connect-ui and create a new HDFS Sink connector with the following config properties:

```
name=pnda-unconfigured
connector.class=io.confluent.connect.hdfs.HdfsSinkConnector
topics=unconfigured
hdfs.url=hdfs://pnda-hdfs-namenode:8020
tasks.max=3
flush.size=2
format.class=io.confluent.connect.hdfs.avro.AvroFormat
key.converter=org.apache.kafka.connect.storage.StringConverter
value.converter=org.apache.kafka.connect.storage.StringConverter
logs.dir=/kafkaconnectlogs
topics.dir=/user/PNDA/datasets
partitioner.class=io.confluent.connect.storage.partitioner.HourlyPartitioner
path.format='year'=YYYY/'month'=MM/'day'=dd/'hour'=HH
locale=ES
timezone=CET
transforms=MakeMap,AddSource
transforms.MakeMap.type=org.apache.kafka.connect.transforms.HoistField$Value
transforms.MakeMap.field=rawdata
transforms.AddSource.type=org.apache.kafka.connect.transforms.InsertField$Value
transforms.AddSource.topic.field=source
transforms.AddSource.timestamp.field=timestamp
```

A new avro file containting $flush.size events should be writen to HDFS. dataset folder is named as the kafka topic, and can be further partitioned with partitioner classes (more info at hdfs-connector-doc).