greatlearning

Spark Advanced

Sajan Kedia

Agenda

- Spark Configuration
- Performance Tuning
- Job Scheduling

Spark Configuration

Configuration

Spark provides three locations to configure the system:

- Spark properties control most application parameters and can be set by using a SparkConf object, or through Java system properties.
- Environment variables can be used to set per-machine settings, such as the IP address, through the conf/spark-env.sh script on each node.
- Logging can be configured through log4j.properties.

Spark Properties

- Spark properties control most application settings and are configured separately for each application.
- These properties can be set directly on a SparkConf passed to your SparkContext.
- SparkConf allows you to configure some of the common properties (e.g. master URL and application name), as well as arbitrary key-value pairs through the set() method.
- For example, we could initialize an application with two threads as follows:

```
val conf = new SparkConf().setMaster("local[2]").setAppName("CountingSheep")
val sc = new SparkContext(conf)
```

Dynamically Loading Spark Properties

- In some cases, you may want to avoid hard-coding certain configurations in a SparkConf.
- For instance, if you'd like to run the same application with different masters or different amounts of memory.
- Spark allows you to simply create an empty conf:

```
val sc = new SparkContext(new SparkConf())
```

- Then, you can supply configuration values at runtime:

```
./bin/spark-submit --name "My app" --master local[4] --conf spark.eventLog.enabled=false
  --conf "spark.executor.extraJavaOptions=-XX:+PrintGCDetails -XX:+PrintGCTimeStamps" myApp.jar
```

Reading Spark Properties

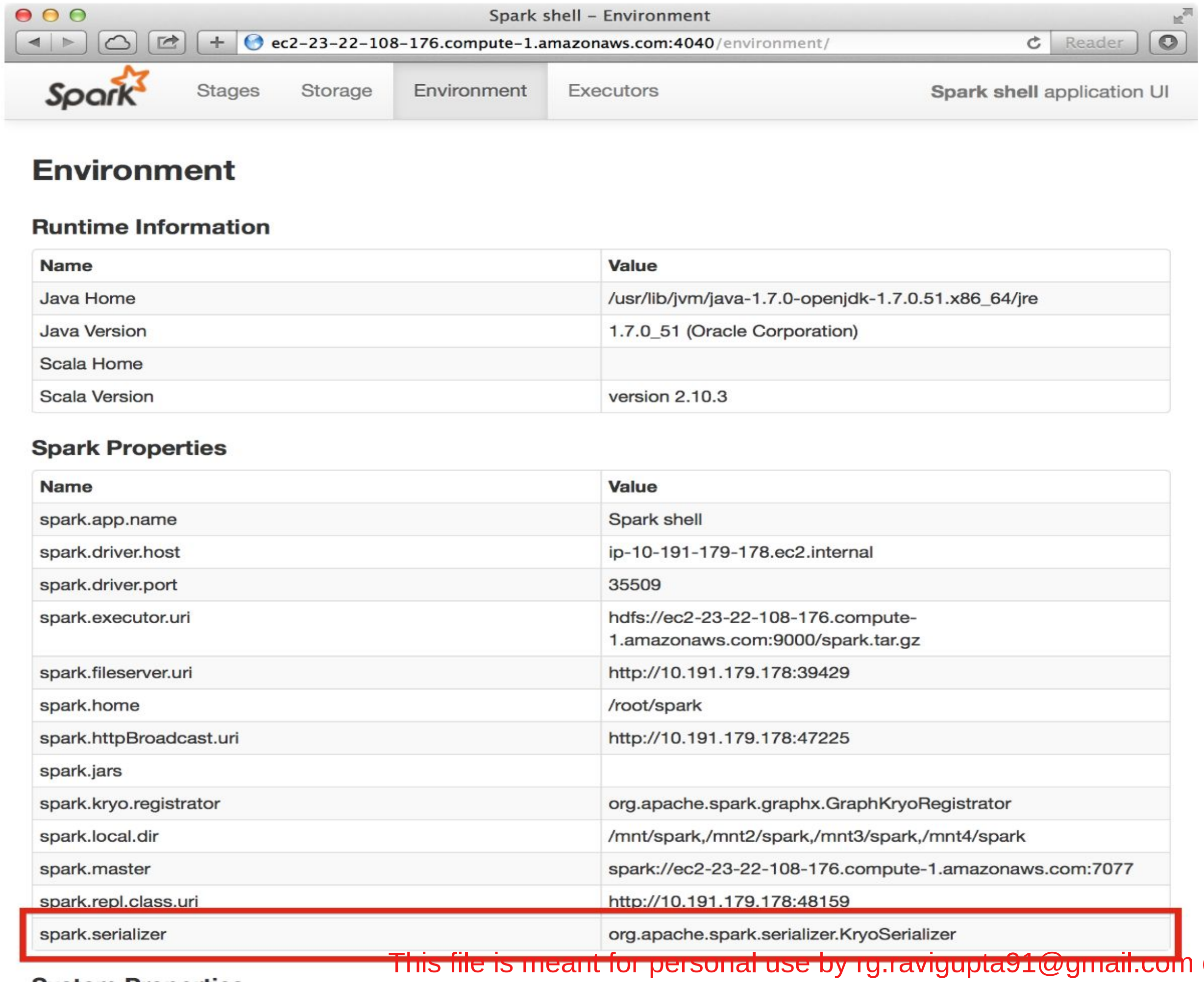
- bin/spark-submit will also read configuration options from conf/spark-defaults.conf, in which each line consists of a key and a value separated by whitespace.

```
spark.master            spark://5.6.7.8:7077
spark.executor.memory   4g
spark.eventLog.enabled  true
spark.serializer        org.apache.spark.serializer.KryoSerializer
```

