..

DataStream Connectors

Date published: 2020-05-19
Date modified:

https://docs.cloudera.com/

Legal Notice

Contents

HBase sink with Flink

Cloudera Streaming Analytics offers HBase connector as a sink. Like this you can store the output of a real-time processing application in HBase. You must develop your application defining HBase as sink and add HBase dependency to your project.

The HBase Streaming connector has the following key features:

- Automatic configuration on the CDP Private Cloud Base platform
- High throughput buffered operations
- Customizable data-driven update/delete logic

To use the HBase integration, add the following dependency to your project:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-hbase_${scala.binary.version}</artifactId>
  <version>${flink.version}</version>
</dependency>
```

The general purpose HBase sink connector is implemented in the org.apache.flink.addons.hbase.HBaseSinkFunction class.

This is an abstract class that must be extended to define the interaction logic (mutations) with HBase. By using the BufferedMutator instance, you can implement arbitrary data driven interactions with HBase. While it is possible to execute all mutations supported by the BufferedMutator interface, Cloudera strongly recommends that users should only use idempotent mutations: Put and Delete.

Creating and configuring the HBaseSinkFunction

You must configure the HBaseSinkFunction with Table names to have HBase as a sink. The HBase table needs to be created before the streaming job is submitted. You should also configure the operation buffering parameters to make sure that every data coming from Flink is buffered into HBase.

The HBase sink instance is always created as a subclass of the HBaseSinkFunction. When users create the subclass they have to provide required and optional parameters through the constructor of the superclass, the HBaseSinkFun ction itself.

Required parameters:

- Table name (the table itself must be created before the streaming job starts)

Optional parameters:

- Hadoop Configuration object for setting up the HBase client
- HBaseOptions for minimal connection configuration

The optional parameters are configured automatically by the Cloudera platform and should only be used for setting up custom HBase connections.

Important: The Flink Gateway node should also be an HBase Gateway node for the automatic configuration to work in the Cloudera environment.

To configure the operation buffering parameters, you need to use the HBaseSinkFunction.setWriteOptions() method. You can set the following configuration parameters using the HBaseWriteOptions object:

- setBufferFlushMaxSizeInBytes : Maximum byte size of the buffered operations before flushing
- setBufferFlushMaxRows : Maximum number of operations buffered before flushing
- setBufferFlushIntervalMillis : Maximum time interval before flushing

See the following example for setting up an HBase sink running on the Cloudera platform:

```
// Define a new HBase sink for writing to the ITEM_QUERIES table
HBaseSinkFunction<QueryResult> hbaseSink = new HBaseSinkFunction<QueryResul
t>("ITEM_QUERIES") {
@Override
public void executeMutations(QueryResult qresult, Context context, Buffer
edMutator mutator) throws Exception {
  // For each incoming query result we create a Put operation
  Put put = new Put(Bytes.toBytes(qresult.queryId));
  put.addColumn(Bytes.toBytes("itemId"), Bytes.toBytes("str"),   Bytes.toBy
tes(qresult.itemInfo.itemId));
  put.addColumn(Bytes.toBytes("quantity"), Bytes.toBytes("int"), Bytes.to
Bytes(qresult.itemInfo.quantity));
  mutator.mutate(put);
  }
};
// Configure our sink to not buffer operations for more than a second (to
reduce end-to-end latency)
hbaseSink.setWriteOptions(HBaseWriteOptions.builder()
  .setBufferFlushIntervalMillis(1000)
  .build()
);
// Add the sink to our query result streamqueryResultStream.addSink(hbaseSi
nk);
```

Kafka with Flink

Cloudera Streaming Analytics offers Kafka connector as a source and a sink to create a complete stream processing architecture with a stream messaging platform. You must develop your application defining Kafka as a source and sink, after adding Kafka dependency to your project.

About this task

In CSA, adding Kafka as a connector creates a scalable communication channel between your Flink application and the rest of your infrastructure. Kafka is often responsible for delivering the input records and for forwarding them as an output, creating a frame around Flink.

When Kafka is used as a connector, Cloudera offers the following integration solutions:

- Schema Registry
- Streams Messaging Manager
- Logging to Kafka
- Kafka Metrics Reporter

Both Kafka sources and sinks can be used with exactly once processing guarantees when checkpointing is enabled.

For more information about Apache Kafka, see the Cloudera Stream Processing documentation.

Procedure

1. Add the Kafka connector dependency to your Flink job.

   Example for Maven:

   ```
   <dependency>
     <groupId>org.apache.flink</groupId>
     <artifactId>flink-connector-kafka_${scala.binary.version}</artifactId>
     <version>${flink.version}</version>
   ```

```
</dependency>
```

2. Set FlinkKafkaConsumer as the source in the Flink application logic.

```
Properties properties = new Properties();
properties.put("bootstrap.servers", "<your_broker_url>");
properties.put("group.id", "<your_group_id>");

FlinkKafkaConsumer<String> source = new FlinkKafkaConsumer<>(
                "<your_input_topic>",
                 new SimpleStringSchema(),
                 properties);
```

3. Set FlinkKafkaProducer as the sink in the Flink application logic.

