Cloudera DataFlow for Data Hub 7.2.12

Apache Flink

Date published: 2019-12-16
Date modified: 2021-10-25

https://docs.cloudera.com/

Legal Notice

Contents

Running a simple Flink application

In this example, you will use the Stateless Monitoring Application from the Flink Tutorials to build your Flink project, submit a Flink job and monitor your Flink application using the Flink Dashboard in an unsecured environment.

Before you begin

- You have a CDP Public Cloud environment.
- You have a CDP username (it can be your own CDP user or a CDP machine user) and a password set to access Data Hub clusters.

  The predefined resource role of this user is at least **EnvironmentUser**. This resource role provides the ability to view Data Hub clusters and set the FreeIPA password for the environment.
- Your user is synchronized to the CDP Public Cloud environment.
- You have a Streaming Analytics cluster.

Procedure

1. Clone the simple tutorial from git:

   ```
   git clone https://github.com/cloudera/flink-tutorials.git
   ```
2. Access the simple tutorial folder:

   ```
   cd flink-tutorials/flink-simple-tutorial
   ```
3. Build your Flink project using maven:

   ```
   mvn clean package
   ```
4. Upload the Flink project to your cluster.

   ```
   scp target/flink-simple-tuto
   rial-<flink_version>-<csa_version>.jar <your_workload_username>@<master_node_FQDN>:.
   ```

   Provide your workload password when prompted.
5. SSH into the node where you uploaded the job jar using your workload username.

   ```
   ssh <your_workload_username>@<master_node_FQDN>
   ```

   Provide your workload password when prompted.
6. Run the Flink application:

   ```
   flink run -d -p 2 -ynm HeapMonitor target/flink-simple-tutor
   ial-<flink_version>-<csa-version>.jar
   ```

   Note: In case you encounter an error about missing mapped role for the user, you need to make sure that you have set the correct access rights. For more information, see the *Give users access to your cluster* section.
7. Navigate to Management Console > Environments , and select the environment where you have created your cluster.
8. Select the Streaming Analytics cluster.
9. Click Flink Dashboard from the services.
   The Flink Dashboard opens in a new window.

10. Click Task Manager on the left side menu.
11. Monitor your Flink application under logs.

Related Information

Give users access to your cluster

Application development

Flink application structure

You must understand the parts of application structure to build and develop a Flink streaming application. To create and run the Flink application, you need to create the application logic using the DataStream API.

A Flink application consists of the following structural parts:

- Creating the execution environment
- Loading data to a source
- Transforming the initial data
- Writing the transformed data to a sink
- Triggering the execution of the program

StreamExecutionEnvironment

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvi
ronment();
...
env.execute("Flink Streaming Secured Job Sample")
```

The getExecutionEnvironment() static call guarantees that the pipeline always uses the correct environment based on the location it is executed on. When running from the IDE, a local execution environment, and when running from the client for cluster submission, it returns the YARN execution environment. The rest of the main class defines the application sources, processing flow and the sinks followed by the execute() call. The execute call triggers the actual execution of the pipeline either locally or on the cluster. The StreamExecutionEnvironment class is needed to configure important job parameters for maintaining the behavior of the application and to create the DataStream.

Related Information

Create Atlas entity type definitions
Flink Project Template
Simple Tutorial: Application logic
Stateful Tutorial: Build a Flink streaming application
Apache Flink documentation: DataStream API overview

Source, operator and sink in DataStream API

A DataStream represents the data records and the operators. There are pre-implemented sources and sinks for Flink, and you can also use custom defined connectors to maintain the dataflow with other functions.

```
DataStream<String> source = env.addSource(consumer)
    .name("Kafka Source")
    .uid("Kafka Source")
    .map(record -> record.getId() + "," + record.getName() + "," + record
.getDescription())
    .name("ToOutputString");
  StreamingFileSink<String> sink = StreamingFileSink
```

```
    .forRowFormat(new Path(params.getRequired(K_HDFS_OUTPUT)), new SimpleSt
ringEncoder<String>("UTF-8"))
    .build();
  source.addSink(sink)
    .name("FS Sink")
    .uid("FS Sink");
  source.print();
```

Choosing the sources and sinks depends on the purpose of the application. As Flink can be implemented in any kind of an environment, various connectors are available. In most cases, Kafka is used as a connector as it has streaming capabilities and can be easily integrated with other services.

Sources

Sources are where your program reads its input from. You can attach a source to your program by using StreamExecutionEnvironment.addSource(sourceFunction). Flink comes with a number of pre-implemented source functions. For the list of sources, see the Apache Flink documentation.

Streaming Analytics in Cloudera supports the following sources:

- HDFS
- Kafka

Operators

Operators transform one or more DataStreams into a new DataStream. When choosing the operator, you need to decide what type of transformation you need on your data. The following are some basic transformation:

- Map

  Takes one element and produces one element.

  ```
  dataStream.map()
  ```

- FlatMap

  Takes one element and produces zero, one, or more elements.

  ```
  dataStream.flatMap()
  ```

- Filter

  Evaluates a boolean function for each element and retains those for which the function returns true.

  ```
  dataStream.filter()
  ```

- KeyBy

  Logically partitions a stream into disjoint partitions. All records with the same key are assigned to the same partition. This transformation returns a KeyedStream

  ```
  dataStream.keyBy() // Key by field "someKey"
  dataStream.keyBy() // Key by the first element of a Tuple
  ```

- Window

  Windows can be defined on already partitioned KeyedStreams. Windows group the data in each key according to some characteristic (e.g., the data that arrived within the last 5 seconds).

  ```
  dataStream.keyBy().window(TumblingEventTimeWindows.of(Time.s
  econds(5))); // Last 5 seconds of data
  ```

For the full list of operators, see the Apache Flink documentation.

Sinks

Data sinks consume DataStreams and forward them to files, sockets, external systems, or print them. Flink comes with a variety of built-in output formats that are encapsulated behind operations on the DataStreams. For the list of sources, see the Apache Flink documentation.

Streaming Analytics in Cloudera supports the following sinks:

- Kafka
- HBase
- Kudu
- HDFS

Related Information

Apache Flink documentation: Operators
Apache Flink documentation: Window operator
Apache Flink documentation: Generating watermarks
Apache Flink documentation: Working with state
Apache Flink documentation: User defined functions

Flink application example

The following is an example of a Flink application logic from the Secure Tutorial. The application is using Kafka as a source and writing the outputs to an HDFS sink.