- Properties set directly on the SparkConf take highest precedence, then flags passed to spark-submit or spark-shell, then options in the spark-defaults.conf file.

Viewing Spark Properties

- The application web UI at `http://<driver>:4040` lists Spark properties in the “Environment” tab.
- This is a useful place to check to make sure that your properties have been set correctly.
- Note that only values explicitly specified through `spark-defaults.conf`, `SparkConf`, or the command line will appear.
- For all other configuration properties, you can assume the default value is used.


Spark shell – Environment
ec2-23-22-108-176.compute-1.amazonaws.com:4040/environment/
Reader
Spark
Stages
Storage
Environment
Executors
Spark shell application UI

Environment

Runtime Information

Name	Value
Java Home	/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.51.x86_64/jre
Java Version	1.7.0_51 (Oracle Corporation)
Scala Home	
Scala Version	version 2.10.3

Spark Properties

Name	Value
spark.app.name	Spark shell
spark.driver.host	ip-10-191-179-178.ec2.internal
spark.driver.port	35509
spark.executor.uri	hdfs://ec2-23-22-108-176.compute-1.amazonaws.com:9000/spark.tar.gz
spark.fileserver.uri	http://10.191.179.178:39429
spark.home	/root/spark
spark.httpBroadcast.uri	http://10.191.179.178:47225
spark.jars	
spark.kryo.registrator	org.apache.spark.graphx.GraphKryoRegistrator
spark.local.dir	/mnt/spark,/mnt2/spark,/mnt3/spark,/mnt4/spark
spark.master	spark://ec2-23-22-108-176.compute-1.amazonaws.com:7077
spark.repl.class.uri	http://10.191.179.178:48159
spark.serializer	org.apache.spark.serializer.KryoSerializer

Resource: http://ampcamp.berkeley.edu/big-data-mini-course/graph-analytics-with-graphx.html

Cluster Managers

1. YARN
2. Mesos
3. Kubernetes
4. Standalone Mode

Environment Variables

Certain Spark settings can be configured through environment variables, which are read from the conf/spark-env.sh script in the directory where Spark is installed.