```
Properties properties = new Properties();
properties.put("bootstrap.servers", "<your_broker_url>");
FlinkKafkaProducer<String> output = new FlinkKafkaProducer<>(
                 <your_output_topic>",
                 new SimpleStringSchema(),
                 properties,
    Semantic.EXACTLY_ONCE);
```

Related Information

Stateful Tutorial: Setting up Kafka inputs and outputs

Checkpointing

Schema Registry with Flink

When Kafka is chosen as source and sink for your application, you can use Cloudera Schema Registry to register and retrieve schema information of the different Kafka topics. You must add Schema Registry dependency to your project and add the appropriate schema object to your Kafka topics.

There are several reasons why you should prefer the Schema Registry instead of custom serializer implementations on both consumer and producer side:

- Offers automatic and efficient serialization/deserialization for avro and basic types (+ JSON in the future)
- Guarantees that only compatible data can be written to a given topic (assuming that every producer uses the registry)
- Supports safe schema evolution on both producer and consumer side
- Offers visibility to developers on the data types and they can track schema evolution for the different Kafka topics

Add the following Maven dependency or equivalent to use the schema registry integration in your project:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-avro-cloudera-registry</artifactId>
  <version>${flink.version}</version>
</dependency>
```

The schema registry can be plugged directly into the FlinkKafkaConsumer and FlinkKafkaProducer using the appropriate schema:

- org.apache.flink.formats.avro.registry.cloudera.ClouderaRegistryKafkaDeserializationSchema
- org.apache.flink.formats.avro.registry.cloudera.ClouderaRegistryKafkaSerializationSchema

See the Apache Flink documentation for Kafka consumer and producer basics.

Supported types

Currently, the following data types are supported for producers and consumers:

- Avro Specific Record types
- Avro Generic Records
- Basic Java Data types: byte[], Byte, Integer, Short, Double, Float, Long, String, Boolean

To get started with Avro schemas and generated Java objects, see the Apache Avro documentation.

Security

You need to include every SSL configuration into a Map that is passed to the Schema Registry configuration.

```
Map<String, String> sslClientConfig = new HashMap<>();
sslClientConfig.put(K_TRUSTSTORE_PATH, params.get(K_SCHEMA_REG_SSL_CLIENT
_KEY + "." + K_TRUSTSTORE_PATH));
sslClientConfig.put(K_TRUSTSTORE_PASSWORD, params.get(K_SCHEMA_REG_SSL_CLI
ENT_KEY + "." + K_TRUSTSTORE_PASSWORD));

Map<String, Object> schemaRegistryConf = new HashMap<>();
schemaRegistryConf.put(K_SCHEMA_REG_URL, params.get(K_SCHEMA_REG_URL));
schemaRegistryConf.put(K_SCHEMA_REG_SSL_CLIENT_KEY, sslClientConfig);
```

For Kerberos authentication, Flink can maintain the authentication and ticket renewal automatically. You can define an additional RegistryClient property to the security.kerberos.login.contexts parameter in Cloudera Manager.

```
security.kerberos.login.contexts=Client,KafkaClient,RegistryClient
```

ClouderaRegistryKafkaSerializationSchema

You can construct the schema serialization with the ClouderaRegistryKafkaSerializationSchema.builder(..) object for FlinkKafkaProducer. You must set the topic configuration and RegistryAddress parameter in the object.

The serialization schema can be constructed using the ClouderaRegistryKafkaSerializationSchema.builder(..) object.

Required settings:

- Topic configuration when creating the builder, which can be static or dynamic (extracted from the data)
- RegistryAddress parameter on the builder to establish the connection

Optinal settings:

- Arbitrary SchemaRegistry client configuration using the setConfig method
- Key configuration for the produced Kafka messages
  - Specifying a KeySelector function that extracts the key from each record
  - Using a Tuple2 stream for (key, value) pairs directly
- Security configuration

```
KafkaSerializationSchema<ItemTransaction> schema = ClouderaRegistryKafkaSeri
alizationSchema
  .<ItemTransaction>builder(topic)
  .setRegistryAddress(registryAddress)
  .setKey(ItemTransaction::getItemId)
  .build();
FlinkKafkaProducer<ItemTransaction> kafkaSink = new FlinkKafkaProducer<>("du
mmy", schema, kafkaProps, FlinkKafkaProducer.Semantic.AT_LEAST_ONCE);
```

ClouderaRegistryKafkaDeserializationSchema

You can construct the schema deserialization with the ClouderaRegistryKafkaDeserializationSchema.builder(..) object for FlinkKafkaProducer to read the messages in the same schema from the FlinkKafkaProducer. You must set the class or schema of the input messages and the RegistryAddress parameter in the object.

The deserialization schema can be constructed using the ClouderaRegistryKafkaDeserializationSchema.builder(..) object.

When reading messages (and keys), you always have to specify the expected Class<T> or record Schema of the input records. This way Flink can do any necessary conversion between the raw data received from Kafka and the expected output of the deserialization.

Required settings:

- Class or Schema of the input messages depending on the data type
- RegistryAddress parameter on the builder to establish the connection

Optinal settings:

- Arbitrary SchemaRegistry client configuration using the setConfig method
- Key configuration for the consumed Kafka messages (only to be specified if reading the keys into a key or value stream is necessary)
- Security configuration

```
KafkaDeserializationSchema<ItemTransaction> schema = ClouderaRegistryKafkaDe
serializationSchema
  .builder(ItemTransaction.class)
  .setRegistryAddress(registryAddress)
  .build();
FlinkKafkaConsumer<ItemTransaction> transactionSource = new FlinkKafkaConsu
mer<>(inputTopic, schema, kafkaProps, groupdId);
```

Kudu with Flink

Cloudera Streaming Analytics offers Kudu connector as a sink to create analytical application solutions. Kudu is an analytic data storage manager. When using Kudu with Flink, the analyzed data is stored in Kudu tables as an output to have an analytical view of your streaming application.

You can read Kudu tables into a DataStream using the KuduCatalog with Table API or using the KuduRowInputForm at directly in the DataStream. The difference between the two methods is that when using the KuduRowInputFormat, you need to manually provide information about the table.

For more information about the Kudu sink in DataStream API, see the official documentation.