Cloudera DataFlow for Data Hub 7.2.18

Using Apache Flink

Date published: 2019-12-16
Date modified: 2024-04-03

CLOUDERA

https://docs.cloudera.com/

Legal Notice

© Cloudera Inc. 2024. All rights reserved.

The documentation is and contains Cloudera proprietary information protected by copyright and other intellectual property rights. No license under copyright or any other intellectual property right is granted herein.

Unless otherwise noted, scripts and sample code are licensed under the Apache License, Version 2.0.

Copyright information for Cloudera software may be found within the documentation accompanying each component in a particular release.

Cloudera software includes software from various open source or other third party projects, and may be released under the Apache Software License 2.0 (“ASLv2”), the Affero General Public License version 3 (AGPLv3), or other license terms. Other software included may be released under the terms of alternative open source licenses. Please review the license and notice files accompanying the software for additional licensing information.

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

Cloudera, Cloudera Altus, HUE, Impala, Cloudera Impala, and other Cloudera marks are registered or unregistered trademarks in the United States and other countries. All other trademarks are the property of their respective owners.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER’S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

Contents

Running a simple Flink application....................5

Application development....................5
- Flink application structure....................5
  - Source, operator and sink in DataStream API....................6
  - Flink application example....................9
  - Testing and validating Flink applications....................11
- Configuring Flink applications....................12
  - Setting parallelism and max parallelism....................12
  - Configuring Flink application resources....................13
  - Configuring state backend....................14
  - Enabling checkpoints for Flink applications....................17
  - Configuring PyFlink applications....................18

DataStream connectors....................19
- HBase sink with Flink....................19
  - Creating and configuring the HBaseSinkFunction....................19
- Kafka with Flink....................20
  - Schema Registry with Flink....................22
  - Kafka Metrics Reporter....................25
- Kudu with Flink....................26
- Iceberg with Flink....................26

File systems....................28

Job lifecycle....................29
- Setting up Python for PyFlink....................29
- Running a Flink job....................30
- Using Flink CLI....................33
- Enabling savepoints for Flink applications....................33

Monitoring....................34
- Enabling Flink DEBUG logging....................34
- Flink Dashboard....................35
- Streams Messaging Manager integration....................35

Flink SQL and Table API....................35
- SQL and Table API supported features....................37
- DataStream API interoperability....................37
  - Converting DataStreams to Tables....................37
  - Converting Tables to DataStreams....................38
  - Supported data types....................39
- SQL catalogs for Flink....................39

Hive catalog....40
Kudu catalog....40
Schema Registry catalog....41
SQL connectors for Flink....41
SQL Statements in Flink....47
CREATE Statements....47
DROP Statements....49
ALTER Statements....49
INSERT Statements....49
SQL Queries in Flink....50

Flink metadata collection using Atlas....50
Atlas entities in Flink metadata collection....51
Creating Atlas entity type definitions for Flink....51
Verifying metadata collection....53

Reference....53
Flink Terminology....53
Cloudera Flink Tutorials....54

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.

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 <location>/flink-stateful-tutorial-1.2-SNAPSHOT.jar root@<your_hostn
ame>:.
```

Note: Provide your password when prompted.

5. Run the Flink application:

```
flink run -d -p 2 -ynm HeapMonitor target/flink-simple-tutorial-1.2-SNAP
SHOT.jar
```

6. Navigate to Management Console > Environments , and select the environment where you have created your cluster.
7. Select the Streaming Analytics cluster.
8. Click Flink Dashboard from the services.
   The Flink Dashboard opens in a new window.
9. Click Task Manager on the left side menu.
10. Monitor your Flink application under logs.

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

For Java

```
StreamExecutionEnvironment env =
    StreamExecutionEnvironment.getExecutionEnvironment();
...

env.execute("My Flink job")
```

For Python

```
env = StreamExecutionEnvironment.get_execution_environment()
...

env.execute('My Flink Job')
```

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

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.

For Java

```
DataStream<String> source = env
        .fromSource(
                kafkaSource,
                WatermarkStrategy.noWatermarks(),
                "Kafka Source")
        .uid("kafka-source")
        .map(record -> record.getId()
                + "," + record.getName()
                + "," + record.getDescription())
        .name("To Output String")
        .uid("to-output-string");

FileSink<String> sink = FileSink
        .forRowFormat(
                new Path(params.getRequired(K_HDFS_OUTPUT)),
                new SimpleStringEncoder<String>("UTF-8"))
        .build();
```

```
source.sinkTo(sink)
  .name("FS Sink")
  .uid("fs-sink");
source.print();
```

For Python

```
source = env.from_source(source=kafka_source,
                         watermark_strategy=WatermarkStrategy.no_watermark
s(),
                         source_name='Kafka Source')
source = source.map(lambda record:
                    record.get_id() + ',' +
                    record.get_name() + ',' +
                    record.get_description(),
                    output_type=Types.STRING())
source = source.name('To Output String').uid('to-output-string')

sink = FileSink.for_row_format(
    base_path=output_path,
    encoder=Encoder.simple_string_encoder())

source.sink_to(sink).name('FS Sink').uid('fs-sink')
source.print()
```

Choosing the sources and sinks depends on the purpo se 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.

  For Java

  ```
  dataStream.map(new MapFunction<Integer, Integer>() {
      @Override
      public Integer map(Integer value) throws Exception {
          return 2 * value;
      }
  ```

```
});
```

For Python

```
data_stream.map(lambda value: 2 * value, output_type=Types.I
NT())
```

- FlatMap

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

For Java

```
dataStream.flatMap(new FlatMapFunction<String, String>() {
    @Override
    public void flatMap(String value, Collector<String> out)
        throws Exception {
        for (String word: value.split(" ")) {
            out.collect(word);
        }
    }
});
```

For Python

```
data_stream.flat_map(lambda value: value.split(' '), output_
type=Types.STRING())
```

- Filter

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

For Java

```
dataStream.filter(new FilterFunction<Integer>() {
    @Override
    public boolean filter(Integer value) throws Exception {
        return value != 0;
    }
});
```

For Python

```
data_stream.filter(lambda value: value != 0)
```

- 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

For Java

```
dataStream.keyBy(value -> value.getSomeKey());
dataStream.keyBy(value -> value.f0);
```

For Python

```
data_stream.key_by(lambda value: value.get_some_key())
```

```
data_stream.key_by(lambda value: value[0], key_type=Types
.STRING())
```

- Window

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

  For Java

  ```
  dataStream
      .keyBy(value -> value.f0)
      .window(TumblingEventTimeWindows.of(Time.seconds(5)));
  ```

  For Python

  ```
  data_stream.key_by(lambda value: value[0], key_type=Types.ST
  RING()) \
              .window(TumblingEventTimeWindows.of(Time.minute
  s(1)))
  ```

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 about a streaming window word count that counts worfs from a web socket in five second windows.

For Java

```
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironme
nt;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.streaming.api.windowing.assigners.TumblingProcess
ingTimeWindows;
import org.apache.flink.util.Collector;
```

```
public class WindowWordCount {
    public static void main(String[] args) throws Exception {

        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();

        DataStream<Tuple2<String, Integer>> dataStream = env
            .socketTextStream("localhost", 9999)
            .flatMap(new Splitter())
            .keyBy(value -> value.f0)
            .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
            .sum(1);

        dataStream.print();

        env.execute("Window WordCount");
    }

    public static class Splitter
        implements FlatMapFunction<String, Tuple2<String, Integer>> {

        @Override
        public void flatMap(String sentence,
                            Collector<Tuple2<String, Integer>> out)
            throws Exception {

            for (String word: sentence.split(" ")) {
                out.collect(new Tuple2<String, Integer>(word, 1));
            }
        }
    }
}
```

For Python

```
import logging
import sys

from pyflink.common import Types
from pyflink.datastream import StreamExecutionEnvironment, RuntimeExecut
ionMode

word_count_data = [
    "Lorem ipsum dolor sit amet, consectetur adipiscing elit,",
    " sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.",
    " Eget est lorem ipsum dolor sit amet.",
    " Venenatis lectus magna fringilla urna porttitor rhoncus dolor purus
non.",
    " Scelerisque felis imperdiet proin fermentum leo vel orci porta non.
",
    " A iaculis at erat pellentesque adipiscing commodo elit at.",
    " Bibendum neque egestas congue quisque egestas.",
    " Massa enim nec dui nunc. Tellus mauris a diam maecenas sed enim ut
 sem.",
    " Mauris in aliquam sem fringilla ut morbi tincidunt augue interdum.",
    " Eget sit amet tellus cras adipiscing enim. Amet porttitor eget dolor
 morbi non.",
    " Lacus suspendisse faucibus interdum posuere lorem ipsum.",
    " Diam phasellus vestibulum lorem sed risus ultricies.",
    " Nulla facilisi nullam vehicula ipsum a arcu.",
    " Diam in arcu cursus euismod quis. Tempor commodo ullamcorper a lacus
 vestibulum."]
```