Environment Variable	**Meaning**
JAVA_HOME	Location where Java is installed
PYSPARK_PYTHON	Python binary executable to use for PySpark in both driver and workers
PYSPARK_DRIVER_PYTHON	Python binary executable to use for PySpark in driver only
SPARKR_DRIVER_R	R binary executable to use for SparkR shell
SPARK_LOCAL_IP	IP address of the machine to bind to.
SPARK_PUBLIC_DNS	Hostname your Spark program will advertise to other machines.

Configuring Logging

- Spark uses log4j for logging.
- You can configure it by adding a `log4j.properties` file in the `conf` directory.
- One way to start is to copy the existing `log4j.properties.template` located there.
- Overriding configuration directory:
    - To specify a different configuration directory other than the default “SPARK_HOME/conf”, you can set SPARK_CONF_DIR

Inheriting Hadoop Cluster Configuration

- If you plan to read and write from HDFS using Spark, there are two Hadoop configuration files that should be included on Spark classpath:
    a. hdfs-site.xml, which provides default behaviors for the HDFS client.
    b. core-site.xml, which sets the default filesystem name.
- common location is /etc/hadoop/conf.
- To make these files visible to Spark, set HADOOP_CONF_DIR in $SPARK_HOME/conf/spark-env.sh to a location containing the configuration files.

Custom Hadoop/Hive Configuration

- If your Spark application is interacting with Hadoop, Hive, there are probably Hadoop/Hive configuration files in Spark's classpath.
- modify hdfs-site.xml, core-site.xml, yarn-site.xml, hive-site.xml
- can be set in $SPARK_HOME/conf/spark-default.conf

```
val conf = new SparkConf().set("spark.hadoop.abc.def","xyz")
```

- you can modify or add configurations at runtime:

```
./bin/spark-submit \   --name "My app" \   --master local[4] \   --conf spark.eventLog.enabled=false \
  --conf "spark.executor.extraJavaOptions=-XX:+PrintGCDetails -XX:+PrintGCTimeStamps" \
  --conf spark.hadoop.abc.def=xyz \   myApp.jar
```

Performance Tuning

Performance Tuning

- Because of the in-memory nature of most Spark computations, Spark can be bottlenecked by any resource in the cluster: CPU, network bandwidth, or memory.
- To avoid this we need tuning such as storing RDDs in serialized form
- There are 2 main topics:
  - data serialization: crucial for good network performance and can also reduce memory use
  - memory tuning

Data Serialization

- Serialization plays an important role in the performance of any distributed application.
- Formats that are slow to serialize objects into, or consume a large number of bytes, will greatly slow down the computation
- It provides two serialization libraries:
  - Java serialization
  - Kryo serialization
- conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")

greatlearning

Data Serialization

- Java serialization:
    - By default, Spark serialize objects using Java ObjectOutputStream framework and can work with any class you create that implements java.io.Serializable.
    - Java serialization is flexible but often quite slow, and leads to large serialized formats for many classes.
- Kryo serialization:
    - Kryo is significantly faster and more compact than Java serialization (often as much as 10x)
    - It does not support all Serializable types and requires you to register the classes you'll use in the program in advance for best performance.

Memory Tuning

- There are three considerations in tuning memory usage:
  - the *amount* of memory used by your objects
  - the *cost* of accessing those objects
  - the overhead of *garbage collection*
- Java objects consume a factor of 2-5x more space than the "raw" data inside their fields. This is due to several reasons:
  - Java object has an "object header", which is about 16 bytes and contains information such as a pointer to its class
  - Java Strings have about 40 bytes of overhead
  - HashMap and LinkedList has "wrapper" object for each entry

Memory Management Overview

- Memory usage in Spark largely falls under one of two categories:

1. Execution: Execution memory refers to that used for computation in shuffles, joins, sorts and aggregations
2. Storage: Storage memory refers to that used for caching and propagating internal data across the cluster

- When no execution memory is used, storage can acquire all the available memory and vice versa
- M: execution and storage share a unified region
- R: Its a subregion within M where cached blocks are never evicted
- R is a threshold within M which is blocked for the storage

Memory Management Overview

- spark.memory.fraction
  - expresses the size of M as a fraction of the (JVM heap space - 300MB) (default 0.6).
  - The rest of the space (40%) is reserved for user data structures, internal metadata in Spark
- spark.memory.storageFraction
  - expresses the size of R as a fraction of M (default 0.5).
  - R is the storage space within M where cached blocks immune to being evicted by execution.

