Chapter 16. Developing Spark Applications

Spark Applications

- are the combination of two things: a Spark cluster and your code
- can build applications using **sbt** or **Apache Maven**, two *Java Virtual Machine (JVM)–based* build tools
- easiest to begin with sbt

SBT

- download, install, and learn about sbt on the sbt website
- to configure an sbt build for our Scala application, we specify a build.sbt file to manage the package information
- Inside the build.sbt file, there are a few key things to include:
  - Project metadata (package name, package versioning information, etc.)
  - Where to resolve dependencies
  - Dependencies needed for your libraryt
- many more options that you can specify beyond the scope of this book

sample Scala built.sbt

```
name := "example"
organization := "com.databricks"
version := "0.1-SNAPSHOT"
scalaVersion := "2.11.8"

// Spark Information
val sparkVersion = "2.2.0"

// allows us to include spark packages
resolvers += "bintray-spark-packages" at  "https://dl.bintray.com/spark-packages/maven/"

resolvers += "Typesafe Simple Repository" at  "http://repo.typesafe.com/typesafe/simple/maven-
releases/"

resolvers += "MavenRepository" at  "https://mvnrepository.com/"

libraryDependencies ++= Seq
(
// spark core
"org.apache.spark" %% "spark-core" % sparkVersion,
 "org.apache.spark" %% "spark-sql" % sparkVersion,
// the rest of the file is omitted for brevity
)
```

- must specify the Scala version as well as the Spark version
- defined a main class that we can run from the command line when we use **spark-submit** to submit it to our cluster for execution
- use **sbt assemble** to build an “uber-jar” or “fat-jar” that contains all of the dependencies in one JAR
  - This can be simple for some deployments but cause complications (especially dependency conflicts)

for others
    - A lighter-weight approach is to run **sbt package**, which will gather all of your dependencies into the target folder but will not package all of them into one big JAR

Running the application

- target folder contains the JAR that we can use as an argument to spark-submit
- After building the Scala package, you end up with something that you can spark-submit on your local machine by using the following code
- this snippet takes advantage of aliasing to create the **$SPARK_HOME** variable; you could replace $SPARK_HOME with the exact directory that contains your downloaded version of Spark

```
$SPARK_HOME/bin/spark-submit \
 --class com.databricks.example.DataFrameExample \
 --master local \
target/scala-2.11/example_2.11-0.1-SNAPSHOT.jar "hello"
```

Writing Python Applications

- similar to writing command-line applications in particular
- Spark doesn't have a build concept, just Python scripts, so to run an application, you simply execute the script against the cluster
- To facilitate code reuse, it is common to package multiple Python files into **egg** or **ZIP** files of Spark code
- To include those files, you can use the **--py-files** argument of spark-submit to add **.py**, **.zip**, or **.egg** files to be distributed with your application
- When it's time to run your code, you **create the equivalent of a "Scala/Java main class" in Python**
- **Specify a certain script as an executable script** that builds the SparkSession
- This is the one that we will pass as the **main argument to spark-submit**

```
from __future__ import print_function
if __name__ == '__main__':
    from pyspark.sql import SparkSession
    spark = SparkSession.builder \
        .master("local") \
        .appName("Word Count") \
        .config("spark.some.config.option", "some-value") \
        .getOrCreate()

    print(spark.range(5000).where("id > 500").selectExpr("sum(id)").collect())
```

- get a SparkSession that you can pass around your application
- pass around this variable at runtime rather than instantiating it within every Python class
- One helpful tip when developing in Python is to use pip to specify PySpark as a dependency
- You can do this by running the command *pip install pyspark* - use it in away that you might use other Python packages

Running the application

- submit it for execution
- executing the same code that we have in the project template
- call spark-submit with that information

```
$SPARK_HOME/bin/spark-submit --master local pyspark_template/main.py
```

Strategic Principles

Input data resilience

- Being resilient to different kinds of input data is something that is quite fundamental to how you write your data pipelines
- The data will change because the business needs will change
- Spark Applications and pipelines should be resilient to at least some degree of change in the input data or otherwise ensure that these failures are handled in a graceful and resilient way
- For the most part this means being smart about writing your tests to handle those edge cases of different inputs and making sure that the pager only goes off when it's something that is truly important