```
def word_count():
    env = StreamExecutionEnvironment.get_execution_environment()
    env.set_runtime_mode(RuntimeExecutionMode.BATCH)


    ds = env.from_collection(word_count_data)

    def split(line):
        yield from line.split()
    ds = ds.flat_map(split) \
        .map(lambda i: (i, 1), output_type=Types.TUPLE([Types.STRING(), T
ypes.INT()])) \
        .key_by(lambda i: i[0]) \
        .reduce(lambda i, j: (i[0], i[1] + j[1]))
    ds.print()
    env.execute()

if __name__ == '__main__':
    logging.basicConfig(stream=sys.stdout, level=logging.INFO, format="%
(message)s")

    word_count()
```

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

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.

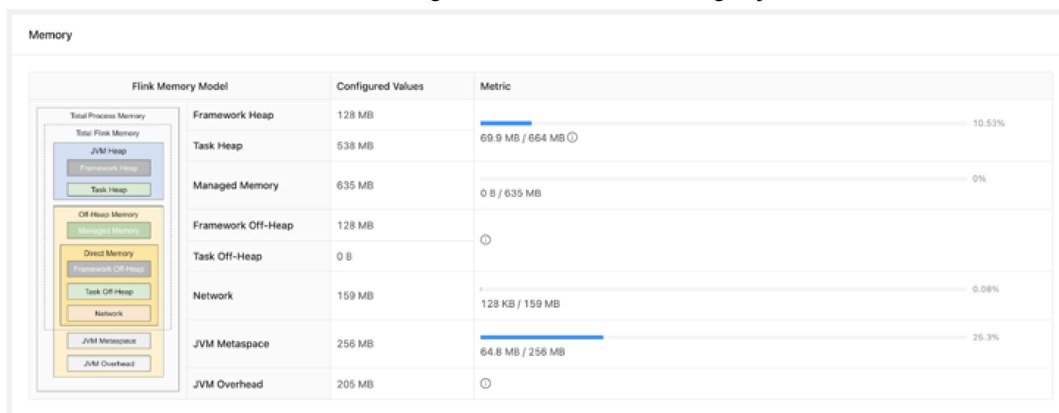
TaskManager Number of Task Slots
taskmanager.numberOfTaskSlots
taskmanager_number_of_task_slots
FLINK-1 (Service-Wide) Undo
4

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.

TaskManager Process Memory Size
taskmanager.memory.process.size
taskmanager_memory_process_size
FLINK-1 (Service-Wide)
2 GiB

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 state backend

You can choose between RocksDB and Hashmap as a state backend for your Flink streaming application. While Hashmap stores data as an object on Java heap, RocksDB can be used to store a larger state that does not fit easily in memory. The RocksDB state backend uses a combination of fast in-memory cache and optimized disk based lookups to manage state.

The following state backend options are available for your Flink application:

RocksDB

By default, RocksDB is configured as the state backend for Flink. The RocksDB state backend holds in-flight data in a RocksDB database that is stored in the TaskManager local data directories and performs asynchronous snapshots. The data is stored as serialized byte arrays. The arrays are defined by the type serializer. This results in key comparisons being byte-wise instead of the hash Code() and equals() Java methods in the Hashmap state backend. The RocksDB state backend is recommended for large state and key/value states, long windows and high-availability setups. The amount of state is limited by the amount of disk space available. This also means that the maximum throughput that can be achieved will be lower as all reads/writes through the RocksDB backend have to go through deserialization and serialization.

Hashmap

The Hashmap state backend holds data internally as objects on the Java heap. The key/value state and window operators hold hash tables that store the values, triggers and so on. The Hashmap state backend is recommended for moderate state and key/value states, long windows and high-availability setups.

You can choose between Hashmap and RocksDB as the state backend based on your performance and scalability requirements. State can be accessed and updated faster with Hashmap on the Java heap, but the size is limited by the available memory in the cluster. Even though RocksDB can scale based on available disk space and is the only state backend to support incremental snapshots, accessing and updating the state requires more resources that can affect the

performance compared to storing the state in-memory. For more information about the State backends in Flink, see the Apache Flink documentation.

You can configure the state backend for your streaming application using Cloudera Manager or you can also specify the state backend in your Flink application.

For Cloudera Manager

1. Open your cluster in Cloduera Manager.
2. Select Flink from the list of services.
3. Click Configuration.
4. Search for state.backend and select HASHMAP or ROCKSDB based on your requirements. ROCKSDB is set by default.

5. Click Save changes.

When configuring the state backend in Cloudera Manager, the configuration serves as a default global setting and is applied for every Flink job. You can override the default set in Cloudera Manager by specifying the state backend in your Flink application.

For Flink application

The per-job state backend can be configured by setting the env.setStateBackend to HashMapStateBackend or EmbeddedRocksDBStateBackend on the StreamExecutionEnvironment of the job as shown in the following example:

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getE
xecutionEnvironment();
env.setStateBackend(new EmbeddedRocksDBStateBackend());
```

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getE
xecutionEnvironment();
env.setStateBackend(new HashMapStateBackend());
```

Note: In case you use Hashmap as state backend, it is recommended to switch to RocksDB when a job fails with the following exception:

```
Caused by: java.lang.OutOfMemoryError: GC overhead limit exceeded
```

Configuring RocksDB

When choosing RocksDB as state backend, you can also adjust how much memory RocksDB should use as a cache to increase lookup performance by setting the memory managed fraction of the TaskManagers.

The default fraction value is 0.4, but with larger cache requirements you need to increase this value together with the total memory size. For more information, see the Set up TaskManager Memory and the Memory Tuning Guide documentations:

For Cloudera Manager

1. Open your cluster in Cloduera Manager.
2. Select Flink from the list of services.
3. Click Configuration.

4. Search for taskmanager.memory.managed.fraction and change the default value as needed.

5. Click Save changes.

For Flink application

You can also configure the Taskmanager Memory when submitting the Flink job in CLI as shown in the following examples:

- Per-job mode:

```
flink run -yD taskmanager.memory.managed.fraction=<fraction_
size> -t
              <job_jar_file>
```

- Session mode:

```
flink-yarn-session -yD
          taskmanager.memory.managed.fraction=<fraction_size>
```

- Application mode:

```
flink run-application -t yarn-application
        -Dtaskmanager.memory.managed.fraction=<fraction_size>
<job_jar_file>
```

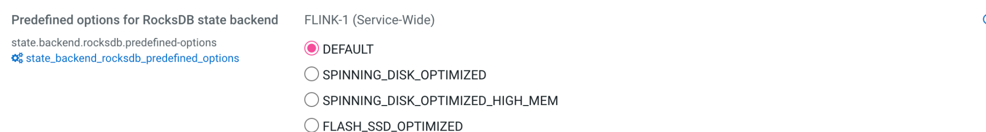
RocksDB also has predefined options that can be changed based on your system configurations:

- DEFAULT: option set by default and applicable for all settings.
- SPINNING_DISK_OPTIMIZED: option for regular spinning disks configured for better performance. The option is configured with the following values:
  - CompactionStyle=LEVEL
  - LevelCompactionDynamicLevelBytes=true
  - MaxBackgroundJobs=4
  - MaxOpenFiles=-1
- SPINNING_DISK_OPTIMIZED_HIGH_MEM: option for regular spinning hard disks configured for better performance, but with higher memory consumption. The option is configured with the following values:
  - BlockCacheSize=256 MBytes
  - BlockSize=128 KBytes
  - FilterPolicy=BloomFilter
  - LevelCompactionDynamicLevelBytes=true
  - MaxBackgroundJobs=4
  - MaxBytesForLevelBase=1 GByte
  - MaxOpenFiles=-1
  - MaxWriteBufferNumber=4
  - MinWriteBufferNumberToMerge=3
  - TargetFileSizeBase=256 MBytes
  - WriteBufferSize=64 MBytes
- FLASH_SSD_OPTIMIZED: option for flash SSDs configured for better performance. The following additional options are set:
  - MaxBackgroundJobs=4
  - MaxOpenFiles=-1

You can set the Rocksdb predefined options in the following ways:

For Cloudera Manager

1. Open your cluster in Cloduera Manager.
2. Select Flink from the list of services.
3. Click Configuration.
4. Search for state.backend.rocksdb.predefined-options and change the default value as needed.



5. Click Save changes.

For Flink application

Update the EmbeddedRocksDBStateBackend for the StreamExecutionEnvironment of the Flink job as shown in the following example:

```
EmbeddedRocksDBStateBackend.setPredefinedOptions(PredefinedOptio
ns.SPINNING_DISK_OPTIMIZED_HIGH_MEM)
```

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.

Enable Checkpoint Compression — ☐ FLINK-1 (Service-Wide)

execution.checkpointing.snapshot-compression

Externalized Checkpoint Retention — FLINK-1 (Service-Wide)

execution.checkpointing.externalized-checkpoint-retention

- (●) RETAIN_ON_CANCELLATION
- ( ) DELETE_ON_CANCELLATION

Checkpointing Interval (milliseconds) — FLINK-1 (Service-Wide)

execution.checkpointing.interval

Max Concurrent Checkpoints — FLINK-1 (Service-Wide)

execution.checkpointing.max-concurrent-checkpoints

1

Min Pause Between Checkpoints (milliseconds) — FLINK-1 (Service-Wide)

execution.checkpointing.min-pause

0

Checkpointing Mode — FLINK-1 (Service-Wide)

execution.checkpointing.mode

- (●) EXACTLY_ONCE
- ( ) AT_LEAST_ONCE

Checkpointing Timeout (milliseconds) — FLINK-1 (Service-Wide)

execution.checkpointing.timeout

60000

Configuring PyFlink applications

Based on the requirements of your PyFlink application, you can optimize the program with different configuration options.

You can configure the Flink application with Python API by setting the options as shown in the following example:

```
from pyflink.common import Configuration
from pyflink.datastream import StreamExecutionEnvironment

config = Configuration()
config.set_integer("python.fn-execution.bundle.size", 1000)
```

```
env = StreamExecutionEnvironment.get_execution_environment(config)
```

The configuration you set in the Flink application affects the Python runtime and how the different processes should be executed. The Flink job related configurations should be set when submiting the Flink job. For the list of Pything configurations, see the Python options section in the Apache Flink documentation. For more information about submitting a Flink job with options, see the Submitting PyFlink jobs section in the Apache Flink documentation.

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 CDP
- High throughput buffered operations
- Customizable data-driven update/delete logic

To use the HBase integration, add one of the following dependencies to your project:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-hbase-1.4</artifactId>
  <version>3.0-csa1.14.0.0</version>
</dependency>
```