```
public class KafkaToHDFSAvroJob {
 private static Logger LOG = LoggerFactory.getLogger(KafkaToHDFSAvroJob.cl
ass);
 public static void main(String[] args) throws Exception {
  ParameterTool params = Utils.parseArgs(args);

  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecution
Environment();

  KafkaDeserializationSchema<Message> schema = ClouderaRegistryKafkaDeserial
izationSchema
    .builder(Message.class)
    .setConfig(Utils.readSchemaRegistryProperties(params))
    .build();
  FlinkKafkaConsumer<Message> consumer = new FlinkKafkaConsumer<Message>(
params.getRequired(K_KAFKA_TOPIC), schema, Utils.readKafkaProperties(params)
);

  DataStream<String> source = env.addSource(consumer)
    .name("Kafka Source")
    .uid("Kafka Source")
    .map(record -> record.getId() + "," + record.getName() + "," + record.g
etDescription())
    .name("ToOutputString");
  StreamingFileSink<String> sink = StreamingFileSink
    .forRowFormat(new Path(params.getRequired(K_HDFS_OUTPUT)), new Simple
StringEncoder<String>("UTF-8"))
    .build();
  source.addSink(sink)
    .name("FS Sink")
    .uid("FS Sink");
  source.print();

  env.execute("Flink Streaming Secured Job Sample");
 }
}
```

Testing and validating Flink applications

After you have built your Flink streaming application, you can create a simple testing method to validate the correct behaviour of your application.

Pipelines can be extracted to static methods and can be easily tested with the JUnit framework.

A simple JUnit test can be written to verify the core application logic. The test is implemented in the test class and should be regarded as an integration test of the application flow.

The test mimics the application main class with only minor differences:

1. Create the StreamExecutionEnvironment the same way.
2. Use the env.fromElements(..) method to pre-populate a DataStream with some testing data.
3. Feed the testing data to the static data processing logic as before.
4. Verify the correctness once the test is finished.

```
    @Test
    public void testPipeline() throws Exception {
        final String alertMask = "42";
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExec
utionEnvironment();
        HeapMetrics alert1 = testStats(0.42);
        HeapMetrics regular1 = testStats(0.452);
        HeapMetrics regular2 = testStats(0.245);
        HeapMetrics alert2 = testStats(0.9423);

        DataStreamSource<HeapMetrics> testInput = env.fromElements(alert1,
 alert2, regular1, regular2);
        HeapMonitorPipeline.computeHeapAlerts(testInput, ParameterTool.fro
mArgs(new String[]{"--alertMask", alertMask}))
                .addSink(new SinkFunction<HeapAlert>() {
                    @Override
                    public void invoke(HeapAlert value) {
                        testOutput.add(value);
                    }
                })
                .setParallelism(1);
        env.execute();
        assertEquals(Sets.newHashSet(HeapAlert.maskRatioMatch(alertMask, al
ert1),
                HeapAlert.maskRatioMatch(alertMask, alert2)), testOutput);
    }
    private HeapMetrics testStats(double ratio) {
        return new HeapMetrics(HeapMetrics.OLD_GEN, 0, 0, ratio, 0, "test
host");
    }
}
```

Related Information

Simple Tutorial: Testing the data pipeline

Stateful Tutorial: Test and validate the streaming pipeline

Flink Project Template

The Quickstart Archetype serves as a template for a Flink streaming application. You can use the Archetype to add source, sink and computation to the template. Like this you can practice the development of a simple Flink application, or use the Archetype as the starting point for a more complex application including state,watermark and checkpoint.

About this task

The Flink quickstart archetype can be used to quickly build a basic Flink streaming project.

Note: You need to install the archetype locally on your host as Cloudera does not release maven archetypes to the Maven Central Repository.

Procedure

1. Perform the following commands to create the archetype locally:

```
git clone https://github.com/cloudera/flink-tutorials
cd flink-tutorials
cd flink-quickstart-archetype
mvn clean install
cd ..
```

   The following entry should be seen in your local catalog:

```
cat ~/.m2/repository/archetype-catalog.xml
 ...
 <archetypes>
    <archetype>
      <groupId>com.cloudera.flink</groupId>
      <artifactId>flink-quickstart-archetype</artifactId>
      <version>1.10.0-csa1.2.0.0</version>
      <description>flink-quickstart-archetype</description>
    </archetype>
  </archetypes>
```

2. Generate the project skeleton with the following commands:

```
mvn archetype:generate \
      -DarchetypeGroupId=com.cloudera.flink \
      -DarchetypeArtifactId=flink-quickstart-archetype \
      -DarchetypeVersion=1.10.0-csa1.2.0.0
```

3. Provide basic information about the streaming application.

   You can choose to customize configurations yourself or use automatically generated information.

   - To use automatically generated information, press Enter.
   - If you choose to customize configuration, set the following properties:

```
Define value for property 'groupId': com.cloudera.flink
                                     Define value for property 'artifactId':
sample-project
                                     Define value for property 'version' 1.0-
SNAPSHOT: :
                                     Define value for property 'package' c
om.cloudera.flink: :
                                     Confirm properties configuration:
                                     groupId: com.cloudera.flink
                                     artifactId: sample-project
                                     version: 1.0-SNAPSHOT
                                     package: com.cloudera.flink
                                     Y: :
```

   The generated project will look like this:

```
sample-project
### pom.xml
```

```
### src
    ### main
        ### java
        #   ### com
        #       ### cloudera
        #           ### flink
        #               ### StreamingJob.java
        ### resources
            ### log4j.properties
```

4. Open StreamingJob.java file.

   The archetype application will look like this:

```
public class StreamingJob {
 public static void main(String[] args) throws Exception {
  final StreamExecutionEnvironment env = StreamExecutionEnvironment.getEx
ecutionEnvironment();
  DataStream<Integer> ds = env.fromElements(1,2,3,4);
  ds.printToErr();
  env.execute("Flink Streaming Java API Skeleton");
 }
}
```

5. Customize the archetype application by adding source, stream transformation and sink to the Datastream class.
6. Run the application with env.execute command.

Results

You have built your Flink streaming project.

What to do next

You can further develop your application, or you can run the Flink application archetype.

Configuring Flink applications

Cloudera Streaming Analytics includes Flink with configuration that works out of the box. It is not mandatory to configure Flink to production, but you can use the available configurations to optimize the application behavior in production. Cloudera Manager includes all the necessary configurations for Flink that can also be accessed from the flink-conf.yaml file.

Setting parallelism and max parallelism

The max parallelism is the most essential part of resource configuration for Flink applications as it defines the maximum jobs that are executed at the same time in parallel instances. However, you can optimize max parallelism in case your production goals differ from the default settings.

In a Flink application, the different tasks are split into several parallel instances for execution. The number of parallel instances for a task is called parallelism. Parallelism can be defined at the operator, client, execution environment and system level. Cloudera recommends setting parallelism to a lower value at first use, and increasing it over time if the job cannot keep up with the input rate.

To configure the max parallelism, setMaxParallelism is called as it controls the number of key-groups created by the state backends. A key-group is a partition of an operator state. The number of key-groups determines how data is going to be distributed among the parallel operators. If the key-groups are not distributed evenly, the data distribution is also uneven.

Consider the following aspects when setting the max parallelism:

- The number should be large enough to accommodate expected future load increases as this setting cannot be changed without starting from an empty state.

- If P is the selected parallelism for the job, the max parallelism should be divisible by P to get even state distribution.
- Please note that larger max parallelism settings have greater cost on the state backend side, for large scale production jobs benchmarking the size of the state based on the maximum parallelism is useful before changing this parameter.

Based on these criteria, Cloudera recommends setting the max paralellism to factorials or other numbers with a large number of divisors (120, 180, 240, 360, 720, 840, 1260), which will make parallelism tuning easier.

Table 1: Reference values

Stateless	In-memory state	RocksDB state
1 million record / sec / core	100 000 records / sec / core	10 000 records / sec / core

Configuring Flink application resources

Generally, Flink automatically identifies the required resources for an application based on the parallelism settings. However, you can adjust the configurations based on your requirements by specifying the number of task managers and their memory allocation for individual Flink applications or for the entire Flink deployment.

To control the resources of individual TaskManager processes and the amount of work allocated to them, Cloudera recommends starting the configuration with the following options:

Number of Task Slots

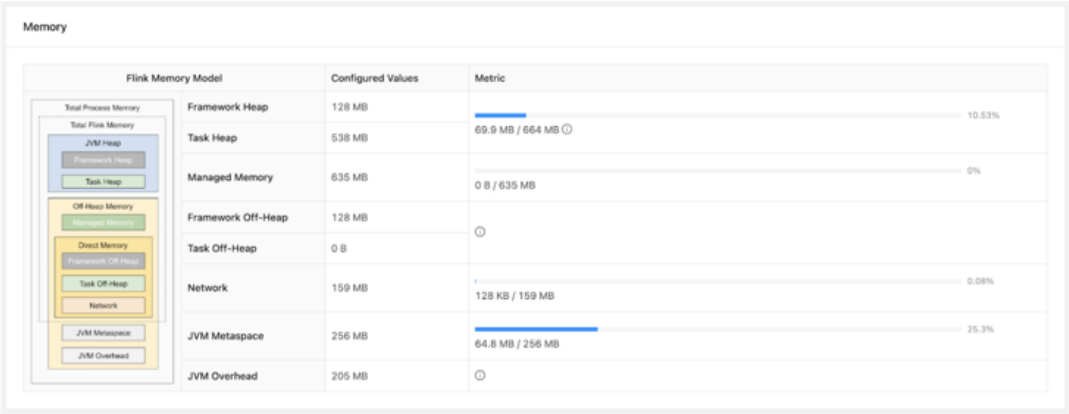
The number of task slots controls how many parallel pipeline/operator instances can be executed in a single TaskManager. Together with the parallelism setting, you can ultimately define how many TaskManagers will be allocated for the job. For example, if you set the job parallelism to 12 and the taskmanager.numberOfTaskSlots to 4, there will be 3 TaskManager containers for the job as the value of parallelism will be divided with the number of task slots.

You can set the number of task slots in Cloudera Manager under the Configuration tab.

TaskManager Process Memory Size