Business logic resilience and evolution

- business logic in your pipelines will likely change as well as the input data
- want to be sure that what you're deducing from the raw data is what you actually think that you're deducing
- This means that you'll need to do robust logical testing with realistic data to ensure that you're actually getting what you want out of it
- One thing to be wary of here is trying to write a bunch of "Spark Unit Tests" that just test Spark's functionality
  - don't want to be doing that; instead, you want to be testing your business logic and ensuring that the complex business pipeline that you set up is actually doing what you think it should be doing

Resilience in output and atomicity

- Assuming that you're prepared for departures in the structure of input data and that your business logic is well tested, you now want to ensure that your output structure is what you expect
- handle output schema resolution
- It's not often that data is simply dumped in some location, never to be read again—most of your Spark pipelines are probably feeding other Spark pipelines
- make certain that your downstream consumers understand the "state" of the data:
  - how frequently it's updated
  - whether the data is "complete" (e.g., there is no late data) or that there won't be any last-minute

corrections to the data

Aforementioned issues are principles that you should be thinking about as you build your data pipelines regardless of whether you're using Spark

Tactical Takeaways

- verify that your business logic is correct by employing proper unit testing
- ensure that you're resilient to changing input data or have structured it so that schema evolution will not become unwieldy in the future
- The decision for how to do this largely falls on you as the developer because it will vary according to your business domain and domain expertise

Managing SparkSessions

- Testing your Spark code using a unit test framework like **JUnit** or **Scala Test** is relatively easy because of Spark's local mode—just create a local mode SparkSession as part of your test harness to run it
- To make this work well, you should try to **perform dependency injection as much as possible** when managing SparkSessions in your code
- **initialize theSparkSession only once** and pass it around to relevant functions and classes at runtime in a way that makes it easy to substitute during testing
- This makes it much easier to test each individual function with a dummy SparkSession in unit tests

Which Spark API to Use?

- Spark offers several choices of APIs, ranging from SQL to DataFrames and Datasets
- each of these can have different impacts for maintainability and testability of your application
- right API depends on your team and its needs:
  - some teams and projects will need the **less strict SQL and DataFrame APIs for speed of development**
  - others will want to use type-safe Datasets or RDDs
- In general, we **recommend documenting and testing the input and output types of each function** regardless of which API you use
- **type-safe API automatically enforces a minimal contract** for your function that makes it easy for other code to build on it
- If your team prefers to use DataFrames or SQL, then **spend some time to document and test what each function returns** and **what types of inputs it accepts** to avoid surprises later (as in any dynamically typed programming language)
- While the lower-level RDD API is also statically typed, we recommend going into it **only if you need low-level features such as partitioning** that are not present in Datasets, which should not be very common;
- the Dataset API allows more performance optimizations and is likely to provide even more of them in the future.

A similar set of considerations applies to which programming language to use for your application:

- each language will provide different benefits
- recommend using statically typed languages like Scala and Java for larger applications or those where you want to be able to drop into low-level code to fully control performance
- Python and R may be significantly better if you need to use some of their other libraries
- Spark code should be testable in the standard unit testing frameworks in every language.

Connecting to Unit Testing Frameworks

- To unit test your code, we recommend using the standard frameworks in your language (e.g.,JUnit or ScalaTest)
- setting up your test harnesses to create and clean up a SparkSession for each test
- Different frameworks offer different mechanisms to do this, such as “before” and“after” methods

Connecting to Data Sources

- should make sure your testing code does not connect to production data sources, so that developers can easily run it in isolation if these data sources change
- One easy way to make this happen is to have all your business logic functions take DataFrames or Datasets as input instead of directly connecting to various sources
  - subsequent code will work the same way no matter what the data source was
- If you are using the structured APIs in Spark, another way to make this happen is **named tables**:
  - **register some dummy datasets** (e.g., *loaded from small text file or from in-memory objects*) as various table names and go from there

The Development Process

- maintain a scratch space, such as an interactive notebook or some equivalent thereof, and then as you build key components and algorithms
- move them to a more permanent location like a library or package
- The notebook experience has simplicity in experimentation
- Tools such as Databricks, that allow you to run notebooks as production applications as well
- When running on your local machine, the **spark-shell** and its various language-specific implementations are probably the best way to develop applications
- For the most part, the shell is for *interactive applications*, whereas spark-submit is for *production applications* on your Spark cluster
- You can use the shell to interactively run Spark, just as we showed you at the beginning of this book. This is the mode with which you will run PySpark, Spark SQL, and Spark R
- In the bin folder, when you download Spark, you will find the various ways of starting these shells
  - Simply run spark-shell(for Scala), spark-sql, pyspark, and sparkR.