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-hbase-2.4</artifactId>
  <version>3.0-csa1.14.0.0</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</artifactId>
  <version>3.2-csa1.14.0.0</version>
</dependency>
```

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

For Java

```
KafkaSource<String> source = KafkaSource.<String>builder()
     .setBootstrapServers("<your_broker_url>")
     .setTopics("<your_source_topic>")
     .setGroupId("<your_group_id>")
     .setStartingOffsets(OffsetsInitializer.earliest())
     .setValueOnlyDeserializer(new SimpleStringSchema())
     .build();
```

For Python

```
source = KafkaSource.builder() \
     .set_bootstrap_servers('<your_broker_url>') \
     .set_topics('<your_source_topic>') \
     .set_group_id('<your_group_id>') \
     .set_starting_offsets(KafkaOffsetsInitializer.earliest()) \
     .set_value_only_deserializer(SimpleStringSchema()) \
     .build()
```

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

For Java

```
DataStream<String> stream = ...
KafkaSink<String> sink = KafkaSink.<String>builder()
        .setBootstrapServers("<your_broker_url>")
        .setRecordSerializer(KafkaRecordSerializationSchema.builder()
            .setTopic("<your_sink_topic>")
            .setValueSerializationSchema(new SimpleStringSchema())
            .build())
        .setDeliveryGuarantee(DeliveryGuarantee.AT_LEAST_ONCE)
        .build();

stream.sinkTo(sink);
```

For Python

```
stream = ...
sink = KafkaSink.builder() \
    .set_bootstrap_servers(brokers) \
    .set_record_serializer(
         KafkaRecordSerializationSchema.builder()
             .set_topic('<your_sink_topic>')
```

```
                .set_value_serialization_schema(SimpleStringSchema())
                .build()) \
        .set_delivery_guarantee(DeliveryGuarantee.AT_LEAST_ONCE) \
        .build()

stream.sink_to(sink)
```

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, JSON and basic types
- 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-cloudera-registry</artifactId>
    <version>${flink.version}</version>
</dependency>
```

The schema registry can be plugged directly into the KafkaSource and KafkaSink using the appropriate schema:

For Avro

```
org.apache.flink.formats.registry.cloudera.avro.ClouderaRegistryAvroKafk
aRecordSerializationSchema
org.apache.flink.formats.registry.cloudera.avro.ClouderaRegistryAvroKafka
DeserializationSchema
```

For JSON

```
org.apache.flink.formats.registry.cloudera.json.ClouderaRegistryJsonKafk
aRecordSerializationSchema
org.apache.flink.formats.registry.cloudera.json.ClouderaRegistryJsonKafka
DeserializationSchema
```

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
- JSON data types

- 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
```

Schema serialization

You can construct the schema serialization with the ClouderaRegistryAvroKafkaRecordSerializationSchema and ClouderaRegistryJsonKafkaRecordSerializationSchema objects for FlinkKafkaProducer based on the required data format. You must set the topic configuration and RegistryAddress parameter in the 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

For Avro

```
        KafkaSerializationSchema<ItemTransaction> schema =
                ClouderaRegistryAvroKafkaRecordSerializationSchema.<Item
Transaction>builder(topic)
                         .setRegistryAddress(registryAddress)
                         .setKey(ItemTransaction::getItemId)
                         .build();
        KafkaSink<ItemTransaction> kafkaSink =
                KafkaSink.<Row>builder()
                        .setKafkaProducerConfig(kafkaProps)
                        .setRecordSerializer(schema)
                        .setDeliveryGuarantee(DeliveryGuarantee.AT_LEAST_O
NCE)
                        .build();
```

For JSON

```
        KafkaSerializationSchema<ItemTransaction> schema =
                ClouderaRegistryJsonKafkaRecordSerializationSchema.<Item
Transaction>builder(topic)
                         .setRegistryAddress(registryAddress)
                         .setKey(ItemTransaction::getItemId)
                         .build();
        KafkaSink<ItemTransaction> kafkaSink =
                KafkaSink.<Row>builder()
                        .setKafkaProducerConfig(kafkaProps)
                        .setRecordSerializer(schema)
                        .setDeliveryGuarantee(DeliveryGuarantee.AT_LEAST_O
NCE)
                        .build();
```

Schema deserialization

You can construct the schema deserialization with the ClouderaRegistryAvroKafkaRecordDeserializationSchema and ClouderaRegistryJsonKafkaRecordDeserializationSchema objects 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 Registry Address parameter in the 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

For Avro

```
        KafkaDeserializationSchema<ItemTransaction> schema =
                        ClouderaRegistryAvroKafkaDeserializationSchema.bui
lder(ItemTransaction.class)
                                .setRegistryAddress(registryAddress)
                                .build();
        KafkaSource<String> transactionSource = KafkaSource.<String>build
er()
                .setBootstrapServers("<your_broker_url>")
                .setTopics(inputTopic)
                .setGroupId(groupId)
                .setStartingOffsets(OffsetsInitializer.earliest())
                .setValueOnlyDeserializer(schema)
                .build();
```

For JSON

```
        KafkaDeserializationSchema<ItemTransaction> schema =
                        ClouderaRegistryJsonKafkaDeserializationSchema.bui
lder(ItemTransaction.class)
```

```
                                .setRegistryAddress(registryAddress)
                                .build();

        KafkaSource<String> transactionSource = KafkaSource.<String>buil
der()
                .setBootstrapServers("<your_broker_url>")
                .setTopics(inputTopic)
                .setGroupId(groupId)
                .setStartingOffsets(OffsetsInitializer.earliest())
                .setValueOnlyDeserializer(schema)
                .build();
```

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

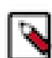
The reporter supports passing arbitrary Kafka producer properties that can be used to modify the behavior, enable security, and so on. Serializer classes should not be modified as it can lead to reporting errors.

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.

Iceberg with Flink

Apache Iceberg is an open, high-performance table format for organizing datasets that can contain petabytes of data. In Cloudera Streaming Analytics, the DataStream API enables you to use Iceberg tables in your Flink jobs.

Iceberg is integrated with Flink as a connector, and you can use the DataStream API to read and write data from a data source to an Iceberg table. Both the V1 and V2 version specifics are supported by the Flink connector. For more information, about the version changes, see the Apache Iceberg documentation.

Table 2: DataStream feature support for the Iceberg and Flink integration

Feature	Flink
Read	Supported
Append	Supported
Overwrite	Supported
Upsert	Technical Preview[1]

When using Iceberg with the DataStream API, you need to create the Flink job that includes referencing the Iceberg table at the TableLoader. The following example shows how to create your Flink job that reads or writes data to or from an Iceberg table using HDFS.

For DataStream read

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.createLocalE
nvironment();

TableLoader tableLoader = TableLoader.fromHadoopTable("hdfs://nn:8020/war
ehouse/path");

DataStream<RowData> stream = FlinkSource.forRowData()
     .env(env)
     .tableLoader(tableLoader)
     .streaming(true)
     .startSnapshotId(3821550127947089987L)
     .build();
// Print all records to stdout.
stream.print();

// Submit and execute this streaming read job.
env.execute("Test Iceberg Streaming Read");
```

For DataStream append

```
StreamExecutionEnvironment env = ...;
DataStream<RowData> input = ... ;
Configuration hadoopConf = new Configuration();
TableLoader tableLoader = TableLoader.fromHadoopTable("hdfs://nn:8020/w
arehouse/path", hadoopConf);

FlinkSink.forRowData(input)
    .tableLoader(tableLoader)
    .append();

env.execute("Test Iceberg DataStream");
```

For DataStream overwrite

```
StreamExecutionEnvironment env = ...;
```

[1] The upsert feature of the Iceberg and Flink integration is in Technical Preview and not ready for production deployment. Cloudera encourages you to explore these features in non-production environments and provide feedback on your experiences through the Cloudera Community Forums.

```
DataStream<RowData> input = ... ;
Configuration hadoopConf = new Configuration();
TableLoader tableLoader = TableLoader.fromHadoopTable("hdfs://nn:8020/wa
rehouse/path", hadoopConf);
FlinkSink.forRowData(input)
   .tableLoader(tableLoader)
   .overwrite(true)
   .append();
env.execute("Test Iceberg DataStream");
```

For DataStream upsert

```
StreamExecutionEnvironment env = ...;
DataStream<RowData> input = ... ;
Configuration hadoopConf = new Configuration();
TableLoader tableLoader = TableLoader.fromHadoopTable("hdfs://nn:8020/wa
rehouse/path", hadoopConf);
FlinkSink.forRowData(input)
   .tableLoader(tableLoader)
   .upsert(true)
   .append();
env.execute("Test Iceberg DataStream");
```

You can add the options to write when configuring the Flink Sink as shown in the following example:

```
FlinkSink.Builder builder = FlinkSink.forRow(dataStream, SimpleDataUtil.FLIN
K_SCHEMA)
   .table(table)
   .tableLoader(tableLoader)
   .set("write-format", "orc")
   .set(FlinkWriteOptions.OVERWRITE_MODE, "true");
```

File systems

You can use file systems in Flink to consume and persistently store data for application results, fault tolerance and data recovery.

The file system used for a particular file is determined by its URI scheme. For example, file:///home/user/te xt.txt refers to a file in the local file system. Flink has built-in support for the file system of the local machine, including any NFS or SAN drives mounted into that local file system. It can be used by default without additional configuration.

For all schemes where Flink cannot find a directly supported file system, it falls back to Hadoop. All Hadoop file systems are automatically available when flink-runtime and the Hadoop libraries are on the classpath. This way, Flink seamlessly supports all of Hadoop file systems implementing the org.apache.hadoop.fs.FileSystem interface, and all Hadoop-compatible file systems (HCFS).

