Hadoop, Hive & Spark Tutorial

1 Introduction

This tutorial will cover the basic principles of Hadoop MapReduce, Apache Hive and Apache Spark for the processing of structured datasets.

For more information about the systems you are referred to the corresponding documentation pages:

- Apache Hadoop 2.6: `http://hadoop.apache.org/docs/r2.6.0/`
- Apache Hive 1.2: `https://hive.apache.org/`
- Spark 1.4: `https://spark.apache.org/docs/1.4.0/`

1.1 Virtual Machine and First Steps

A Virtual Machine (VM) is provided with all the software already installed and configured. Within the machine the source code and jars are also provided, along with the used datasets.

As a first step you have to download the VM and open it with VirtualBox. The VM holds an Ubuntu 14.04 LTS operative system with the essential packages needed for the execution of the examples. The VM is configured with 4GB RAM by default, but you can modify the corresponding settings to fit best with your host system. The VM is linked to 2 hard disks, one containing the OS and the other configured to be used as swap if necessary.

The VM's name is `hadoopvm`. Be careful to change this name, as it is used in the configuration files of the provided systems. The user created for the tests is `hadoop` with password `hadoop`. You may also connect to the VM with `ssh` on port 2222.

Hadoop 2.6, Apache Hive 1.2 and Apache Spark 1.4 are installed in the user account under `bin` directory. You can use the corresponding start/stop scripts (e.g., `start-dfs.sh`, `start-yarn.sh`) but a pair of scripts are provided to start and stop all the services. Their usage is as following:

```
# start_servers.sh
...
Use Hadoop, Hive or Spark
...
# stop_servers.sh
```

Hadoop's distributed filesystem (HDFS) is already formatted. If you want to reformat it from scratch you may use the following command prior to starting the servers:

```
# hadoop namenode -format
```

1.2 Examples and Datasets

The examples are included in the `$HOME/examples` directory. There you can find the sources of all the examples of the tutorial as well as the jar used for the MapReduce examples.

The datasets are included in the `$HOME/datasets` directory. Two scripts are given to extract, transform and load the datasets into HDFS.

1.3 NASA web logs

This dataset contains 205MB of uncompressed logs extracted from the HTTP requests submitted to the NASA Kennedy Space Center web server in July 1995. This is an example of an HTTP request log:

```
199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] "GET /hist/apollo/ HTTP/1.0" 200 6245
```

The line contains the requesting host, the date, the HTTP request (HTTP methods, url and optionally the HTTP version), the HTTP return code and the number of returned bytes. More information about the dataset can be found in `http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html`.

In order to load the dataset into HDFS, move to the datasets directory (`/home/hadoop/datasets`) and execute:

```
# ./load_nasa_access_log.sh
```

As a result, the uncompressed file will be uploaded to `/data/logs` HDFS directory. You may check this with the following command:

```
# hadoop fs -ls -R /data
```

1.4 MovieLens dataset

The second dataset corresponds to the MovieLens 1M dataset, which contains 1 million ratings from 6000 users on 4000 movies. The description of the dataset is provided as part of the zip file and is also available at `http://files.grouplens.org/datasets/movielens/ml-1m-README.txt`.

The dataset is divided into three main files:

- **Users:** Each line contains information about a user: its identifier, gender, age, occupation and zip-code.
- **Movies:** Each line contains information about a movie: identifier, title (as it appears in IMDB, including release year) and a list of genres.
- **Ratings:** Each line contains a user rating with the following information: user identifier, movie identifier, rating (5-star scale) and timestamp.

Originally the fields within the files are separated by two colons (`::`), but in order to facilitate importing in Apache Hive, those are replaced by asterisks (`*`) before loading the data into HDFS. Genres are separated by the vertical bar (`|`).

Additionally, the movies data is also formatted in `JSON` format, as this format will be used in Apache Spark.

In order to load the dataset into HDFS, move to the datasets directory and execute:

```
# ./load_movieles.sh
```

2 Hadoop MapReduce

In the first part of the tutorial we are going to use Apache Hadoop MapReduce to process the information contained in the NASA logs datasets.

Before executing the examples, make sure you have successfully started HDFS and YARN servers.

2.1 Introduction

MapReduce programs basically consist in two user defined functions called respectively map and reduce. These functions are applied to records of key-value pairs and are executed in parallel. They also produce key-value pairs, the only restriction being that output key-value pairs of the map functions are of the same type as input key-value pairs of the reduce function[1].

