Cloudera Streaming Analytics

Configuration

Date published: 2019-12-17
Date modified: 2019-12-17

https://docs.cloudera.com/

Legal Notice

Contents

How to configure Apache Flink?

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 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 the initially 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 resources for an application. You can configure the Taskmanager, buffer timeout and high availability for a Flink application to maintain resources in a more efficient way.

YARN automatically kills application containers that use more memory than their allocated limit. To avoid Flink TaskManagers getting killed by YARN use the following calculation to set the size: TM total size = TM Heap + Heap-cutoff. The default heap cutoff is 25% and it is also configurable.

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

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.

Flink on YARN jobs are configured to tolerate a maximum number of failed containers before they terminate. 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 Job Manager 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.

Table 2: Reference values

Configuration	Parameter	Recommended value
TM container memory	-ytm / taskmanager.heap.size	*TM Heap + Heap-cutoff*
Max parallelism	env.setMaxParallelism(num)	*120,720,1260,5040*
Container heap cutoff	containerized.heap-cutoff-ratio	*0.25-0.75*
Buffer timeout	env.setBufferTimeout(millis)	*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*

RocksDB state backend configuration

You can use RocksDB as a state backend when your Flink streaming application requires a larger state. With ClouderaConfigurableOptionsFactory, you are able to configure RocksDB for YARN applications to have a limit on memory usage supported by the Write Buffer Manager.

The default RocksDB state backend configuration available in Flink is not suitable for YARN applications with large state, as it does not effectively limit the native memory usage of the embedded RocksDB database. To effectively enforce a limit on memory usage, support for Write Buffer Manager is introduced in the ClouderaConfigurableOptionsFactory.

ClouderaConfigurableOptionsFactory can be added to the Flink configuration file to get a good initial configuration of the state backend:

```
state.backend: rocksdb
state.backend.rocksdb.options-factory: org.apache.flink.contrib.streaming
.state.ClouderaConfigurableOptionsFactory
```

For the default RocksDB state backend configuration, see the Apache Flink documentation. See more information about the Write Buffer Manager here.

Related Information

Stateful Tutorial: Configure the application for production

Controlling memory usage

When you enable WriteBufferManager for a Flink application, the block-cache size becomes the single most important option in limiting the memory size. You can set it 256 mb as a starting value, and later increase it based on your production experience.

It is configured with the following option:

- state.backend.rocksdb.block.cache-size: *256mb*

The block cache serves as the in-memory cache for both reads and writes for the RocksDB instances. Flink creates an independent RocksDB instance for each stateful (keyed state) operator and subtask, so the minimum RocksDB native memory requirement is num_task_slots * num_stateful_ops * block_cache_size. The block size should be chosen depending on the size of the state and number of keys, but 256 MB is a good starting value.

If the read performance is insufficient, the block-size can be increased to 512 MB, 1 GB or even 2 GB on machines with large memory sizes.

Once you made the calculation, you need to ensure that the YARN container has enough extra memory on top of the TM heap size to cover the native memory requirement. This can be controlled by the following settings:

- containerized-heap-cutoff-min
- containerized-heap-cutoff-ratio

Configuration parameters for RocksDB

Cloudera Streaming Analytics offers you new configurable options factory settings beside the pre-existing one.

In addition to the options factory, the following new options (and their defaults) are introduced:

- state.backend.rocksdb.disk-type : *SPINNING*
- state.backend.rocksdb.writebuffer.manager.size : *blockcache / 4*
- state.backend.rocksdb.writebuffer.manager.enabled : *true*
- state.backend.rocksdb.cache-index-and-filter : *true*
- state.backend.rocksdb.optimize-for-hits : *true*
- state.backend.rocksdb.stats.interval-seconds : *600*

For the list of configuration parameters, see the Apache Flink documentation.

RocksDB based timers

Even if you set RocksDB as a state backend, Flink timers, registered by the window operator or defined by the user, are stored on-heap by default. You can change this by setting the timer service property to RocksDB to reduce the heap consumption.

Timers can be user-defined or registered by the system for window operators. While heap based timers are very quick, they might require too large container sizes.

By setting the state-backend-rocksdb-timer-service-factory configuration property to *ROCKSDB*, Flink stores all timers in the state backend. This has a severe impact on timer performance, but also greatly reduces the heap consumption.