There are also pluggable file systems that are not included by default. CSA contains the following pluggable file systems:

- Amazon S3
- Azure Data Lake Store Gen2
- Azure Blob Storage
- Google Cloud Storage

For more information about how to use the supported file systems, see the Apache Flink documentation.

Using Ozone with Flink

You can use Ozone as a sink with Flink via the implementation of Hadoop file system, but it has some limitations.

Limitations using Ozone File System (OFS) with DataStream connector

When using Ozone as a file system, you must use the OnCheckpointRollingPolicy configuration, which rolls part files on every checkpoint. This configuration is needed because in case part files traverse the checkpoint interval, upon recovery from a failure, the FileSink may use the truncate() method of the file system to discard uncommitted data from the in-progress file. This method is not supported by OFS and Flink will throw an exception.

Limitations using OFS with Table API connector

Because of the limitation present for the DataStream connector, it is required to roll the files on every checkpoint when using the Table API connector. If checkpointing is enabled and a bulk format is used, then this is done automatically. In case of a row format, automatic compaction has to be turned on.

Job lifecycle

Setting up Python for PyFlink

Before you can use Flink with the Python API, it is required to install and configure Python on every relevant node, or create and initialize a Python virtual environment.

For Installing Python

1. Connect to the Flink Gateway node using CLI.

```
ssh root@[***FLINK GATEWAY NODE***]
```

   You are prompted to provide your password to the cluster.
2. Check the version of Python.

```
python --version
```

   If the command fails or the versions are lower than 3.6, install Python.
3. Create a python virtual environment using the following command:

```
conda create --copy -y -n flink_venv python=3.8
```

4. Install PyFlink using the following command:

```
python -m pip install apache-flink==1.19.1
```

5. Install PyFlink on the YARN NameNode as well using the same steps.

For Creating Python venv

1. Connect to the Flink Gateway node using CLI.

```
ssh <[***WORKLOAD USERNAME***]>@[***FLINK MANAGER NODE***]
```

   Provide your workload password when prompted.

2. Create a Python virtual environment using the following command:

```
conda create --copy -y -n flink_venv python=3.8
```

Important: --copy is essential so the created Python environment contains all the required files, not just symlinks.

3. Activate the newly created virtual environment:

```
conda activate flink_venv
```

4. Install PyFlink to the flink_venv virtual environment using the following command:

```
python -m pip install apache-flink==1.19.1
```

5. Create a ZIP archive from the flink_venv virtual environment so it can be deployed with a Flink job:

```
cd path/to/flink_venv && zip -r venv.zip .
```

Note: Zipping the virtual environment in its root folder makes the deployment paths shorter and more straightforward.

When the Python installation is complete, you can submit Flink application that were created using the Python API.

Running a Flink job

After developing your application, you can submit your Flink job in YARN per-job, session or application 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 deployment modes in which you can run your Flink jobs:

- Per-job mode

  Per-job mode means that you run the Flink job in a dedicated 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.
- Application mode

  Regarding resource management, Application mode is much like Per-job mode. That means every deployed application has its dedicated Flink cluster. The key difference is the main() method of the deployed application is executed by the Job Manager rather than the client.

For more information about the deployment modes of Flink, see the Apache Flink documentation.

Note:

Even though it is deprecated for Apache Flink, the per-job deployment mode of Flink is still supported in Cloudera Streaming Analytics.

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-application or 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 CDP Private Cloud Base 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 HOSTNAME***]
   ```

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

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

   Per-job mode

   For Java

   ```
   flink run \
     -d \
     -t yarn-per-job \
     -ynm <job_name_in_yarn> \
     -p <job_parallelism> \
     -ys <slots_per_task_manager> \
     -ytm <memory_per_container_in_mb> \
     <job_jar_file> <job_parameters>
   ```

   For Python

   ```
   flink run \
     -d \
     -t yarn-per-job \
     -ynm <job_name_in_yarn> \
     -p <job_parallelism> \
     -ys <slots_per_task_manager> \
     -ytm <memory_per_container_in_mb> \
     -py <python_script> <job_parameters>
   ```

   For Python venv

   ```
   # Assuming the content of '/tmp/shipfiles': venv.zip, word_c
   ount.py
   flink run -t yarn-per-job \
   -Dyarn.ship-files=/tmp/shipfiles \
   -pyarch /tmp/shipfiles/venv.zip \
   ```

```
-pyclientexec venv.zip/bin/python3 \
-pyexec venv.zip/bin/python3 \
-pyfs /tmp/shipfiles \
-pym word_count
```

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.

For Java

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

For Python

```
flink run \
  -d \
  -t yarn-session \
  -yid <application_id> \
  -p <parallelism> \
  -py <python_script> <job_parameters>
```

Application mode

For Java

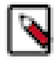
```
flink run-application \
  -d \
  -t yarn-application \
  -p <job_parallelism> \
  -ys <slots_per_task_manager> \
  -ytm <memory_per_container_in_mb> \
  <job_jar_file> <job_parameters>
```

For Python

```
flink run-application \
  -d \
  -t yarn-application \
  -p <job_parallelism> \
  -ys <slots_per_task_manager> \
  -ytm <memory_per_container_in_mb> \
```

```
-py <python_script> <job_parameters>
```

For more examples and PyFlink possibilities, see the Submitting PyFlink Jobs section of the Apache Flink documentation.



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 table summarizes the supported actions of Flink CLI:

Action	Purpose
run	You can use this action to execute jobs. It requires at least the jar containing the job. Flink- or job-related arguments can be passed if necessary.
run-application	You can use this action to execute jobs in application mode. It requires the same parameters as the run action.
info	You can use this action to print an optimized execution graph of the passed job. It requires the jar containing the job.
list	You can use this action to list all running or scheduled jobs.
savepoint	You can use this action to create or dispose savepoints for a given job. You might need to specify a savepoint directory besides the job ID if the state.savepoints.dir was not specified in Cloudera Manager.
checkpoint	You can use this action to create checkpoints for a given job. The checkpoint type (full or incremental) can be specified.
cancel	You can sue this action to cancel running jobs based on their job ID.
stop	You can use this action to stop a running job, but also create a savepoint to start from again. This action combined cancel and save point.

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]

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

Monitoring

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) Go to your cluster in Cloudera Manager.
   b) Select Flink from the list of services.
   c) 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 .
6. Access the YARN Resource Manager user interface to stop the YARN job of the Flink application.
   a) Go back to your cluster in Cloudera Manager.
   b) Select YARN from the list of Services.
   c) Select Applications.

      You are redirected to the Resource Manager page, and the running Flink applications are displayed.
7. Select the application you need to stop.
8. Click Settings.
9. Select Kill application.
10. Navigate to **Flink Dashboard** and review the log level for the running job.
    a) Go back to Cloudera Manager.
    b) Select Flink from the list of services.
    c) Click Flink Dashboard.
11. Select Task Managers from the main menu.
12. Select the previously submitted job.
13. Click Logs.

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.

Global Dashboard
Version: 1.10.0-csa1.2.0.0-SNAPSHOT | Commit: 44ac4bd @ 20.04.2020 @ 16:54:12 CEST | Message: 0
Overview
Jobs
Running Jobs
Completed Jobs

Available Task Slots
2
Total Task Slots 16 | Task Managers 5

Running Jobs
4
Finished 0 | Canceled 4 | Failed 3

Running Job List

Job Name	Start Time	Duration	End Time	Tasks	Status	Cluster
default: select * from KioskIntrusionSFO	2020-04-20 22:44:26	2m 20s	-	1 1	RUNNING	application_1587365700987_0019
SFO Predictive Maintenance	2020-04-20 22:24:13	22m 32s	-	16 16	RUNNING	application_1587365700987_0015
Kiosk Outlier Classifier	2020-04-20 22:22:00	24m 45s	-	16 16	RUNNING	application_1587365700987_0014
Kiosk Alerts	2020-04-20 22:20:03	26m 42s	-	20 20	RUNNING	application_1587365700987_0013

Completed Job List

Job Name	Start Time	Duration	End Time	Tasks	Status	Cluster
TXL Predictive Maintenance	2020-04-20 22:25:19	19m 46s	2020-04-20 22:45:06	12 12	FINISHED	
default: select * from KioskIntrusionSFO	2020-04-20 22:43:07	1m 1s	2020-04-20 22:44:09	1 1	CANCELED	application_1587365700987_0019

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.

Flink SQL and Table API

In Cloudera Streaming Analytics, you can enhance your streaming application with analytical queries using Table API or SQL API. These are integrated in a joint API and can also be embedded into regular DataStream applications. The central concept of the joint API is a Table that serves as the input and output of your streaming data queries.

There are also two planners that translate Table/SQL queries to Flink jobs: the old planner and the Blink planner. Cloudera Streaming Analytics only supports the Blink planner for Table/SQL applications.

Adding the following Maven dependency to the Flink configuration file allows you to use the Table API with the Blink planner.

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-api-java-bridge</artifactId>
  <version>1.19.1-csa1.14.0.0</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-planner-loader-bundle</artifactId>
  <version>1.19.1-csa1.14.0.0</version>
</dependency>
```

SQL programs in Flink follow a structure similar to regular DataStream applications:

1. Create a StreamTableEnvironment with the Blink planner.
2. Register catalogs and tables.
3. Run the queries/updates.
4. Run the StreamTableEnvironment.

You can see an example of the structure here:

```
StreamExecutionEnvironment streamEnv = StreamExecutionEnvironment.getExecuti
onEnvironment();

EnvironmentSettings tableSettings = EnvironmentSettings
      .newInstance()
      .build();

StreamTableEnvironment tableEnv = StreamTableEnvironment
      .create(streamEnv, tableSettings);

tableEnv.executeSql("CREATE TABLE ...");
Table table = tableEnv.sqlQuery("SELECT ... FROM ...");

DataStream<Row> stream = tableEnv.toDataStream(table, Row.class);
stream.print();
```

The Table API exposes different flavors of TableEnvironment to the end users that cover different feature sets. To ensure smooth interaction between other DataStream applications, CSA only supports using StreamTableEnvironme nt.