Between the execution of both functions, an intermediate phase, called shuffle, is executed by the framework, in which intermediate records are grouped and sorted by key.

Hadoop MapReduce provides two different APIs to express mapreduce jobs, called `mapred` and `mapreduce`. In this tutorial, the first API is used.

2.2 Basic Examples

Methods Count. In the first example we are going to count how many requests have been submitted for each HTTP method. This example is very close to the typical `WordCount` example given in many MapReduce tutorials. The source code of this example can be found in the class `fr.inria.zenith.hadoop.tutorial.logs.MethodCount`.

MapReduce allows the user to specify the `InputFormat` in charge of reading the files, and produce the input key-value pairs. Throughout this tutorial, we will use `TextInputFormat`, which generates a record for each line, where the key is the offset of the beginning of the line and the value the content of the line.

The map function for this example just extracts the HTTP method from the line and emits a key-value pair containing the method and `ONE`, meaning that this method have been found one time. Here is the main part of the code of the map function:

```
private final static IntWritable ONE = new IntWritable(1);
private Text method = new Text();

public void map(LongWritable key, Text value,
                OutputCollector<Text, IntWritable> output, Reporter reporter)
                throws IOException {

    String lineStr = value.toString();
    int startRequest = lineStr.indexOf('"');
    int endRequest = lineStr.indexOf('"', startRequest + 1);

    String httpRequest = lineStr.substring(startRequest + 1, endRequest);
    method.set(httpRequest.split(" ")[0]);

    output.collect(method, ONE);

}
```

Input types (`LongWritable` and `Text`) are given by the `InputFormat`. The intermediate key value pairs are sent to the `OutputCollector` and their types defined in the class. Note that in

[1]Actually, the reduce function receives a key and a set of values of the same type as the map output

the source code provided in the VM, additional sanity checks are included to deal with incorrect inputs.

The reduce function will receive for each method, a collection with the counts of appearances. It has just to sum them and emit the sum.

```
public void reduce(Text code, Iterator<IntWritable> values,
                   OutputCollector<Text, IntWritable> output, Reporter reporter)
                   throws IOException {
    int sum = 0;
    while (values.hasNext()) {
        sum += values.next().get();
    }
    output.collect(code, new IntWritable(sum));
}
```

Finally, a MapReduce job has to be created and launched. The job is represented by a `JobConf`. This object is used to specify all the configuration of the job, including map and reduce classes, input and output formats, etc.

Map and reduce functions have been included respectively into two pair of classes called `Map` and `Reduce`. A combiner, which is in charge of performing partial aggregates, is also specified (in this case, the same class as the reduce is used). Input and output paths are specified through `FileInputFormat` and `FileOutputFormat` classes.

```
public int run(String[] args) throws Exception {
    JobConf conf = new JobConf(getConf(), MethodsCount.class);
    conf.setJobName("methods_count");

    String inputPath = args[0];
    String outputPath = args[1];

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);

    FileInputFormat.setInputPaths(conf, new Path(inputPath));
    FileOutputFormat.setOutputPath(conf, new Path(outputPath));

    JobClient.runJob(conf);
    return 0;
}
```

To execute this program you can use the following command:

```
# hadoop jar HadoopTutorial.jar
        fr.inria.zenith.hadoop.tutorial.logs.MethodsCount
        /data/logs /out/logs/methods_count
```

The command contains the jar with the program, the class to be executed, and the input and output paths specified as parameters.

Web Page Popularity. The next example obtains for each web page, the number of requests received during the month. The code is very similar to the previous one and is included in the class `WebPagePopularity` in the same package as before. The only difference is the addition of a filter, as we only consider as web pages, those resources finishing by `.html` or `/`. The addition of filters in MapReduce is done programmatically and implies just to emit or not a given key-value pair. This is represented by this block of code:

```
String url = httpRequest.split(" ")[1];
if (url.endsWith(".html") || url.endsWith("/")) {
    pageURL.set(url);
    output.collect(pageURL, ONE);
}
```

`Parameters and Counters`

Image Traffic. In the last example we want to obtain the total amount of traffic (returned bytes) generated by the images (resources ending with `.gif`). Only the images generating more that a specific amount of traffic (specified as a parameter) are returned.

