Cloudera Streaming Analytics 1.8.0

Application Development

Date published: 2019-12-17
Date modified: 2022-09-28

https://docs.cloudera.com/

Legal Notice

Contents

Flink application structure.....4
Source, operator and sink in DataStream API.....4
Flink application example.....6

Testing and validating Flink applications.....6

Configuring Flink applications.....7
Setting parallelism and max parallelism.....8
Configuring Flink application resources.....8
Configuring RocksDB state backend.....10
Enabling checkpoints for Flink applications.....10

Flink application structure

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

A Flink application consists of the following structural parts:

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

StreamExecutionEnvironment

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

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

Related Information

Flink Project Template

Simple Tutorial: Application logic

Stateful Tutorial: Build a Flink streaming application

Apache Flink document: DataStream API overview

Source, operator and sink in DataStream API

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

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

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

Sources

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

Streaming Analytics in Cloudera supports the following sources:

- HDFS
- Kafka

Operators

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

- Map

  Takes one element and produces one element.

  ```
  dataStream.map()
  ```

- FlatMap

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

  ```
  dataStream.flatMap()
  ```

- Filter

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

  ```
  dataStream.filter()
  ```

- KeyBy

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

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

- Window

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

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

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

Sinks

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

Streaming Analytics in Cloudera supports the following sinks:

- Kafka
- HBase

- Kudu
- HDFS

Related Information

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

Flink application example

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

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

  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecution
Environment();

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

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

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

Testing and validating Flink applications

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

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

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

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

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

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

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

Related Information

Simple Tutorial: Testing the data pipeline

Stateful Tutorial: Test and validate the streaming pipeline

Configuring Flink applications

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

Setting parallelism and max parallelism

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

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

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

Consider the following aspects when setting the max parallelism:

- The number should be large enough to accommodate expected future load increases as this setting cannot be changed without starting from an empty state.
- If P is the selected parallelism for the job, the max parallelism should be divisible by P to get even state distribution.
- Please note that larger max parallelism settings have greater cost on the state backend side, for large scale production jobs benchmarking the size of the state based on the maximum parallelism is useful before changing this parameter.

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

Table 1: Reference values

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

Configuring Flink application resources

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

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

Number of Task Slots

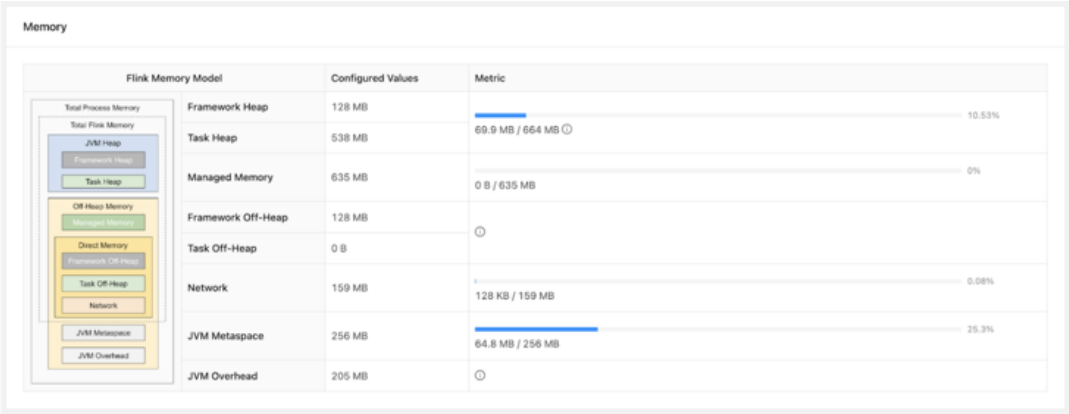
The number of task slots controls how many parallel pipeline/operator instances can be executed in a single TaskManager. Together with the parallelism setting, you can ultimately define how many TaskManagers will be allocated for the job. For example, if you set the job parallelism to 12 and the taskmanager.numberOfTaskSlots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

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

TaskManager Process Memory Size

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

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



Network buffers for throughput and latency

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

Yarn Related Configurations

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

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

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

Reference values for the configurations

Configuration	Parameter	Recommended value
TM container memory	-ytm / taskmanager.heap.size	*TM HEAP + HEAP-CUTOFF*
Managed Memory Fraction	taskmanager.memory.managed.fraction	*0.4 - 0.9*
Max parallelism	pipeline.max-parallelism	*120,720,1260,5040*

Configuration	Parameter	Recommended value
Buffer timeout	execution.buffer-timeout	*1-100*
YARN queue	-yqu	*A QUEUE WITH NO PREEMPTION*
YARN max failed containers	yarn.maximum-failed-containers	*3*NUM_CONTAINERS*
YARN max AM failures	yarn.application-attempts	*3-5*