StreamTableEnvironment wraps a regular StreamExecutionEnvironment. This allows you to seamlessly go from streams to tables and back within the same pipeline.

You can create StreamTableEnvironment with the following code entry:

```
StreamExecutionEnvironment streamEnv = ...
EnvironmentSettings tableSettings = EnvironmentSettings
      .newInstance()
      .build();

StreamTableEnvironment tableEnv = StreamTableEnvironment
      .create(streamEnv, tableSettings);
```

When combining regular DataStream and Table/SQL applications, make sure to always call the .execute command on the StreamTableEnvironment instead of the regular StreamExecutionEnvironment to ensure correct execution.

SQL and Table API supported features

The following SQL and Table API features are supported in Cloudera Streaming Analytics.

<table>
<tr><th>StreamTableEnvironment (Blink Planner)</th><th>Catalogs</th></tr>
<tr><td>DataStream conversions:
Row
Tuple
Temporary views
SQL update and Query operations
Creating and using catalogs
Listing catalogs, tables, databases and functions</td><td>In-memory - Flink default catalog
Hive
Schema Registry
Kudu</td></tr>
</table>

<table>
<tr><th>SQL update statements</th><th>SQL query statements</th></tr>
<tr><td>CREATE TABLE
Computed columns
Watermark definitions
Table connectors
Kafka
Kudu
Hive (through catalog)
Data formats (Kafka)
JSON
Avro
CSV
INSERT INTO
VALUES
SELECT queries</td><td>Basic operations
Scan, Project, Filter
Basic aggregations
Group By
Group By Window
Distinct (only windowed)
Joins
Time-windowed stream-stream join
User defined scalar functions (scalar udf)</td></tr>
</table>

DataStream API interoperability

The DataStream API interoperability offers you new ways to build your Flink streaming application logic as you can convert the DataStreams to Tables, and the Tables back to Datastreams. This means that you can run SQL queries on your DataStreams. You can also convert the result back to other streams, or insert them into one of the supported table sinks.

The following DataStream type conversions are supported:

- DataStream<Row>
- DataStream<TupleX<...>>

Converting DataStreams to Tables

When converting DataStreams to Tables you need to define the StreamTableEnvironment for the conversion. Cloudera recommends creating the tables with names as it is easier to refer to them in SQL. You should also take the processing and event time into consideration as crucial elements of Flink streaming applications.

StreamTableEnvironment is used to convert a DataStream into a Table. You can use the fromDataStream and crea teTemporaryView methods for the conversion. Cloudera recommends that you use the createTemporaryView method as it provides a way to assign a name to the created table. Named tables can be referenced directly in SQL afterwards.

Both of these methods take an optional, but recommended, string parameter to define field name mappings. The string must contain a comma separated list of the desired column names. If the string is not specified, the column names are set to f0, f1, ...fn.

```
DataStream<Tuple2<Integer, String>> stream = ...
```

```
Table table = tableEnv.fromDataStream(stream, "col_1, col_2");

tableEnv.createTemporaryView("MyTableName", stream, "col_1, col_2");
```

You need to take into consideration the event timestamps and watermarks when converting DataStreams.

The processing time attribute must be defined as an additional (logical) column marked with the .proctime property during schema definition.

```
DataStream<Row> stream = ...

Table table = tableEnv.fromDataStream(stream, "col_1, col_2, t
s_col.proctime");
```

For more information on time handling in SQL, see the Apache Flink documentation.

Even time attributes are defined by the .rowtime property during schema definition. This can either replace an existing field or create a new one, but in either case, the field holds the event timestamp of the current record.

```
DataStream<Tuple2<Timestamp, String>> stream = …
stream.assignTimestampsAndWatermarks(...)

Table table = tableEnv.fromDataStream(stream, "event_ts.rowtime,
 col_2");
```

For more information on time handling in SQL, see the Apache Flink documentation.

Converting Tables to DataStreams

Tables are updated dynamically as the result of streaming queries. To convert them into DataStreams, you can either append them or retract them based on the SQL query you have chosen.

The Table changes as new records arrive on the query's input streams. These Tables can be converted back into Data Streams by capturing the change of the query output.

There are two modes to convert a Table into a DataStream:

- Append Mode: This mode can only be used if the dynamic Table is only modified by INSERT changes. For example, it is append-only and previously emitted results are never updated.
- Retract Mode: This mode can always be used. It encodes INSERT and DELETE changes with a boolean flag. True marks inserts, and false marks deletes.

Both toAppendStream and toRetractStream methods take the conversion class or conversion type information as parameters. For the recommended Row conversions, you need to provide the Row.class. For Tuple conversions, you need to provide the Tuple TypeInformation object manually.

```
Table table = tableEnv.sqlQuery("SELECT name, age FROM People");

DataStream<Row> appendStream = tableEnv.toAppendStream(table, Row.class);
DataStream<Tuple2<Boolean, Row>> retractStream = tableEnv.toRetractStream(ta
ble, Row.class);

DataStream<Tuple2<String, Integer>> tupleStream = tableEnv.toAppendStream(
  table,
  new TypeHint<Tuple2<String, Integer>>() {}.getTypeInfo()
);
```

Note: For a detailed discussion about dynamic tables and their properties, see the Apache Flink documentation.

Supported data types

You should review the supported data types before designing your application to have all the information regarding SQL type mappings, timestamp and date types.

Java to SQL type mappings

SQL Type	From DataStream	To DataStream
STRING	String	String
BOOLEAN	boolean/Boolean	boolean/Boolean
BYTES	byte[]	byte[]
DECIMAL(38,18)	BigDecimal	BigDecimal
TINYINT	byte/Byte	byte
SMALLINT	short/Short	short/Short
INT	int/Integer	int/Integer
BIGINT	long/Long	long/Long
FLOAT	float/Float	float/Float
DOUBLE	double/Double	double/Double
MAP	Maps of supported types using MapTypeInfo	Maps of supported types
ARRAY	primitive/object arrays	primitive/object arrays*
ROW	Row	Row

Timestamp and Date types

The Table API supports a wide variety of conversions between java.sql , java.time and SQL types. For smooth operation, it is recommended to use java.sql time classes whenever possible.

<table>
<tr><th></th><th></th><th colspan="2">To DataStream</th></tr>
<tr><th>SQL Type</th><th>From DataStream</th><th>Tuple</th><th>Row</th></tr>
<tr><td rowspan="2">DATE</td><td>java.sql.Date</td><td>java.sql.Date</td><td rowspan="2">java.time.LocalDate</td></tr>
<tr><td>java.time.LocalDate*</td><td>java.time.LocalDate*</td></tr>
<tr><td rowspan="2">TIME(0)</td><td>java.sql.Time</td><td>java.sql.Time</td><td rowspan="2">java.time.LocalTime</td></tr>
<tr><td>java.time.LocalTime*</td><td>java.time.LocalTime*</td></tr>
<tr><td rowspan="2">TIMESTAMP(3)</td><td>java.sql.Timestamp</td><td>java.sql.Timestamp</td><td rowspan="2">java.time.LocalDateTime</td></tr>
<tr><td>java.time.LocalDateTime (no .row time support)*</td><td>java.time.LocalDateTime*</td></tr>
<tr><td>TIMESTAMP_WITH_LOCAL _TIME_ZONE</td><td>java.time.Instant*</td><td>java.time.Instant*</td><td>java.time.Instant</td></tr>
</table>

Note: For information about the limitations regarding DataStream conversion, see the Release Notes.

SQL catalogs for Flink

Cloudera Streaming Analytics supports Hive, Kudu and Schema Registry catalogs to provide metadata for the stored data in a database or other external systems. You can choose the SQL catalogs based on your Flink application design.

For more information about Flink Catalogs, see the Apache Flink documentation.

In-memory catalog

A generic in-memory catalog is enabled by default. However when the in-memory catalog is used, all objects are only available for the lifetime of the session.

Hive catalog

You can add Hive as a catalog in Flink SQL by adding Hive dependency to your project, registering the Hive table in Java and setting it either globally in Cloudera Manager or the custom environment file.

The Hive catalog serves two purposes:

- It is a persistent storage for pure Flink metadata
- It is an interface for reading and writing existing Hive tables

Maven Dependency

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-hive_2.12</artifactId>
  <version>1.19.1-csa1.14.0.0</version>
</dependency>
```

The following example shows how to register and use the Hive catalog from Java:

```
String HIVE = "hive";
String DB = "default";
String HIVE_CONF_DIR = "/etc/hive/conf";
String HIVE_VERSION = "3.1.3000";
HiveCatalog catalog = new HiveCatalog(HIVE, DB, HIVE_CONF_DIR, HIVE_VERSION
);
tableEnv.registerCatalog(HIVE, catalog);
tableEnv.useCatalog(HIVE);
```

Note: According to the latest recommended setup on CDP, the Hive service hosts only the Hive Metastore Server. Make sure that a Gateway role is installed, or the HMS itself is deployed on the node where the Flink commands are submitted.

Note: Do not use Flink to create general purpose batch tables in the Hive metastore that you expect to be used from other SQL engines. While these tables will be visible, Flink uses the additional properties extensively to describe the tables, and thus other systems might not be able to interpret them. Use Hive directly to create these tables instead.

Kudu catalog

You can add Kudu as a catalog in Flink SQL by adding Kudu dependency to your project, registering the Kudu table in Java, and enabling it in the custom environment file.

The Kudu connector comes with a catalog implementation to handle metadata about your Kudu setup and perform table management. By using the Kudu catalog, you can access all the tables already created in Kudu from Flink SQL queries.

The Kudu catalog only allows to create or access existing Kudu tables. Tables using other data sources must be defined in other catalogs, such as in-memory catalog or Hive catalog.

Maven Dependency