This example contains several additional elements. Firstly, in this case, all the fields of the logs are obtained with a regular expression. Whenever the expression does not find any match we increment a counter. We proceed the same way for entries with no information about the returned bytes. When executing the program, those counters will be appended to the information given to the user.

```
enum Counters {MALFORMED_LOGS, NO_BYTES_LOGS}

...

if (m.find()) {
   ...
} else {
    reporter.incrCounter(Counters.MALFORMED_LOGS, 1);
}
```

The second novelty of this example is the inclusion of a user defined parameter. In this case, the parameter filters the pages generating less than a given amount of traffic, specified in MBs. Parameters are passed in the job configuration like this:

```
conf.setInt("img_traffic.min_traffic", Integer.parseInt(args[2]));
```

Parameters are read in the map or reduce class by using the `configure` method.

```
@Override
public void configure(JobConf job) {
    minTraffic = job.getInt("img_traffic.min_traffic", 0) * (1024 * 1024);
}
```

To specify the minimum traffic when executing the job, we add it to the command line arguments:

```
# hadoop jar HadoopTutorial.jar
         fr.inria.zenith.hadoop.tutorial.logs.ImageTraffic
         /data/logs /out/logs/image_traffic 2
```

Additional queries Note that the result of the last query is sorted by image url. It would be more interesting to sort the pages by descending order of traffic. We can not do this in a single MapReduce job. In order to do that we need to execute an additional job. In the method `main()`, we should create an additional `JobConf` to be executed after the first job and in which the input path is the output path of the first job. The map and reduce functions to sort the output would be very simple.

3 Apache Hive

In this part of the tutorial we are going to use Apache Hive to execute the same queries as in the previous example and observe that it is much easier to write them in the new framework. Additionally we are going to use the MovieLens dataset to execute other type of queries.

Hive provides a interactive shell that we are going to use to execute the queries. To enter the shell, just type:

```
# hive
```

Alternatively, you can create a file for each query and execute it independently with the option `hive -f <filename>`.

3.1 Introduction

Hive is a data warehouse solution to be used on top of Hadoop. It allows to access the files in HDFS the same way as MapReduce and query them using an SQL-like query language, called HiveQL. The queries are transformed into MapReduce jobs and executed in the Hadoop framework.

The metadata containing the information about the databases and schemas is stored in a relational database called the metastore. You can plug any relational database providing JDBC. In this tutorial we are using the simpler embedded derby database provided with Hive.

All the queries used in this part of the tutorial can be found in the directory `examples/src/hive`.

3.2 Nasa Logs

3.2.1 Databases and Tables

As in any traditional database, in Hive we can create and explore databases and tables. We are going to create a database for the Nasa logs dataset.

```
create database if not exists nasa_logs;
```

We can get information about this database by using the command `describe`.

```
describe database nasa_logs;
```

As a result of this command we can learn about the location of this database, which should be the HDFS directory `hive/warehouse/nasa_logs.db`, as it is the default location indicated in the configuration files. Of course this default location can be changed. It is also possible to specify a particular location for a given database.

Now we move to the database by typing:

```
use nasa_logs;
```

The next step is to create a table containing the information of the logs. We will specify the fields and types as in standard SQL. Additionally we are going to tell Hive how to parse the input files. In this case, we use a regular expression as in the `ImageTraffic` example.

```
create external table logs (
  host string,
  req_time string,
  req_code string,
  req_url string,
  rep_code string,
  rep_bytes string
)
row format serde 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
with serdeproperties (
  "input.regex" = '([^ ]*) - - \\[(.*)\\] "([^ ]*) ([^ ]*).*" ([^ ]*) ([^ ]*)'
)
stored as textfile
location "/data/logs";
```

Different row formats and file types can be specified. The default row format for `textfile` is based on separators. In this case we use a *SerDe* provided with Hive and based on regular expressions. You could create your own *SerDe* and use it to parse the tables in a personalized way.

We have defined the table as external and specified the location. This keeps the table in that location and makes Hive assume that it does not own the data, meaning that the data is not removed when the database is dropped.

Now that the table is created we can query it directly using SQL commands, for instance:

```
select * from logs limit 10;
```

Note: if we want the output to include the headers of the columns we can set the corresponding property:

```
set hive.cli.print.header=true;
```

3.2.2 Queries

At this point we are ready to execute the same examples we have implemented previously with MapReduce.

Methods Count. We use a simple SQL expression with a group by:

```
select req_code, count(*) as num_requests
from logs
group by req_code;
```

Note that this query generates a MapReduce job as you can see from the output logs.