The taskmanager.memory.process.size option controls the total memory size of the TaskManager containers. For applications that store data on heap or use large state sizes, it is recommended towe should increase the process size accordingly. You can set the number of task slots in Cloudera Manager under the Configuration tab.

For more information about the TaskManager memory management, see the Apache Flink documentation. You can also check the TaskManager configuration of your running application on the Flink Dashboard to review the configured values before making adjustments.



Network buffers for throughput and latency

Flink uses network buffers to transfer data from one operator to another. These buffers are filled up with data during the specified time for the timeout. In case of high data rates, the set time is usually never reached. For cases when the data rate is high, the throughput can be further increased with setting the buffer timeout to an intentionally higher value due to the characteristics of the TCP channel. However, this in turn increases the latency of the pipeline.

Yarn Related Configurations

Flink on YARN jobs are configured to tolerate a maximum number of failed containers before they terminate. You can configure the YARN maximum failed containers setting in proportion to the total parallelism and the expected lifetime of the job.

High Availability is enabled by default in CSA. This eliminates the JobManager as a single point of failure. You can also tune the application resilience by setting the YARN maximum application attempts, which determines how many times the application will retry in case of failures.

Furthermore, you can use a YARN queue with preemption disabled to avoid long running jobs being affected when the cluster reaches its capacity limit.

Reference values for the configurations

Configuration	Parameter	Recommended value
TM container memory	-ytm / taskmanager.heap.size	*TM Heap + Heap-cutoff*
Managed Memory Fraction	taskmanager.memory.managed.fraction	*0.4 - 0.9*
Max parallelism	pipeline.max-parallelism	*120,720,1260,5040*
Buffer timeout	execution.buffer-timeout	*1-100*
YARN queue	-yqu	*A queue with no preemption*
YARN max failed containers	yarn.maximum-failed-containers	*3*num_containers*
YARN max AM failures	yarn.application-attempts	*3-5*

Configuring RocksDB state backend

You can use RocksDB as a state backend when your Flink streaming application requires a larger state that doesn't fit easily in memory. The RocksDB state backend uses a combination of fast in-memory cache and optimized disk based lookups to manage state.

You can configure the state backend for your streaming application by using the state.backend parameter directly or in Cloudera Manager under the Configuration tab:

You can adjust how much memory RocksDB should use as a cache to increase lookup performance by setting the memory managed fraction of the TaskManagers in Cloudera Manager under the Configuration tab:

The default fraction value is 0.4, but with larger cache requirements you need to increase this value together with the total memory size.

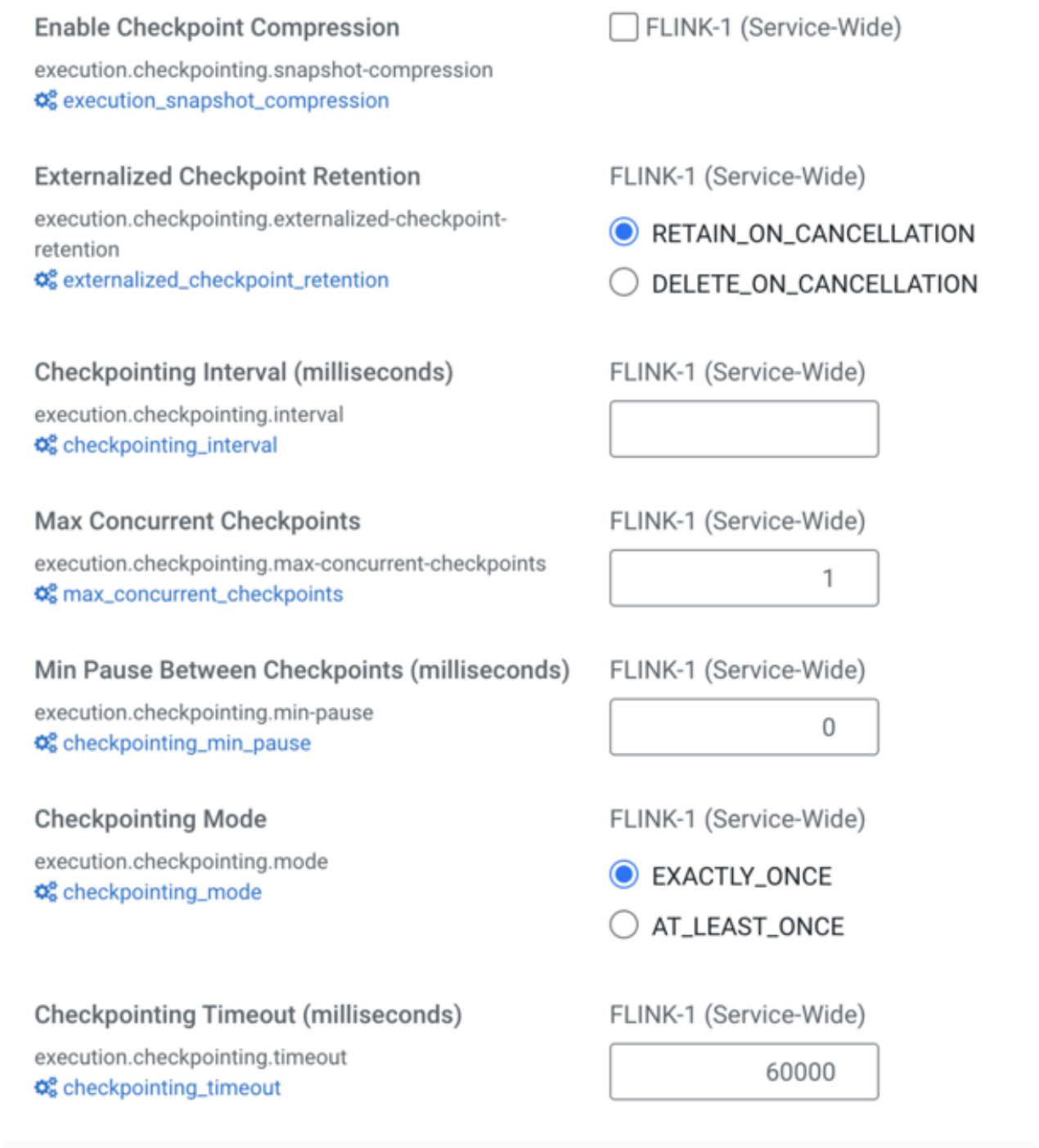
Enabling checkpoints for Flink applications

To make your Flink application fault tolerant, you need to enable automatic checkpointing. When an error or a failure occurs, Flink will automatically restarts and restores the state from the last successful checkpoint. Checkpointing is not enabled by default.

While it is possible to enable checkpointing programmatically through the StreamExecutionEnvironment, Cloudera recommends to enable checkpointing either using the configuration file for each job, or as a default configuration for all Flink applications through Cloudera Manager.

To enable checkpointing, you need to set the execution.checkpointing.interval configuration option to a value larger than 0. It is recommended to start with a checkpoint interval of 10 minutes (600000 milliseconds).

You can acces the configuration options of checkpointing in Cloudera Manager under the Configuration tab.



DataStream connectors

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 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 run 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
- 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 Runtime 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

Optional 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

Optional 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, groupId);
```