```
<dependency>
  <groupId>org.apache.bahir</groupId>
  <artifactId>flink-connector-kudu_2.12</artifactId>
  <version>1.1.0-csa1.14.0.0</version>
</dependency>
```

The following example shows how to register and use the Kudu catalog from Java:

```
String KUDU_MASTERS="host1:port1,host2:port2"
KuduCatalog catalog = new KuduCatalog(KUDU_MASTERS);
tableEnv.registerCatalog("kudu", catalog);
tableEnv.useCatalog("kudu");
tableEnv.listTables();
```

For more information about the Kudu connector and catalog, see the official documentation.

Note: For information about the limitations regarding Kudu catalog, see the Release Notes.

Schema Registry catalog

The Schema Registry catalog allows you to access Kafka topics with registered schemas as Flink SQL tables. You can add Schema Registry as a catalog in Flink SQL by adding the dependency to your project, registering it in Java, and enabling it in the custom environment file.

Each Kafka topic will be mapped to a table with TableSchema that matches the Avro schema.

Maven Dependency

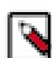
```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-cloudera-registry</artifactId>
  <version>1.0-csa1.14.0.0</version>
</dependency>
```

The following example shows how to register and use the Schema Registry catalog from Java:

```
SchemaRegistryClient client = new SchemaRegistryClient(
 ImmutableMap.of(
  SchemaRegistryClient.Configuration.SCHEMA_REGISTRY_URL.name(),
"http://<your_hostname>:7788/api/v1"
      )
 );
Map<String, String> connectorProps = new Kafka()
  .property(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
                      "<your_hostname>:9092")
  .startFromEarliest()
  .toProperties();

tableEnv.registerCatalog(
   "registry", new ClouderaRegistryCatalog("registry", client, connectorPr
ops)
 );
tableEnv.useCatalog("registry");
```

Note: The SSL related properties have to be set independently for both the SchemaRegistryClient and Kafka.



Note: For information about the limitations regarding Schema Registry, see the Release Notes.

SQL connectors for Flink

In Flink SQL, the connector describes the external system that stores the data of a table. Cloudera Streaming Analytics offers you Kafka and Kudu as SQL connectors. You need to further choose the data formats and table schema based on your connector.

Some systems support different data formats. For example, a table that is stored in Kafka can encode its rows with CSV, JSON, or Avro. The table schema defines the schema of a table that is exposed to SQL queries. It describes how sources and sinks map the data format to the table schema.

Kudu connector

The Kudu connector in Cloudera Streaming Analytics offers compatibility with other supported catalogs, and capability to convert your Kudu tables into DataStreams. You need to add the Kudu dependency to your project, set up the catalog, and you can either use SQL queries or the Kudu catalog directly to create tables.

Kafka connector

Cloudera Streaming Analytics provides Kafka as not only a DataStream connector, but also enables Kafka in the Flink SQL feature. This means if you have designed your streaming application to have Kafka as source and sink, you can retrieve your output data in tables. When using the Kafka connector, you are required to specify one of the supported message formats.

Maven dependency

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kafka</artifactId>
  <version>3.2-csa1.14.0.0</version>
</dependency>
```

For more information about the Kafka connector, see the Apache Flink documentation.

The following example shows a CREATE TABLE statement with Kafka connector:

```
CREATE TABLE source_table (
 id  BIGINT,
 ts  BIGINT,
 itemId STRING,
 quantity  INT
) WITH (
 'connector.type'       = 'kafka',
 'connector.version'    = 'universal',
 'connector.topic'      = 'input_topic',
 'connector.startup-mode' = 'latest-offset',
 'connector.properties.bootstrap.servers' = '<hostname>:<port>',
 'connector.properties.group.id' = 'test',
 'format.type' = 'json'
);
```

On a secured environment where Kerberos and SSL is enabled, the following example can be used:

```
CREATE TABLE source_table (
  c1 STRING
) WITH (
  'connector.type'                                      = 'kafka',
   'connector.version'                                   = 'universal',
   'connector.topic'                                     = 'source_topic',
   'connector.startup-mode'                              = 'earliest-offset',
   'connector.properties.bootstrap.servers'              = '<host>:<port>',
   'connector.properties.group.id'                       = 'test',
   'connector.properties.security.protocol'              = 'SASL_SSL',
   'connector.properties.sasl.kerberos.service.name'     = 'kafka',
   'connector.properties.ssl.truststore.location'        =
 '<absolute_path_to_jks>',
   'format.type'                                         = 'csv'
)
```

Important: The Kerberos related properties should be passed to the run command when submitting your job:

```
flink run -m yarn_cluster -d \
-yD security.kerberos.login.principal=<principal> \
-yD security.kerberos.login.keytab=<local_path_to_keytab> \
-c com.cloudera.streaming.examples.flink.KafkaSecureIT \
flink-sql-tests-1.0-SNAPSHOT.jar
```

Data types for Kafka connector

When reading data using the Kafka table connector, you must specify the format of the incoming messages so that Flink can map incoming data to table columns properly.

JSON format

The JSON format enables you to read and write JSON data. You must add the JSON dependency to your project and define the format type in CREATE table to JSON.

The expected JSON schema will be derived from the table schema by default. Specifying the JSON schema manually is not supported.

Maven dependency

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-json</artifactId>
  <version>1.19.1-csa1.14.0.0</version>
</dependency>
```

For more information about the JSON format, see the Apache Flink documentation.

The following example shows the Kafka connector with JSON data type:

```
CREATE TABLE source_table (
  c1 INT,
  c2 STRING,
  c3 DECIMAL(38,18),
  c4 TIMESTAMP(3),
) WITH (
   'connector.type'      = 'kafka',
 'connector.version'   = 'universal',
 'connector.topic'     = 'input_topic',
 'connector.startup-mode' = 'latest-offset',
 'connector.properties.bootstrap.servers' = '<hostname>:<port>',
 'connector.properties.group.id' = 'test',
 'format.type' = 'json'
);
```

CSV format

The CSV format allows your applications to read data from, and write data to different external sources in CSV. You must add the CSV dependency to your project and define the format type in CREATE table to CSV.

The CSV format will derive format schema from the table schema by default. The format schema can be defined with Flink types also, but this functionality is not supported yet.

Maven dependency

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-csv</artifactId>
  <version>1.19.1-csa1.14.0.0</version>
</dependency>
```

For more information about the JSON format, see the Apache Flink documentation.

The following example shows the Kafka connector with CSV data type:

```
CREATE TABLE source_table (
  c1 INT,
  c2 STRING,
  c3 TIMESTAMP(3)
) WITH (
   'connector.type'                              = 'kafka',
   'connector.version'                           = 'universal',
   'connector.topic'                             = 'sink_topic',
   'connector.properties.bootstrap.servers'     = '<host>:<port>',
   'connector.properties.group.id'               = 'test',
   'format.type'                                 = 'csv'
)
```

Avro format

The Apache Avro format enables you to read and write Avro data. You must add the Avro dependency to your project and define the format type in CREATE table to Avro. You also need to specify the fields of the Avro record within the table.

The format schema can be defined either as a fully qualified class name of an Avro specific record or as an Avro schema string. If a class name is used, the class must be available in the classpath during runtime.

When using the Avro schema string, you must specify the fields of the Avro record. The schema must correspond to the schema of the table in Flink.

For a detailed description of Avro schemas, see the Apache Avro documentation.

Maven dependency

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-avro</artifactId>
  <version>1.19.1-csa1.14.0.0</version>
</dependency>
```

The following example shows how to use an Avro schema string when creating a Kafka connector table. It is specified as a JSON object, having record as type, a name, and the specification of its fields. Note the correspondence of various data types, especially the decimal and array fields.

Example

```
CREATE TABLE source_table(
 string_field STRING,
 long_field    BIGINT,
 decimal_field  DECIMAL(38,18),
     int_arr_field  ARRAY<INT>
) WITH (
 'connector.type'                              = 'kafka',
 'connector.version'                           = 'universal',
 'connector.topic'                             = 'input_topic',
     'connector.properties.group.id'              = 'test',
 'connector.properties.bootstrap.servers'     = '<hostname>:<port>'
 'format.type'                                 = 'avro',
     'format.avro-schema' =
   '{
        "type": "record",
        "name": "test",
        "fields" : [
       {"name": "string_field", "type": "string"},
       {"name": "long_field",   "type": "long"},
       {"name": "decimal_field", "type":
{"type": "bytes",
```

```
  "logicalType": "decimal",
  "precision": 38,
  "scale": 18}},
            {"name": "int_arr_field", "type":
{"type":"array",
  "items":"int"}}
        ]
      }'
)
```

Supported basic data types

Before creating your table with Avro format for the Kafka connector, you should review the supported basic data types that you can use in the application.

Avro schema type	Flink data type
string	STRING
boolean	BOOLEAN
bytes	BYTES
int	INT
long	BIGINT
float	FLOAT
double	DOUBLE

Note: Time, Date, Timestamp are not yet supported.

To specify a field with additional properties, such as the decimal or array fields in the example, the type field must be a nested object which has a type field itself, as well as the needed properties.

An example of a property that must be set this way is a field's logical type. Some types cannot be directly represented by an Avro data type, so they use one of the supported types as an underlying representation. The logical type attribute tells how it should be interpreted. For example, the decimal type – described below – is stored as bytes, while its logical type is decimal.

Decimal type

- Only 38 precision and 18 scale are supported
- Flink data type is DECIMAL(38,18)
- To specify in the Avro schema: {"name": "decimal_field", "type": {"type": "bytes", "logicalType": "decimal", "precision": 38, "scale": 18}}

Array type

Avro allows arrays of supported basic types, except:

- String
- Decimal
- Constructed types (nesting of arrays, rows, maps)

For example, defining an Array of long values:

- In the table definition: arr_field ARRAY<BIGINT>
- Avro schema:

```
{"name":"arr_field, "type": {"type":"array",
    "items":"long"}}}
```

Row type

Flink rows can be specified as records in the Avro schema. Fields must be named both in the SQL of the table definition, as well as in the Avro schema string.

- Field names must match between the table declaration and the Avro schema's record description.
- The two name fields in the Avro schema have the following structure:
  - one on the outside is the name of the field
  - one inside is the type object, pertaining to the record definition
- Decimal fields are not supported within rows.
- Rows can be nested, Arrays are also allowed as fields of the Row.

Example table definition:

```
CREATE TABLE source(row_field ROW<f1 INT,f2 STRING,f3 BOOLEAN>)
WITH (...)
```

Corresponding Avro schema:

```
'format.avro-schema' =
     '{
      "type": "record",
      "name": "test",
      "fields" : [
        {"name": "row_field", "type": {
             "type":"record",
             "name":"row_field",
             "fields": [
             {"name":"f1", "type":"int"},
             {"name":"f2", "type":"string"},
             {"name":"f3", "type":"boolean"}]
  }}]
}'
```

Map type

Only Maps with String keys are supported. The value field can be any type of the supported ones, except decimal.

- In the table definition: map_field MAP<STRING,BIGINT>
- In the Avro schema:

```
{"name": "map_field", "type": {"type":"map",
    "values":"long"}}
```

Nullability

To set a field nullable in the Avro schema, create a union of the field's type and null. A nullable integer field would be defined as: {"name": "int_field", "type": ["int",   "null"]}

Schema Registry formats

You can avoid defining the Avro and JSON schema for Kafka table sources and sinks, when the schema is stored in Cloudera Schema Registry.

Such topics are accessible through automatically generated tables from the read-only registry catalog.

Maven dependency

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-cloudera-registry</artifactId>
  <version>1.0-csa1.14.0.0</version>
```

```
</dependency>
```

If you need to define a table outside the registry catalog, the following example can be used:

```
CREATE TABLE source_table (
   id BIGINT,
   name STRING,
   description STRING
) WITH (
   'connector.type' = 'kafka',
   'connector.version' = 'universal',
   'connector.topic' = 'message',
   'connector.startup-mode' = 'latest-offset',
   'connector.properties.bootstrap.servers' = '<hostname>:<port>',
   'connector.properties.group.id' = 'test',
   'format.type' = 'registry',
   'format.registry.properties.schema.registry.url' = 'http(s)://
<hostname>:<port>/api/v1'
)
```

Cloudera Schema Registry connector for Flink stores the schema version info in the Kafka messages by default. This means that the format.registry.properties.store.schema.version.id.in.header property is set to false by default.

The schema name in the registry is usually the same as the Kafka topic name, but can be overridden by the format.r egistry.schema-name property.

Note: The schema used in this case must be already registered in the Schema Registry, otherwise an error will occur.

SQL Statements in Flink

You can use the CREATE / ALTER / INSERT / DROP statements to modify objects in the chosen catalog. The statements are executed with the sqlUpdate() method of the TableEnvironment.

```
...
StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env, setting
s);
tableEnv.sqlUpdate("CREATE TABLE t1(c1 STRING) WITH (...)");
tableEnv.sqlUpdate("DROP TABLE t1");
...
```

CREATE Statements

You can use CREATE statements to register database, table, and function objects into catalogs. You should add the connector, the name of the table, the schema and the data format to the statement based on your application design. You can further customize your table statement with computed columns to reflect time in a Flink application.

For more information about CREATE statements, see the Apache Flink documentation.

CREATE TABLE

You can connect the Flink Table API and the Flink SQL programs with external systems to read and write streaming tables. The table declaration is similar to a SQL CREATE TABLE statement. The followings can be defined upfront for connecting to an external system:

- Name of the table
- Schema of the table
- Connector
- Data format

You can see an example of the defined parameters:

```
tableEnvironment.sqlUpdate(
    "CREATE TABLE MyTable (\n" +
    "  ...    -- declare table schema \n" +
    ") WITH (\n" +
    "  'connector.type' = '...',  -- connector specific properties\n" +
    "  ...\n" +
    "  'update-mode' = 'append',  -- declare update mode\n" +
    "  'format.type' = '...',     -- format specific properties\n" +
    "  ...\n" +
    ")");
```

Computed column and watermark

A computed column is a virtual column that is generated using the syntax "column_name AS computed_column _expression". Computed columns are commonly used in Flink for defining time attributes in CREATE TABLE statements.

The WATERMARK defines the event time attributes of a table, and allows computed columns to calculate the watermark in the following form: WATERMARK FOR rowtime_column_name AS watermark_strategy_expres sion. The expression return type must be TIMESTAMP(3).

```
CREATE TABLE ItemTransactions (
 transactionId    BIGINT,
 ts    BIGINT,
 itemId    STRING,
 quantity INT,
 event_time AS CAST(from_unixtime(floor(ts/1000)) AS TIMESTAMP(3)),
 WATERMARK FOR event_time AS event_time - INTERVAL '5' SECOND
) WITH (
 'connector.type'      = 'kafka',
 'connector.version'   = 'universal',
 'connector.topic'     = 'transaction.log.1',
 'connector.startup-mode' = 'earliest-offset',
 'connector.properties.bootstrap.servers' = '<hostname>:<port>',
 'connector.properties.group.id' = 'test',
 'format.type' = 'json'
);
```

CREATE DATABASE

```
tableEnvironment.sqlUpdate("CREATE DATABASE sample_database");
tableEnvironment.useDatabase("sample_database");
```

CREATE FUNCTION

```
package com.cloudera.udfs;
public static class HashCode extends ScalarFunction {
  public int eval(String s) {
     return s.hashCode();
  }
}
tableEnvironment.sqlUpdate("CREATE FUNCTION hashcode AS
'com.cloudera.udfs.HashCode");
```

DROP Statements

You can remove the database, table, and function objects from catalogs with DROP statements. You should also include the IF EXIST statement beside the DROP statement to avoid errors due to non existing objects.

Use IF EXISTS statement to avoid errors on non-existing objects.

For more information about DROP statements, see the Apache Flink documentation.

DROP TABLE/DATABASE/FUNCTION

```
tableEnv.sqlUpdate("DROP TABLE|DATABASE|FUNCTION [IF EXISTS] object_name");
```

ALTER Statements

You can use the ALTER statements to modify already registered databases, tables, and function definitions in the chosen catalogs. There are different limitations for databases, tables and functions when altering them.

For more information about ALTER statements, see the Apache Flink documentation.

ALTER DATABASE

Database properties can be changed, but databases cannot be renamed. You can use DROP and CREATE instead of renaming.

```
tableEnv.sqlUpdate("CREATE DATABASE sample_database");
tableEnv.sqlUpdate("ALTER DATABASE sample_database SET ('key1'='value1')");
```

ALTER TABLE

Tables can be renamed and table properties can also be changed.

```
tableEnv.sqlUpdate("CREATE TABLE sample_table(c1 STRING) WITH ('key1'='value
1')");
tableEnv.sqlUpdate("ALTER TABLE sample_table SET ('key1'='value2')");
tableEnv.sqlUpdate("ALTER TABLE sample_table RENAME TO sample_table2");
```

ALTER FUNCTION

New identifiers, which are full classpath for JAVA/SCALA objects, can be assigned to registered functions.

```
tableEnv.sqlUpdate("CREATE FUNCTION myudf AS 'com.example.MyUdf'");
tableEnv.sqlUpdate("ALTER FUNCTION myudf AS 'com.example.MyUdf'");
```

INSERT Statements

Data can be inserted into sink tables in various ways. You can use the INSERT clause to insert the query result into a table, or you can use the VALUE clause to insert data into tables directly from SQL.

You can use the INSERT clause with different source tables, or the VALUES clause. The following examples show how to use INSERT and VALUES at the same time:

```
tableEnv.sqlUpdate("CREATE TABLE source_table (c1 STRING)  WITH (...)");
tableEnv.sqlUpdate("CREATE TABLE sink_table (c1 STRING)  WITH (...)");
tableEnv.sqlUpdate("INSERT INTO source_table VALUES ('foo')");
Table elements = tableEnv.fromDataStream(env.fromElements("bar"));
tableEnv.sqlUpdate("INSERT INTO source_table SELECT f0 from " + elements);
tableEnv.sqlUpdate("INSERT INTO sink_table SELECT * from source_table");
Table table = tableEnv.sqlQuery("SELECT * FROM sink_table");
tableEnv.toAppendStream(table, Row.class).printToErr();
```

For more information about INSERT statements, see the Apache Flink documentation.

Note: The following composite types are supported:

- To construct an array in the SQL insert statement: ARRAY[1,2,3]
- To construct a row: ROW[1,'asd',1.2]
- To construct a map: MAP['key1','val1','key2','val2']
- If you get an error such as Unsupported cast from 'MAP' to 'MAP', (or ROW to ROW, ARRAY to ARRAY), it can be because the inner types are not matching. Make sure that when inserting, you properly cast the values that build up the composite type. Example: INSERT INTO t VALUES (map['www', cast(13 as bigint)])

SQL Queries in Flink

A Table can be used for subsequent SQL and Table API queries, to be converted into a DataSet or DataStream, and to be written to a TableSink. You need to specify the SELECT queries with the sqlQuery() method of the TableEnvironment to return the result of the SELECT query as a Table.

SQL and Table API queries can be seamlessly mixed, and are holistically optimized and translated into a single program.

In order to access a Table in a SQL query, it must be registered in the TableEnvironment. A Table can be registered from the following ways:

- TableSource
- Table
- CREATE TABLE statement
- DataStream
- DataSet

Alternatively, users can also register catalogs in a TableEnvironment to specify the location of the data sources.