Tuning Data Structures

The first way to reduce memory consumption is to avoid the Java features that add overhead, such as pointer-based data structures and wrapper objects. There are several ways to do this:

1. Design your data structures to prefer arrays of objects, and primitive types, instead of the standard Java or Scala collection classes (e.g. HashMap).
2. Avoid nested structures with a lot of small objects and pointers when possible.
3. Consider using numeric IDs or enumeration objects instead of strings for keys.
4. If you have less than 32 GB of RAM, set the JVM flag -XX:+UseCompressedOops to make pointers be four bytes instead of eight. You can add these options in spark-env.sh.

Serialized RDD Storage

- When your objects are still too large to efficiently store despite this tuning, a much simpler way to reduce memory usage is to store them in serialized form using the serialized Storage Levels in the RDD persistence API
- The only downside of storing data in serialized form is slower access times, due to having to deserialize each object on the fly
- Use Kryo if you want to cache data in serialized form, as it leads to much smaller sizes than Java serialization

Garbage Collection(GC)

- JVM garbage collection can be a problem when you have large “churn” in terms of the RDDs stored by your program.
- When Java needs to evict old objects to make room for new ones, it will need to trace through all your Java objects and find the unused ones.
- The main point to remember here is that *the cost of garbage collection is proportional to the number of Java objects*
- An even better method is to persist objects in serialized form, as described above: now there will be only *one* object (a byte array) per RDD partition.

Garbage Collection(GC) Tuning

- Measuring the Impact of GC:
  - The first step in GC tuning is to collect statistics on how frequently garbage collection occurs and the amount of time spent GC.
  - This can be done by adding -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps to the Java options.
- Advanced GC Tuning:
  - Java Heap space is divided into two regions Young and Old. The Young generation is meant to hold short-lived objects while the Old generation is intended for objects with longer lifetimes.
  - The Young generation is further divided into three regions [Eden, Survivor1, Survivor2].
  - A simplified description of the garbage collection procedure: When Eden is full, a minor GC is run on Eden and objects that are alive from Eden and Survivor1 are copied to Survivor2. The Survivor regions are swapped. If an object is old enough or Survivor2 is full, it is moved to Old. Finally when Old is close to full, a full GC is invoked.

Garbage Collection(GC) Tuning

The goal of GC tuning in Spark is to ensure that only long-lived RDDs are stored in the Old generation. This will help avoid full GCs to collect temporary objects created during task execution. Some steps which may be useful are:

- Check if there are too many garbage collections by collecting GC stats. If a full GC is invoked multiple times for before a task completes, it means that there isn't enough memory available for executing tasks.
- Try the G1GC garbage collector with -XX:+UseG1GC. It can improve performance in some situations where garbage collection is a bottleneck.
- As an example, if your task is reading data from HDFS, the amount of memory used by the task can be estimated using the size of the data block read from HDFS.
- Monitor how the frequency and time taken by garbage collection changes with the new settings.

Level of Parallelism

- Clusters will not be fully utilized unless you set the level of parallelism for each operation high enough
- Spark automatically sets the number of “map” tasks to run on each file according to its size
- Distributed “reduce” operations, such as groupByKey and reduceByKey, it uses the largest parent RDD’s number of partitions.
- You can pass the level of parallelism as a second argument
- set the config property spark.default.parallelism
- In general, we recommend 2-3 tasks per CPU core in your cluster.