After you’ve finished your application and created a package or script to run, spark-submit will become your best friend to submit this job to a cluster

```
/bin/spark-submit \
```

```
  --class <main-class> \
  --master <master-url> \
  --deploy-mode <deploy-mode> \
  --conf <key>=<value> \
  ...
# other options
<application-jar-or-script> \
  [application-arguments]
```

- can always specify whether to run in client or cluster mode when you submit a Spark job with spark-submit.
- you should **almost always favor running in cluster mode (or in client mode on the cluster itself) to reduce latency between the executors and the driver**
- When submitting appliciations, pass a .py file in the place of a .jar, and add Python .zip, .egg,or .py to the search path with --py-files
- To enumerate all available spark-submit options yourself, run spark-submit with --help

all of the available spark-submit options, including those that are particular to some cluster managers:

Table 16-1. Spark submit help text

Parameter	Description
`--master MASTER_URL`	spark://host:port, mesos://host:port, yarn, or local
`--deploy-mode DEPLOY_MODE`	Whether to launch the driver program locally (“client”) or on one of the worker machines inside the cluster (“cluster”) (Default: client)
`--class CLASS_NAME`	Your application’s main class (for Java / Scala apps).
`--name NAME`	A name of your application.
`--jars JARS`	Comma-separated list of local JARs to include on the driver and executor classpaths.
`--packages`	Comma-separated list of Maven coordinates of JARs to include on the driver and executor classpaths. Will search the local Maven repo, then Maven Central and any additional remote repositories given by `--repositories`. The format for the coordinates should be `groupId:artifactId:version`.
`--exclude-packages`	Comma-separated list of `groupId:artifactId`, to exclude while resolving the dependencies provided in `--packages` to avoid dependency conflicts.
`--repositories`	Comma-separated list of additional remote repositories to search for the Maven coordinates given with `--packages`.
`--py-files PY_FILES`	Comma-separated list of *.zip*, *.egg*, or *.py* files to place on the `PYTHONPATH` for Python apps.
`--files FILES`	Comma-separated list of files to be placed in the working directory of each executor.

`--conf PROP=VALUE`	Arbitrary Spark configuration property.
`--properties-file FILE`	Path to a file from which to load extra properties. If not specified, this will look for `conf/spark-defaults.conf`.
`--driver-memory MEM`	Memory for driver (e.g., 1000M, 2G) (Default: 1024M).
`--driver-java-options`	Extra Java options to pass to the driver.
`--driver-library-path`	Extra library path entries to pass to the driver.
`--driver-class-path`	Extra class path entries to pass to the driver. Note that JARs added with `--jars` are automatically included in the classpath.
`--executor-memory MEM`	Memory per executor (e.g., 1000M, 2G) (Default: 1G).
`--proxy-user NAME`	User to impersonate when submitting the application. This argument does not work with `--principal / --keytab`.
`--help, -h`	Show this help message and exit.
`--verbose, -v`	Print additional debug output.
`--version`	Print the version of current Spark.

Deployment-Specific Configurations:

Table 16-2. Deployment Specific Configurations

Cluster Managers	Modes	Conf	Description
Standalone	Cluster	`--driver-cores NUM`	Cores for driver (Default: 1).
Standalone/Mesos	Cluster	`--supervise`	If given, restarts the driver on failure.
Standalone/Mesos	Cluster	`--kill SUBMISSION_ID`	If given, kills the driver specified.
Standalone/Mesos	Cluster	`--status SUBMISSION_ID`	If given, requests the status of the driver specified.
Standalone/Mesos	Either	`--total-executor-cores NUM`	Total cores for all executors.
Standalone/YARN	Either	`--executor-cores NUM1`	Number of cores per executor. (Default: 1 in YARN mode or all available cores on the worker in standalone mode)
YARN	Either	`--driver-`	Number of cores used by the driver, only in cluster mode

		`cores NUM`	(Default: 1).
YARN	Either	`queue QUEUE_NAME`	The YARN queue to submit to (Default: “default”).
YARN	Either	`--num-executors NUM`	Number of executors to launch (Default: 2). If dynamic allocation is enabled, the initial number of executors will be at least `NUM`.
YARN	Either	`--archives ARCHIVES`	Comma-separated list of archives to be extracted into the working directory of each executor.
YARN	Either	`--principal PRINCIPAL`	Principal to be used to log in to KDC, while running on secure HDFS.
YARN	Either	`--keytab KEYTAB`	The full path to the file that contains the keytab for the principal specified above. This keytab will be copied to the node running the Application Master via the Secure Distributed Cache, for renewing the login tickets and the delegation tokens periodically.