Kafka Metrics Reporter

In Cloudera Streaming Analytics, Kafka Metrics Reporter is available as another monitoring solution when Kafka is used as a connector within the pipeline to retrieve metrics about your streaming performance.

Flink offers a flexible Metrics Reporter API for collecting the metrics generated by your streaming pipelines. Cloudera provides an additional implementation of this, which writes metrics to Kafka with the following JSON schema:

```
{
  "timestamp" : number -> millisecond timestamp of the metric record
  "name" : string -> name of the metric
    (e.g. numBytesOut)
  "type" : string -> metric type enum: GAUGE, COUNTER, METER, HISTOGRAM
  "variables" : {string => string} -> Scope variables
    (e.g. {"<job_id>" : "123", "<host>" : "localhost"})
  "values" : {string => number} -> Metric specific values
    (e.g. {"count" : 100})
}
```

For more information about Metrics Reporter, see the Apache Flink documentation.

Configuration of Kafka Metrics Reporter

The Kafka metrics reporter can be configured similarly to other upstream metric reporters.

Required parameters

- topic: target Kafka topic where the metric records will be written at the configured intervals
- bootstrap.servers: Kafka server addresses to set up the producer

Optional parameters

- interval: reporting interval, default value is 10 seconds, format is 60 SECONDS
- log.errors: logging of metric reporting errors, value either true or false

You can configure the Kafka metrics reporter per job using the following command line properties:

```
flink run -d -p 2 -ynm HeapMonitor \
-yD metrics.reporter.kafka.class=org.apache.flink.metrics.kafka.KafkaMetric
sReporter \
-yD metrics.reporter.kafka.topic=metrics-topic.log \
-yD metrics.reporter.kafka.bootstrap.servers=<kafka_broker>:9092 \
-yD metrics.reporter.kafka.interval="60 SECONDS" \
-yD metrics.reporter.kafka.log.errors=false \
flink-simple-tutorial-1.3-SNAPSHOT.jar
```

The following is a more advanced Flink command that also contains security related configurations:

```
flink run -d -p 2 -ynm HeapMonitor \
-yD security.kerberos.login.keytab=some.keytab \
-yD security.kerberos.login.principal=some_principal \
-yD metrics.reporter.kafka.class=org.apache.flink.metrics.kafka.KafkaMetric
sReporter \
-yD metrics.reporter.kafka.topic=metrics-topic.log \
-yD metrics.reporter.kafka.bootstrap.servers=<kafka_broker>:9093 \
-yD metrics.reporter.kafka.interval="60 SECONDS" \
-yD metrics.reporter.kafka.log.errors=false \
-yD metrics.reporter.kafka.security.protocol=SASL_SSL \
-yD metrics.reporter.kafka.sasl.kerberos.service.name=kafka \
-yD metrics.reporter.kafka.ssl.truststore.location=truststore.jks \
flink-simple-tutorial-1.3-SNAPSHOT.jar
```

You can also set the metrics properties globally in Cloudera Manager using Flink Client Advanced Configuration Snippet (Safety Valve) for flink-conf-xml/flink-conf.xml.

Arbitrary Kafka producer properties

The reporter supports passing arbitrary Kafka producer properties that can be used to modify the behavior, enable security, and so on. Serializer classes should not be modified as it can lead to reporting errors.

See the following example configuration of the Kafka Metrics Reporter:

```
# Required configuration
metrics.reporter.kafka.class:
org.apache.flink.metrics.kafka.KafkaMetricsReporter
metrics.reporter.kafka.topic: metrics-topic.log
metrics.reporter.kafka.bootstrap.servers: broker1:9092,broker2:9092

# Optional configuration
metrics.reporter.kafka.interval: 60 SECONDS
metrics.reporter.kafka.log.errors: false

# Optional Kafka producer properties
metrics.reporter.kafka.security.protocol: SSL
metrics.reporter.kafka.ssl.truststore.location:
/var/private/ssl/kafka.client.truststore.jks
```

Note: Any optional property with metrics.reporter.kafka. prefix tag is processed as Kafka client configuration.

For example: metrics.reporter.kafka.property_name: property_value will be converted to property_name: p roperty_value.

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.

Job lifecycle

Running a Flink job

After developing your application, you can submit your Flink job in YARN per-job or session mode. To submit the Flink job, you need to run the Flink client in the command line including security parameters and other configurations with the run command.

About this task

Submitting a job means uploading the job's JAR and related dependencies to the Flink cluster and initiating the job execution.

The Flink jobs you submit to the cluster are running on YARN. Submitting a job means that the JAR file of the Flink application is uploaded to the cluster with the related dependencies. and the job execution is initiated. You have the following mode in which you can run your Flink jobs:

- Per-job mode

  Per-job mode means that you run the Flink job in a dedaction YARN application. In this case each submitted Flink job has its own Flink cluster in YARN, with its own Job Manager and Task Managers. When you run Flink jobs in per-job mode, every job submission creates a new cluster. As the cluster deployment has to be created with every submission, the execution of the job can take up time.
- Session mode

  Session mode means that you run multiple Flink jobs in the same YARN sessions. In this case every Flink job shares the cluster, the allocated resources, the Job Manager and Task Managers. When you run Flink jobs in session mode, the submitted jobs are created in one cluster and are long-lived. The execution time is shorter than in per-job mode, however you need to consider that in a session mode a cluster failure affects every Flink job, and recreation from a savepoint can take up time.

You can set how to run your Flink job with the execution.target setting in the Flink configuration file. By default, execution.target is set to yarn-per-job, but you can change it to yarn-session. Alternatively, you can add the corresponding arguments to the flink run command when submitting the Flink job.

Before you begin

- You have installed and configured the Flink service on your cluster.

  For more information, see the Adding Flink as a service documentation.
- You have HDFS Gateway, Flink and YARN Gateway roles assigned to the host you are using for Flink submission.

  For more information, see the Cloudera Manager documentation.

- You have uploaded the Flink application JAR file and job properties file to the Flink cluster.

Procedure

1. Connect to the cluster using ssh where you want to run the Flink application.

```
ssh root@<your_host_name>
```

Note: You are prompted to provide your password to the cluster.