Memory Usage of Reduce Tasks

- Sometimes, you will get an OutOfMemoryError not because your RDDs don’t fit in memory, but because the working set of one of your tasks, such as one of the reduce tasks in groupByKey, was too large.
- Spark’s shuffle operations (sortByKey, groupByKey, reduceByKey, join, etc) build a hash table within each task to perform the grouping, which can often be large
- The simplest fix here is to increase the level of parallelism

Broadcasting Large Variables

- Using the broadcast functionality available in SparkContext can greatly reduce the size of each serialized task, and the cost of launching a job over a cluster
- If your tasks use any large object from the driver program inside of them (e.g. a static lookup table) consider turning it into a broadcast variable

Data Locality

- Data locality can have a major impact on the performance of Spark jobs. If data and the code that operates on it are together then computation tends to be fast.
- Data locality is how close data is to the code processing it. several levels of locality based on the data’s current location:
  - PROCESS_LOCAL data is in the same JVM as the running code. This is the best locality possible
  - NODE_LOCAL data is on the same node.
  - NO_PREF data is accessed equally quickly from anywhere and has no locality preference
  - RACK_LOCAL data is on the same rack of servers.
  - ANY data is elsewhere on the network and not in the same rack
- Set spark.locality parameters

Job Scheduling

Scheduling across application

- Each Spark application runs an independent set of executor processes
- within each Spark application, multiple jobs may be running concurrently if they were submitted by different threads
- Each Spark application gets an independent set of executor JVMs that only run tasks and store data for that application
- Multiple users can share the same cluster depending on the cluster manager.
- Static Partitioning of resource:
    - each application is given a maximum amount of resources it can use, and holds onto them for its whole duration

Standalone Mode

- By default cluster will run in FIFO (first in first out) order
- Each application will try to use all available nodes
- Can limit the no of nodes to be used by each application by setting *spark.cores.max* configuration property.
- change the default for applications by *spark.deploy.defaultCores*
- each application's *spark.executor.memory* setting controls its memory use

Mesos Mode

- It handles the workload in distributed environment by dynamic resource sharing
- many physical resources are club into a single virtual resource
- To use static partitioning on Mesos, set the *spark.mesos.coarse* configuration property to true
- optionally set *spark.cores.max* to limit each application’s resource share as in the standalone mode
- set *spark.executor.memory* to control the executor memory

Yarn Mode

- The --num-executors option to the Spark YARN client controls how many executors it will allocate on the cluster
- --executor-memory (spark.executor.memory configuration property) and --executor-cores (spark.executor.cores configuration property) control the resources per executor

Dynamic Resource Allocation

- Spark provides a mechanism to dynamically adjust the resources your application occupies based on the workload.
- This means that your application may give resources back to the cluster if they are no longer used and request them again later when there is demand.
- This feature is particularly useful if multiple applications share resources in your Spark cluster.

Configuration & Setup

- set spark.dynamicAllocation.enabled to true
- set up an external shuffle *service* on each worker node in the same cluster
- set spark.shuffle.service.enabled to true
- The purpose of the external shuffle service is to allow executors to be removed without deleting shuffle files written by them

Resource Allocation Policy

- Spark should relinquish executors when they are no longer used and acquire executors when they are needed.
- Since there is no definitive way to predict whether an executor that is about to be removed will run a task in the near future, or whether a new executor that is about to be added will actually be idle,
- We need a set of heuristics to determine when to remove and request executors. Two ways of it:
    - Request Policy
    - Remove Policy

Request Policy

- A Spark application with dynamic allocation enabled requests additional executors when it has pending tasks waiting to be scheduled.
- Number of executors requested in each round increases exponentially from the previous round.
- For instance, an application will add 1 executor in the first round, and then 2, 4, 8 and so on executors in the subsequent rounds.
- spark.dynamicAllocation.schedulerBacklogTimeout seconds

Remove Policy

- A Spark application removes an executor when it has been idle for more than spark.dynamicAllocation.executorIdleTimeout seconds.
- This condition is mutually exclusive with the request condition, in that an executor should not be idle if there are still pending tasks to be scheduled.