If you’re stuck on how to use certain parameters, simply try them first on your local machine and use the **SparkPi** class as the main class:

```
/bin/spark-submit \
  --master spark://207.184.161.138:7077 \
  examples/src/main/python/pi.py \
1000
```

- You run it from the Spark directory and this will allow you to submit a Python application (all in one script) to the standalone cluster manager
- You can change this to run in local mode as well by **setting the master to local or local[*] to run on all the cores on your machine**

Spark provides **three locations** to configure the system:

- Spark properties control most application parameters and can be set by using a **SparkConf** object
- Java system properties
- Hardcoded configuration files

There are several templates that you can use, which you can find in the **/conf directory** available in the root of the Spark home folder

- set these properties as hardcoded variables in your applications
- or by specifying them at runtime

You can **use environment variables to set per-machine settings**, such as the IP address, through the **conf/spark-env.sh script** on each node

- can configure logging through log4j.properties

SparkConf

- manages all of our application configurations

- create one via the import statemen
- SparkConf is immutable for that specific Spark Application

```
from pyspark import SparkConf
conf = SparkConf().setMaster("local[2]").setAppName("DefinitiveGuide")\
  .set("some.conf", "to.some.value")
```

use the SparkConf to configure individual Spark Applications with Spark properties

- Spark properties control how the Spark Application runs and how the cluster is configured.
- configures the local cluster to have two threads and specifies the application name that shows up in the Spark UI
- You can configure these at runtime and through command-line arguments
- helpful when starting a Spark Shell that will automatically include a basic Spark Application for you

```
./bin/spark-submit --name "DefinitiveGuide" --master local[4] ...
```

when setting time duration-based properties, you should use the following format:

- 25ms (milliseconds)
- 5s (seconds)
- 10m or 10min (minutes)
- 3h (hours)
- 5d (days)
- 1y (years

Application Properties:

Table 16-3. Application properties

Property name	Default	Meaning
`spark.app.name`	(none)	The name of your application. This will appear in the UI and in log data.
`spark.driver.cores`	`1`	Number of cores to use for the driver process, only in cluster mode.
`spark.driver.maxResultSize`	`1g`	Limit of total size of serialized results of all partitions for each Spark action (e.g., collect). Should be at least `1M`, or `0` for unlimited. Jobs will be aborted if the total size exceeds this limit. Having a high limit can cause `OutOfMemoryErrors` in the driver (depends on `spark.driver.memory` and memory overhead of objects in JVM). Setting a proper limit can protect the driver from `OutOfMemoryErrors`.
`spark.driver.memory`	`1g`	Amount of memory to use for the driver process, where `SparkContext` is initialized. (e.g. `1g`, `2g`). Note: in client mode, this must not be set through the `SparkConf` directly in your application, because the driver JVM has already started at that point. Instead, set this through the `--driver-memory` command-line option or in your default properties file.
`spark.executor.memory`	`1g`	Amount of memory to use per executor process (e.g., `2g`, `8g`).

`spark.extraListeners`	(none)	A comma-separated list of classes that implement `SparkListener`; when initializing `SparkContext`, instances of these classes will be created and registered with Spark's listener bus. If a class has a single-argument constructor that accepts a `SparkConf`, that constructor will be called; otherwise, a zero-argument constructor will be called. If no valid constructor can be found, the `SparkContext` creation will fail with an exception.
`spark.logConf`	`FALSE`	Logs the effective `SparkConf` as INFO when a `SparkContext` is started.
`spark.master`	(none)	The cluster manager to connect to. See the list of allowed master URLs.
`spark.submit.deployMode`	(none)	The deploy mode of the Spark driver program, either "client" or "cluster," which means to launch driver program locally ("client") or remotely ("cluster") on one of the nodes inside the cluster.
`spark.log.callerContext`	(none)	Application information that will be written into Yarn RM log/HDFS audit log when running on Yarn/HDFS. Its length depends on the Hadoop configuration `hadoop.caller.context.max`.size. It should be concise, and typically can have up to 50 characters.
`spark.driver.supervise`	`FALSE`	If true, restarts the driver automatically if it fails with a non-zero exit status. Only has effect in Spark standalone mode or Mesos cluster deploy mode.