The following is an example of SQL query in Java:

```
DataStream<Tuple2<String, Integer>> transactionStream = ...
tEnv.createTemporaryView("Transactions", transactionStream, "account, amoun
t");

Table balance = tEnv.sqlQuery(
  "SELECT account, sum(amount) as balance FROM Transactions GROUP BY accoun
t"
);

DataStream<Tuple2<Boolean, Row>> balanceStream = tEnv.toRetractStream(bal
ance, Row.class);
```

For the detailed documentation and the example code for the different query types, see the Apache Flink documentation.

Note: For information about the supported queries in SQL and Table API, see the Supported Features.

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.

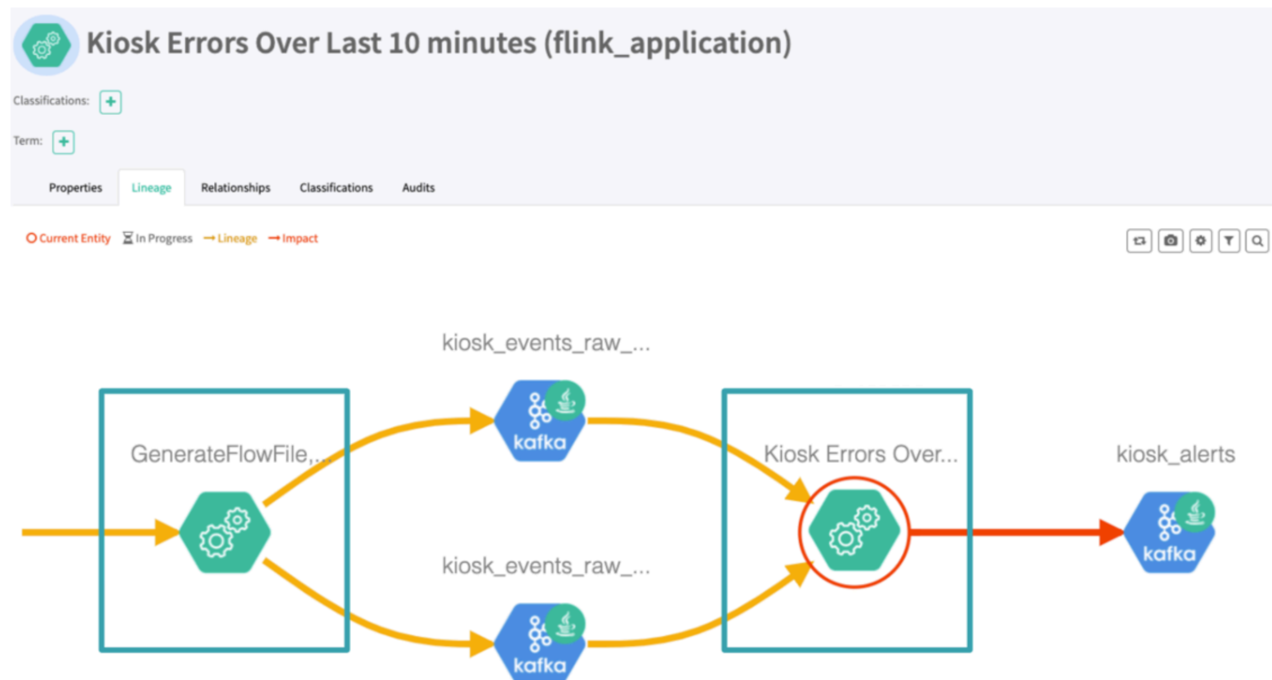
Related Information

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 Information

Create Atlas entity type definitions

Creating Atlas entity type definitions for Flink

Before submitting Flink jobs to collect their metadata, you need to create Atlas entity type definitions for Flink. In the command line, you need to connect to the atlas server and add the predefined type definitions. You also need to enable Atlas for Flink in Cloudera Manager.

About this task

Atlas does not include the metadata source of Flink by default. The administrator must manually upload the entity type definitions to the cluster to be able to start the Flink metadata collection.

Note: The default ports for the Atlas admin server are 31433 and 31000 when TLS is enabled or disabled respectively.

Procedure

1. Upload the designed entity type definitions to the cluster using the Atlas REST API.

```
curl -k -u <atlas_admin>:<atlas_admin_pwd> --location --request POST 'ht
tps://<atlas_server_host>:<atlas_server_port>/api/atlas/v2/types/typedefs'
 \
--header 'Content-Type: application/json' \
--data-raw '{
    "enumDefs": [],
    "structDefs": [],
    "classificationDefs": [],
    "entityDefs": [
        {
            "name": "flink_application",
            "superTypes": [
                "Process"
            ],
            "serviceType": "flink",
            "typeVersion": "1.0",
            "attributeDefs": [
                {
                    "name": "id",
                    "typeName": "string",
                    "cardinality": "SINGLE",
                    "isIndexable": true,
                    "isOptional": false,
                    "isUnique": true
                },
                {
                    "name": "startTime",
                    "typeName": "date",
                    "cardinality": "SINGLE",
                    "isIndexable": false,
                    "isOptional": true,
                    "isUnique": false
                },
                {
                    "name": "endTime",
                    "typeName": "date",
                    "cardinality": "SINGLE",
                    "isIndexable": false,
                    "isOptional": true,
                    "isUnique": false
                },
                {
                    "name": "conf",
                    "typeName": "map<string,string>",
                    "cardinality": "SINGLE",
                    "isIndexable": false,
                    "isOptional": true,
                    "isUnique": false
                }
            ]
        }
    ],
    "relationshipDefs": []
}'
```

2. Log in to Cloudera Manager.
3. Go to Flink>Configuration.
4. Search for 'enable atlas' in the search bar.
5. Enable Atlas Metadata Collection.

Results

The Flink client notifies Atlas about the metadata of the job on successful submission.

Verifying metadata collection

After enabling Atlas metadata collection, newly submitted Flink jobs on the cluster are also submitting their metadata to Atlas. You can verify the metadata collection with messages in the command line by requesting information regarding the Atlas hook.

To verify the metadata collection, you can run the "Streaming WordCount" example from *Running a simple Flink application*.

In the log, the following new lines appear:

```
...
20/05/13 06:28:12 INFO hook.FlinkAtlasHook: Collecting metadata for a new Fl
ink Application: Streaming WordCount
...
20/05/13 06:30:35 INFO hook.AtlasHook: <== Shutdown of Atlas Hook
```

Flink communicates with Atlas through a Kafka topic, by default the one named ATLAS_HOOK.

Reference

Flink Terminology

The list of Flink terminology details the Flink specific terms that are used in the Cloudera Streaming Analytics documentation.

Event

An event is a statement about a change of the state of the domain modelled by the application. Events can be input and/or output of a stream or batch processing application. Events are special types of records.

Function

Functions are implemented by the user and encapsulate the application logic of a Flink program. Most Functions are wrapped by a corresponding Operator.

Flink Application

A Flink application is a Java Application that submits one or multiple Flink Jobs from the main() method. Submitting jobs is usually done by calling execute() on an execution environment.

Flink Job

A Flink Job is the runtime representation of a logical graph (also often called dataflow graph) that is created and submitted by calling execute() in a Flink Application.

Flink JobManager

The JobManager is the orchestrator of a Flink Cluster. It contains three distinct components: Flink Resource Manager, Flink Dispatcher and one Flink JobMaster per running Flink Job.

Logical Graph

A logical graph is a directed graph where the nodes are Operators and the edges define input/output-relationships of the operators and correspond to data streams or data sets. A logical graph is created by submitting jobs from a Flink Application. Logical graphs are also often referred to as dataflow graphs.

Operator

Node of a Logical Graph. An Operator performs a certain operation, which is usually executed by a Function. Sources and Sinks are special Operators for data ingestion and data egress.

Flink Session Cluster

A long-running Flink Cluster which accepts multiple Flink Jobs for execution. The lifetime of this Flink Cluster is not bound to the lifetime of any Flink Job.

State Backend

For stream processing programs, the State Backend of a Flink Job determines how its state is stored on each TaskManager (Java Heap of TaskManager or (embedded) RocksDB) as well as where it is written upon a checkpoint (Java Heap of JobManager or Filesystem).

Task

Node of a Physical Graph. A task is the basic unit of work, which is executed by Flink’s runtime. Tasks encapsulate exactly one parallel instance of an Operator or Operator Chain.

Flink TaskManager

TaskManagers are the worker processes of a Flink Cluster. Tasks are scheduled to TaskManagers for execution. They communicate with each other to exchange data between subsequent Tasks.

Transformation

A Transformation is applied on one or more data streams or data sets and results in one or more output data streams or data sets. A transformation might change a data stream or data set on a per-record basis, but might also only change its partitioning or perform an aggregation. While Operators and Functions are the “physical” parts of Flink’s API, Transformations are only an API concept. Specifically, most transformations are implemented by certain Operators.

For the complete list of Flink terminology, see the Apache documentation.

Cloudera Flink Tutorials

The Cloudera Flink Tutorials walks you through the basic steps to create a Stateless Monitoring, a Stateful Inventory and a Secure application using Flink.

For newcomers, Cloudera recommends starting with the Simple Tutorial.

The Simple Tutorial details the following steps:

- Basic structure of a Flink application
- Logging with Kafka
- Job submission
- Alerting functionality

For a more advanced application, you can use the Stateful Tutorial to get familiar with using state and windowing.

The Stateful Tutorial details the following steps:

- Using state within the application
- Windowing function
- Generating data from Kafka
- Production configuration

The Secure Tutorial serves as a first step to learn everything about the Flink security features within Cloudera.

The Secure Tutorial details the following steps:

- Securing Kafka
- Security parameters in a Flink job
- Kafka Metrics Reporter
- Integration with Schema Registry