Graceful Decommission of Executors

- Before dynamic allocation, a Spark executor exits either on failure or when the associated application has also exited. In both scenarios, all state associated with the executor is no longer needed and can be safely discarded
- With dynamic allocation, however, the application is still running when an executor is explicitly removed.
- If the application attempts to access state stored in or written by the executor, it will have to perform a recompute the state.
- Thus, Spark needs a mechanism to decommission an executor gracefully by preserving its state before removing it.

Graceful Decommission of Executors

- dynamic allocation may remove an executor before the shuffle completes, in which case the shuffle files written by that executor must be recomputed unnecessarily.
- The solution for preserving shuffle files is to use an external shuffle service.
- This service refers to a long-running process that runs on each node of your cluster independently of your Spark applications and their executors.
- If the service is enabled, Spark executors will fetch shuffle files from the service instead of from each other.
- This means any shuffle state written by an executor may continue to be served beyond the executor’s lifetime.

Scheduling Within an Application

- Spark’s scheduler runs jobs in FIFO fashion.
- Each job is divided into “stages” (e.g. map and reduce phases)
- the first job gets priority on all available resources while its stages have tasks to launch, then the second job gets priority, etc.
- If the jobs at the head of the queue don’t need to use the whole cluster, later jobs can start to run right away, but if the jobs at the head of the queue are large, then later jobs may be delayed significantly.

Fair Scheduler

- it is also possible to configure fair sharing between jobs.
- Under fair sharing, Spark assigns tasks between jobs in a “round robin” fashion, so that all jobs get a roughly equal share of cluster resources.
- This means that short jobs submitted while a long job is running can start receiving resources right away and still get good response times, without waiting for the long job to finish. This mode is best for multi-user settings.
- To enable the fair scheduler, simply set the spark.scheduler.mode property to FAIR when configuring a SparkContext:
- `conf.set("spark.scheduler.mode", "FAIR")`

Fair Scheduler Pools

- The fair scheduler also supports grouping jobs into *pools*, and setting different scheduling options (e.g. weight) for each pool.
- This can be useful to create a “high-priority” pool for more important jobs, for example, or to group the jobs of each user together and give *users* equal shares regardless of how many concurrent jobs they have instead of giving *jobs* equal shares.
- This approach is modeled after the Hadoop Fair Scheduler.
- jobs pools can be set by adding the spark.scheduler.pool

```
// Assuming sc is your SparkContext variable
sc.setLocalProperty("spark.scheduler.pool", "pool1")
```

Default Behaviour of Pools

- By default, each pool gets an equal share of the cluster but inside each pool, jobs run in FIFO order.
- For example, if you create one pool per user, this means that each user will get an equal share of the cluster, and that each user’s queries will run in order instead of later queries taking resources from that user’s earlier ones.

Configuring Pool Properties

- Specific pools properties can also be modified through a configuration file. Each pool supports three properties:

1. schedulingMode: This can be FIFO or Fair scheduler
2. weight: This controls the pool's share of the cluster relative to other pools.
3. minShare: Apart from an overall weight, each pool can be given a minimum shares (as a number of CPU cores) that the administrator would like it to have.

- The pool properties can be set by creating an XML file, named fairscheduler.xml on the classpath, or setting spark.scheduler.allocation.file property in your SparkConf

Configuring Pool Properties

The pool properties can be set by creating an XML file:

```
<?xml version="1.0"?>
<allocations>
  <pool name="production">
    <schedulingMode>FAIR</schedulingMode>
    <weight>1</weight>
    <minShare>2</minShare>
  </pool>
  <pool name="test">
    <schedulingMode>FIFO</schedulingMode>
    <weight>2</weight>
    <minShare>3</minShare>
  </pool>
</allocations>
```

Thanks :)

Reference : https://spark.apache.org/docs/latest/