- ensure that you've correctly set these values by checking the application's web UI on port 4040 of the driver on the "Environment" tab
- 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

Runtime Properties

- Although less common, there are times when you might also need to configure the runtime environment of your application
- Refer to the relevant table on the Runtime Environment in the Spark documentation
- These properties allow you to configure:
  - extra class paths and python paths for both drivers and executors
  - Python worker configurations
  - as well as miscellaneous logging properties

Execution Properties

- these configurations are some of the most relevant for you to configure because they give you **finer-grained control on actual execution**
- Refer to the relevant table on Execution Behavior in the Spark documentation
- The most common configurations to change:
  - spark.executor.cores (to control the number of available cores)
  - spark.files.maxPartitionBytes (maximum partition size when reading files)

Configuring Memory Management

- manually manage the memory options to try and optimize your applications
- many of these are not particularly relevant for end users because they involve a lot of legacy concepts or fine-grained controls that were obviated in Spark 2.X because of automatic memory management
- Refer to the relevant table on Memory Management in the Spark documentation

Configuring Shuffle Behavior

- shuffles can be a bottleneck in Spark jobs because of their high communication overhead
- there are a number of low-level configurations for controlling shuffle behavior
- refer to the relevant table on Shuffle Behavior in the Spark documentation.

Environmental Variables

- configure certain Spark settings through environment variables, which are read from the **conf/spark-env.sh** script in the directory where Spark is installed (or c**onf/spark-env.cmd** on Windows)
- In **Standalone** and **Mesos** modes, this file can give machine-specific information such as hostnames
- It is also sourced when running local Spark Applications or submission scripts
- Note that **conf/spark-env.sh does not exist by default** when Spark is installed.
    - can **copy conf/spark-env.sh.template to create it -** be sure to make the copy executable

The following variables can be set in spark-env.sh:

- JAVA_HOME - Location where Java is installed (if it's not on your default PATH)
- PYSPARK_PYTHON - Python binary executable to use for PySpark in both driver and workers (default is python 2.7 if available; otherwise, python) Property **spark.pyspark.python** takes precedence if it is set
- PYSPARK_DRIVER_PYTHON - Python binary executable to use for PySpark in driver only (default is PYSPARK_PYTHON) Property **spark.pyspark.driver.python** takes precedence if it is set
- SPARKR_DRIVER_RR - binary executable to use for SparkR shell (default is R) Property **spark.r.shell.command** takes precedence if it is set
- SPARK_LOCAL_IPIP - address of the machine to which to bind
- SPARK_PUBLIC_DNS - Hostname your Spark program will advertise to other machines

In addition to the variables listed, there are also options for setting up the Spark standalone cluster scripts

- such as number of cores to use on each machine and maximum memory
- because spark-env.sh is a shell script, you can set some of these programmatically
- compute SPARK_LOCAL_IP by looking up the IP of a specific network interface

When running Spark on YARN in cluster mode, you need to set environment variables by using the **spark.yarn.appMasterEnv.[EnvironmentVariableName] property** in your **conf/spark-defaults.conf** file

- Environment variables that are set in spark-env.sh will not be reflected in the YARN Application Master process in cluster mode

Job Scheduling Within an Application

- Within a given Spark Application, **multiple parallel jobs can run simultaneously if they were submitted from separate threads**
- *By job, in this section, we mean a Spark action and any tasks that need to run to evaluate that action*
- Spark's scheduler is fully thread-safe and supports this use case to enable applications that serve multiple requests (e.g., queries for multiple users)
- By default, Spark's scheduler runs jobs in FIFO fashion
- If the jobs at the head of the queue don't need to use the entire cluster, later jobs can begin to run right away, but if the jobs at the head of the queue are large, later jobs might be delayed significantly
- 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 begin receiving resources right away and still achieve good response times without waiting for the long job to finish
    - This mode is best for multi user settings
    - To enable the fair scheduler, set the **spark.scheduler.mode** property to **FAIR** when configuring a **SparkContext**
    - The fair scheduler also supports grouping jobs into pools, and setting different scheduling options, or weights, for each pool
    - This can be useful to create a high-priority pool for more important jobs 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
    - Without any intervention, newly submitted jobs go into a default pool, but jobs pools can be set by adding the spark.scheduler.pool local property to the SparkContext in the thread that's submitting them.
    - This is done as follows (assuming **sc** is your **SparkContext**:

```
sc.setLocalProperty("spark.scheduler.pool", "pool1")
```

    - After setting this local property, all jobs submitted within this thread will use this pool name
    - The setting is per-thread to make it easy to have a thread run multiple jobs on behalf of the same user
    - If you'd like to clear the pool that a thread is associated with, set it to null.