2. Submit the Flink job using the flink run command.

Per-job mode

```
flink run \
  -d \
  -m yarn-cluster \
  -ynm <job_name_in_yarn> \
  -p <job_parallelism> \
  -ys <slots_per_task_manager> \
  -ytm <memory_per_container_in_mb> \
  <job_jar_file> <job_parameters> …
```

Session mode

a. Start a Flink session cluster.

```
flink-yarn-session \
  -d \
  -nm <job_name_in_yarn> \
  -s <slots_per_task_manager> \
  -tm <memory_per_container_in_mb>
```

The flink-yarn-session command outputs the ID of the corresponding YARN application. You need to add the YARN application ID to the flink run command.

```
YARN ApplicationID: application_1616633166424_0024
```

b. Submit the Flink job.

```
flink run \
  -d \
  -m yarn-cluster \
  -e yarn-session \
  -yid <application_id> \
  -p <parallelism> \
  <job_jar_file> <job_parameters>
```

Note:

To run a Flink job, your HDFS Home Directory has to exist. If it does not exist, you receive an error message similar to:

```
Permission denied: user=$USER_NAME, access=WRITE, inode="/user"
```

Related Information

Simple Tutorial: Running the application from IntelliJ

Simple Tutorial: Running the application on a Cloudera cluster

Stateful Tutorial: Deploy and monitor the application

Using Flink CLI

You can use the Flink command line interface to operate, configure and maintain your Flink applications.

The Flink CLI works without requiring the user to always specify the YARN application ID when submitting commands to Flink jobs. Instead, the jobs are identified uniquely on the YARN cluster by their job IDs.

The following improvements are implemented for Flink CLI:

- flink list: This command lists all the jobs on the YARN cluster by default, instead of listing the jobs of a single Flink cluster.
- flink savepoint <jobId> and flink cancel <jobId>: The savepoint and cancel commands, along with the other single job commands, no longer require the -yId parameter, and work if you provide only the ID of the job.
- flink run: You do not need to specify -m yarn-cluster, as it is included in the run command by default.

Enabling savepoints for Flink applications

Beside checkpointing, you are also able to create a savepoint of your executed Flink jobs. Savepoints are not automatically created, so you need to trigger them in case of upgrade or maintenance. You can also resume your applications from savepoint.

You can set the default savepoint directory in flink-conf.yaml under state.savepoints.dir property.

The following command lines can be used to maintain savepoints:

Trigger savepoint	$ bin/flink savepoint -yid <*yarnAppID*> <*jobId*> [targetDirectory]
Stop job with savepoint	$ bin/flink stop -yid <*yarnAppID*> <*jobId*>
Resume from savepoint	$ bin/flink run -s <*savepointPath*> [runArgs]
Deleting savepoint	$ bin/flink savepoint -d <*savepointPath*>

Monitoring

Flink Dashboard

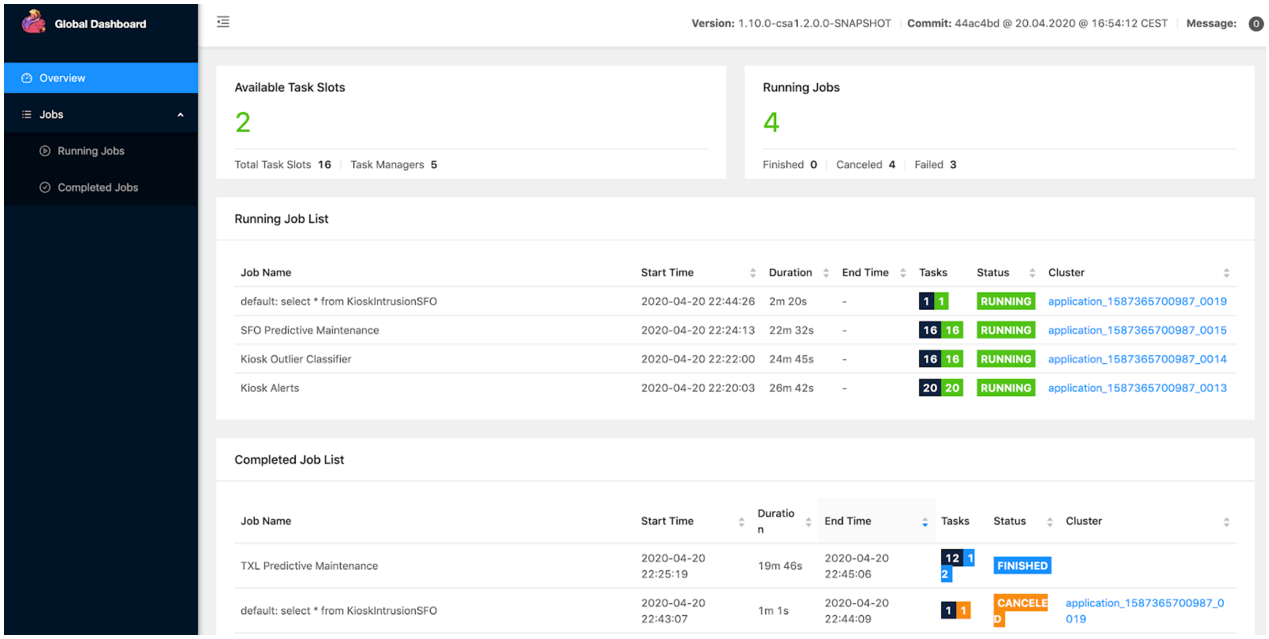
The Flink Dashboard is a built-in monitoring interface for Flink applications in Cloudera Streaming Analytics. You can monitor your running, completed and stopped Flink jobs on the dashboard. You reach the Flink Dashboard through Cloudera Manager.

After deploying Flink and the required components, you can configure and monitor each component individually, or the whole cluster with Cloudera Manager. For the general use of Cloudera Manager, see the Cloudera Manager documentation.

The Flink Dashboard acts as a single UI for monitoring all the jobs running on the YARN cluster. It shows all the running, failed, and finished jobs.

Note: The Flink Dashboard is an updated version of the Flink HistoryServer.

You can also use the dashboard to navigate between the different Flink clusters from a central place.



Streams Messaging Manager integration

You can use the Streams Messaging Manager (SMM) UI to monitor end-to-end latency of your Flink application when using Kafka as a datastream connector.

End-to-end latency throughout the pipeline can be monitored using SMM. To use SMM with Flink, interceptors need to be enabled for Kafka in the Flink connectors.

For more information about enabling interceptors, see the SMM documentation.

Enabling Flink DEBUG logging

You can review the log text files of the Flink jobs when an error is detected during the processes. When you set the log level of Flink to DEBUG, you can easily trace the log file for errors.