Configuring RocksDB state backend

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

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

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

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

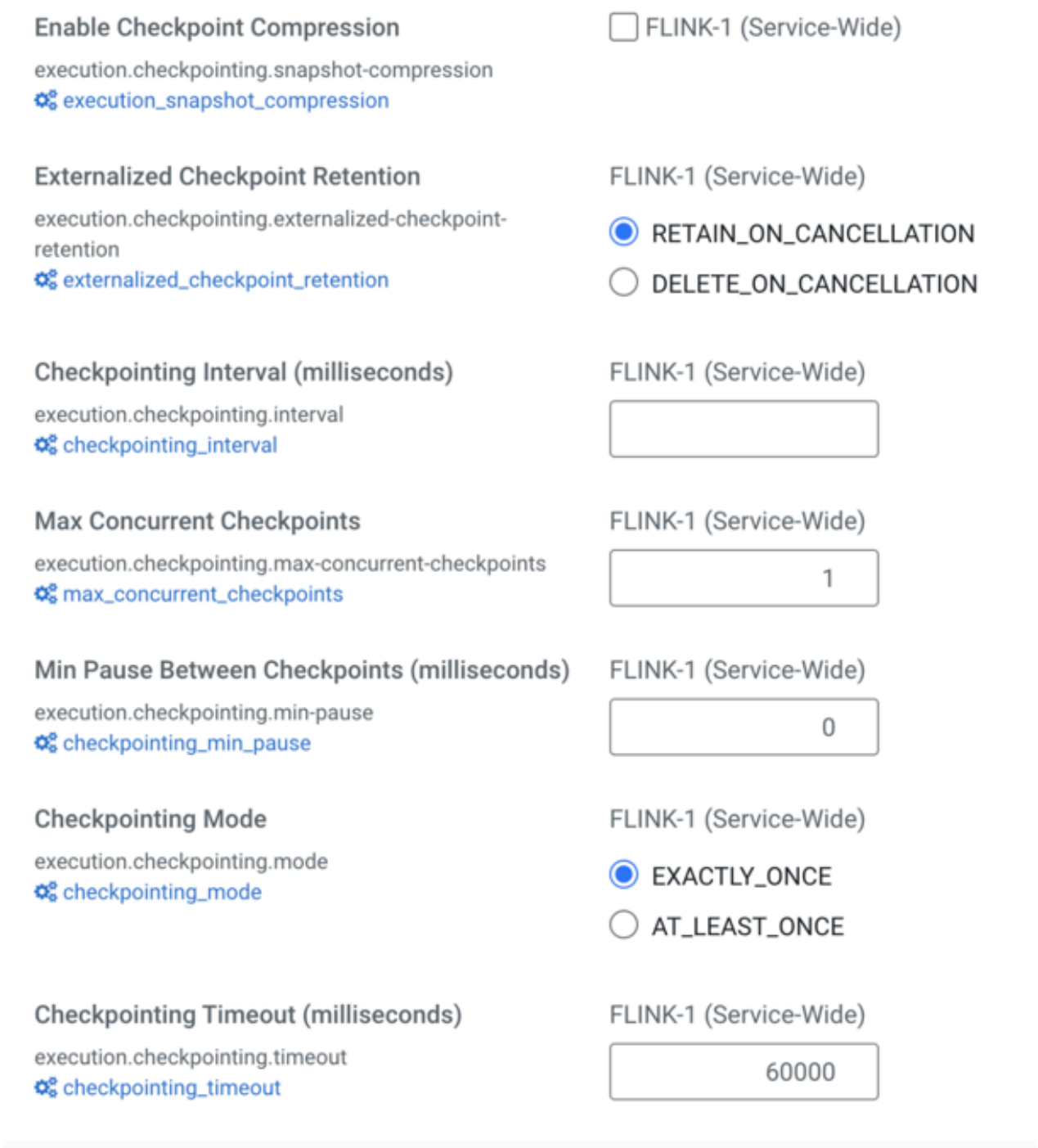
Enabling checkpoints for Flink applications

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

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

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

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


Enable Checkpoint Compression
execution.checkpointing.snapshot-compression
execution_snapshot_compression
FLINK-1 (Service-Wide)
Externalized Checkpoint Retention
execution.checkpointing.externalized-checkpoint-retention
externalized_checkpoint_retention
FLINK-1 (Service-Wide)
RETAIN_ON_CANCELLATION
DELETE_ON_CANCELLATION
Checkpointing Interval (milliseconds)
execution.checkpointing.interval
checkpointing_interval
FLINK-1 (Service-Wide)
Max Concurrent Checkpoints
execution.checkpointing.max-concurrent-checkpoints
max_concurrent_checkpoints
FLINK-1 (Service-Wide)
1
Min Pause Between Checkpoints (milliseconds)
execution.checkpointing.min-pause
checkpointing_min_pause
FLINK-1 (Service-Wide)
0
Checkpointing Mode
execution.checkpointing.mode
checkpointing_mode
FLINK-1 (Service-Wide)
EXACTLY_ONCE
AT_LEAST_ONCE
Checkpointing Timeout (milliseconds)
execution.checkpointing.timeout
checkpointing_timeout
FLINK-1 (Service-Wide)
60000