About this task

A log file is created for every Flink process that contains messages for the different events happening in the given process. You can use these log files to solve the errors and problems that can occur during Flink processes. You can access the Flink logs using the Flink Dashboard.

Procedure

1. Navigate to the **Configuration** page in Cloudera Manager.
   a) Navigate to Management Console > Environments , and select the environment where you have created your cluster.
   b) Select the Streaming Analytics cluster from the list of Data Hub clusters.
   c) Select Cloudera Manager from the services.
      You are redirected to the **Cloudera Manager** user interface.
   d) Click Clusters > Flink .
   e) Click Configuration.
2. Search for Flink Client Advanced Configuration Snippet (Safety Valve) for flink-conf/log4j.properties configuration.

3. Add the following parameters to the Safety Valve:

```
logger.flink.name = org.apache.flink
logger.flink.level = DEBUG
```

4. Click Save Changes.
5. Restart the Flink service with Action > Restart .

   After updating the Flink log configuration in Cloudera Manager, you need to access the YARN Resource Manager user interface from the Streaming Analytics cluster page.
6. Access the YARN Resource Manager user interface to stop the YARN job of the Flink application.
   a) Navigate to Management Console > Environments, and select the environment where you have created your cluster.
   b) Select the Streaming Analytics cluster.
   c) Select Resource Manager from the list of Services.

      You are redirected to the **Resource Manager** user interface.
   d) Select Applications.

      The running Flink applications are displayed.
7. Select the application you need to stop.
8. Click Settings.
9. Select Kill application.

   After stopping the Flink application, you can review the log file of the Flink job by accessing the Flink Dashboard from the Streaming Analytics cluster page.
10. Navigate to **Flink Dashboard** and review the log level for the running job.
    a) Navigate to Management Console > Environments, and select the environment where you have created your cluster.
    b) Select the Streaming Analytics cluster.
    c) Click Flink Dashboard from the list of services.
11. Select Task Managers from the main menu.
12. Select the previously submitted job.
13. Click Logs.

Governance

Flink metadata collection using Atlas

In Cloudera Streaming Analytics, you can use Flink with Apache Atlas to track the input and output data of your Flink jobs.

Atlas is a lineage and metadata management solution that is supported across the Cloudera Data Platform. This means that you can find, organize and manage different assets of data about your Flink applications and how they relate to each other. This enables a range of data stewardship and regulatory compliance use cases.

For more information about Atlas, see the Cloudera Runtime documentation.

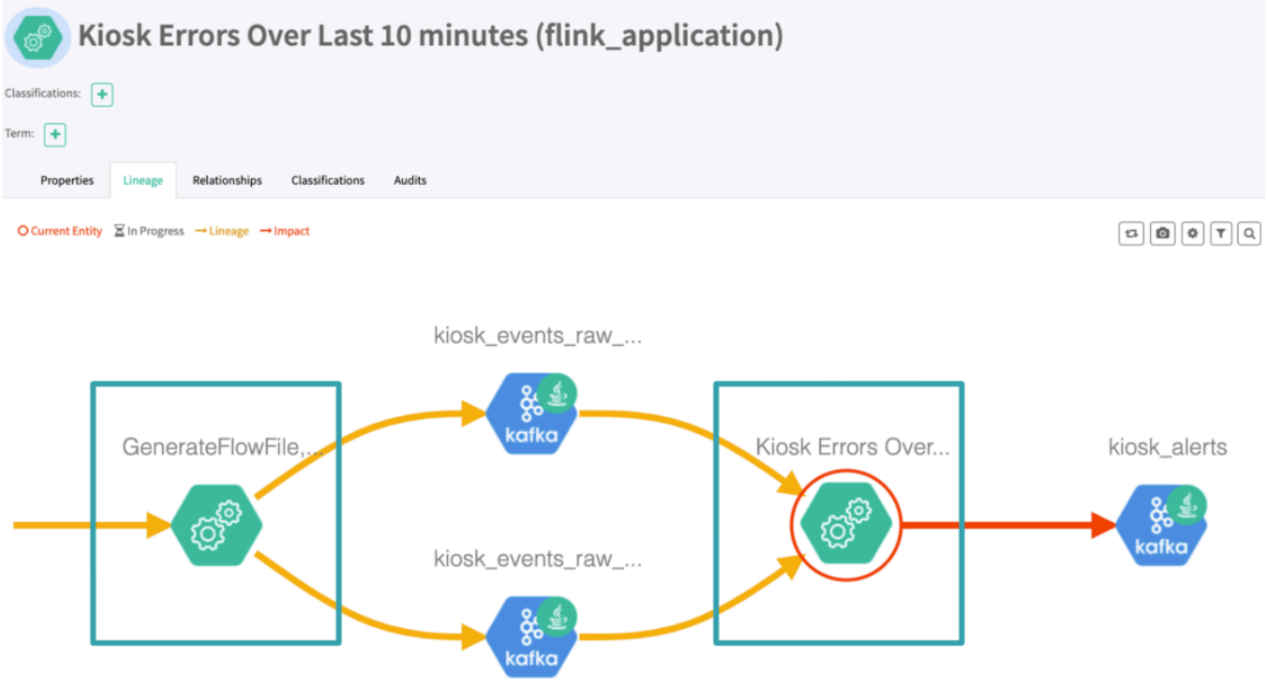
Related Information

Create Atlas entity type definitions

Atlas entities in Flink metadata collection

In Atlas, the core concept of representing Flink applications, Kafka topics, HBase tables, and so on, is called an entity. You need to understand the relation and definition for entities in a Flink setup to enhance the metadata collection.

When submitting updates to Atlas, a Flink application describes itself and the entities it uses as sources and sinks. Atlas creates and updates the corresponding entities, and creates lineage from the collected and already available entities. Internally, the communication between the Flink client and the Atlas server is implemented using a Kafka topic. This solution is referred to as Flink hook by the Atlas community.



Related Concepts

Flink application structure

Related Information

Create Atlas entity type definitions

Flink Project Template

Simple Tutorial: Application logic

Stateful Tutorial: Build a Flink streaming application

Apache Flink documentation: DataStream API overview

Migrating Flink jobs

After upgrading the version of the Data Lake in your CDP Public Cloud environment, you need to create Data Hub clusters using the Streaming Analytics cluster template that corresponds to the new version of the Data Lake. To restart your existing Flink jobs on the upgraded Data Hub clusters, you need to migrate your Flink jobs. The migration process depends on the state of the Flink jobs.

After creating a new Data Hub cluster using the latest Streaming Analytics cluster template, you need to migrate your Flink jobs to the new cluster. The process of migration depends on the state of your Flink jobs: if you have stateful jobs, you need to stop your Flink applications with creating savepoints. Before starting the job migration, make sure that the job dependies are updated based on the latest version of Flink in CDP Public Cloud.

Migrating Flink jobs without state

When you run Flink application without state, you only need to stop your currently running Flink jobs, and restart them on your upgraded Data Hub clusters.

Procedure

1. Connect to your host where you run your Flink jobs using ssh.

```
ssh <workload_username>@<master_host>
Password: <your_workload_password>
```

2. Determine which jobs you want to stop by listing the Flink job IDs.

```
flink list
```

3. Stop your Flink jobs using command line interface (CLI).

```
flink cancel <flink_job_ID>
```

4. Resubmit the stopped Flink jobs on your new cluster.
   a) Connect to your host where you intend to run your Flink jobs using ssh.

```
ssh <workload_username>@<master_host>
Password: <your_workload_password>
```

   b) Resubmit your Flink application.

```
flink run <run_arguments> \
  <jar_file> <app_arguments>
```

Migrating stateful Flink jobs

When you run Flink application with state, you must stop the Flink jobs with a savepoint, so you can restart them from the exact point on your upgraded Data Hub clusters. Based on the version of your original cluster, you can provide the savepoint path either in HDFS or S3.

About this task

The following savepoint paths are supported based on the version of your original cluster:

- If the version of the original cluster was 7.2.9 or lower, you need to provide the savepoint path in HDFS.
- If the version of the original cluster was 7.2.10 or higher, you need to provide the savepoint path in S3.

Procedure

1. Connect to your host where you run your Flink jobs using ssh.

```
ssh <workload_username>@<master_host>
Password: <your_workload_password>
```

2. Determine which jobs you want to stop by listing the Flink job IDs.

```
flink list
```

3. Stop your Flink jobs with a savepoint using command line interface (CLI).

   If the version of your original cluster was 7.2.10 or higher, store the savepoint in S3:

   ```
   flink stop --savepointPath s3a://<bucket_name>/savepoints <flink_job_id>
   ```

   If the version of your original cluster was 7.2.9 or lower, store the savepoint in HDFS:

   ```
   flink stop --savepointPath hdfs:///tmp/savepoints <flink_job_id>
   ```

4. Take note of the savepoint path in the output of the command as you must provide the path when resuming the Flink applications.
5. Copy the savepoint from HDFS to S3, only if the version of your original cluster was 7.2.9 or lower.
   a) If the savepoint size is minimal:

   ```
   hdfs dfs -mkdir -p s3a://aa-uet2/savepoints/
   hdfs dfs -cp <savepoint_path> s3a://<bucket_name>/savepoints/
   ```

   b) If the savepoint size is large (several GB or larger):

   ```
   hdfs dfs -mkdir -p s3a://aa-uet2/savepoints/
   hadoop distcp -m 10 <savepoint_path> s3a://<bucket_name>/savepoints/
   ```

6. Create the Data Hub cluster with the new version.

   For more information, see the *Creating Streaming Analytics cluster* section.
7. Copy the savepoint from S3 to HDFS, only if the version of your original cluster was 7.2.9 or lower.

   Make sure that you provide the same path on HDFS as the original savepoint location in the original cluster.
   a) If the savepoint size is minimal:

   ```
   hdfs dfs -cp \
     s3a://<bucket_name>/savepoints/<savepoint_folder_name> \
     hdfs:///tmp/savepoints/
   ```

   b) If the savepoint size is large (several GB or larger):

   ```
   hadoop distcp -m 10 \
     s3a://<bucket_name>/savepoints/<savepoint_folder_name> \
     hdfs:///tmp/savepoints/
   ```

8. Update the Flink job dependencies in your POM file, and rebuild your JAR file.

   For more information, see the *Updating Flink job dependencies* section.
9. Resume your Flink application from savepoint.

   If the version of your original cluster was 7.2.10 or higher, restore your application from S3:

   ```
   flink run <run_arguments> \
     -fromSavepoint s3a://<bucket_name>/savepoints \
     <jar_file> <app_arguments>
   ```

   If the version of your original cluster was 7.2.9 or lower, restore your application from HDFS:

   ```
   flink run <run_arguments> \
     -fromSavepoint hdfs:///tmp/savepoints/<savepoint_name> \
     <jar_file> <app_arguments>
   ```

Updating Flink job dependencies

When you migrate your Flink jobs to a cluster that has a new supported version of Flink, the applications need to use a new version of the artifacts provided by the Flink deployment in your cluster. To avoid incompatibilities between the packaged artifacts of your application and the artifacts provided by the Flink cluster, ensure that the POM file of the application is updated to match the Flink version of the new Data Hub cluster.

Procedure

1. Navigate to Management Console > Environments , and select the environment where you have created your cluster.
2. Select the Streaming Analytics cluster from the list of Data Hub clusters.
3. Select Repository Details.
4. Search for the **Cloudera Runtime Repository Specification** section, and review the Flink version.

   Name: FLINK

   Version: 1.13.2-csadh1.5.0.0-cdh7.2.12.0-35-17794544
5. Update the <flink.version> property of your POM file using the Flink and Cloudera Streaming Analyitcs (CSA) version of your Data Hub cluster.

   You need to copy and paste only the prefix before the 'cdh' version number: 1.13.2-csadh1.5.0.0.

```
<properties>
        ...
        <flink.version>1.13.2-csadh1.5.0.0</flink.version>
        ...
    </properties>
    ...
    <dependencies>
        ...
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-java</artifactId>
            <version>${flink.version}</version>
        </dependency>
        ...
    <dependencies>
```

6. Rebuild your JAR file.

```
mvn clean package
```