Web Page Popularity. We have enriched the query by sorting the output and limiting the number of pages. Note that here we use a HiveQL specific expression, `rlike`, which allows to use standard regular expressions.

```
select req_url, count(*) as num_requests
from logs
where req_url rlike "html$" or req_url rlike "/$"
group by req_url
order by num_requests desc
limit 20;
```

Image Traffic Again we sort the output by traffic, as suggested at the end of MapReduce section. As explained before we need two jobs to perform that, and we can observe that in the output logs.

```
select req_url, sum(rep_bytes) as traffic
from logs
where req_url like "%gif"
group by req_url
having traffic > 5 * 1024 * 1024
order by traffic desc;
```

3.3 Movielens Dataset

3.3.1 Databases and Tables

As before we create a database and move to it:

```
create database if not exists movielens;
use movielens;
```

Now we have to create 3 tables, one for the users, one for the movies and one for the ratings. In this case we are going to use the defualt `SerDe` for `textfile`. We just have to specify the field separator and the array separator. Indeed, Hive, as opposed to SQL, allows to use complex structures as types, such as arrays, structs and maps.

```
create table movies (
  id int,
  title string,
  genres array<string>
)
row format delimited
fields terminated by "*"
collection items terminated by "|"
lines terminated by "\n"
stored as textfile;
```

Note than in this case we do not specify a location, Hive creates a directory for the table in the database directory. Additionally we have to load the data to it.

```
load data inpath "/data/movielens_1m_simple/movies"
into table movies;
```

As a result of the `load` command, Hive has moved the files into the table directory. Since we need the other tables for the last part of the tutorial, we define them as external and specify their he location.

```
create external table users (
  id int,
  sex string,
  age int,
  occupation int,
  zipcode string
)
row format delimited
fields terminated by "*"
collection items terminated by "|"
lines terminated by "\n"
stored as textfile
location "/data/movielens_1m_simple/users";
```

```
create external table ratings (
  user_id int,
  movie_id int,
  rating int,
  ts bigint
)
row format delimited
fields terminated by "*"
collection items terminated by "|"
lines terminated by "\n"
stored as textfile
location "/data/movielens_1m_simple/ratings";
```

3.3.2 Arrays

We have seen that Hive also allows arrays as types in the tables. It is possible to access specific positions of the array as in a typical programming language. It is also possible to generate a tuple for each element of the array with the command `explode`:

```
select explode(genres) as genre
from movies;
```

In order to combine `explode` with a projection of any other column we need to use `lateral view`. In this way, if we want to show the movie title and the genres at the same time we have to execute a query like this:

```
select title, genre
from movies
lateral view explode(genres) genre_view as genre;
```

Most Popular Genre. With these new expressions we can do some interesting queries. For instance, which is the most popular genre. We also use another syntax feature of HiveQL:

```
from (select explode(genres) as genre from movies) genres
select genres.genre, count(*) as popularity
group by genres.genre
order by popularity desc;
```

3.3.3 Joins

Hive also allows to perform joins on tables. The syntax is similar to that of standard SQL. Note that the order of the tables in the join matters, as Hive assumes that the last table is the largest to optimize the resulting MapReduce jobs.

Titles of the Best Rated Movies. With this join query we show the title of the 20 movies with the best average rating.

```
select m.title as movie, avg(r.rating) as avg_rating
from movies m join ratings r on m.id = r.movie_id
group by m.title
order by avg_rating desc
limit 20;
```

4 Apache Spark

In the last part of the tutorial we use Spark to execute the same queries previously executed with MapReduce and Hive.

Spark provides APIs to execute jobs in Java, Python, Scala and R and two interactive shells in Python and Scala. The examples of this tutorial are executed with the Python API. To open the corresponding shell just type:

```
# pyspark
```

Alternatively, if you want to benefit from iPython features you can use the following command:

```
# IPYTHON=1 pyspark
```

4.1 Introduction

Apache Spark is a general-purpose cluster computing system which provides efficient execution by performing in-memory transformations of data. As opposed to MapReduce, it allows arbitrary workflows with any number of stages and provides a broader choice of transformations, including map and reduce.

Spark can work with different cluster managers such as Hadoop's YARN or Apache Mesos. In this tutorial we are using the standalone mode.

The first thing to do when executing a program in Spark is to create a `SparkContext`. When using the shell, this is automatically done so this step is not necessary. In this case, the context is stored in the variable `sc`.

```
conf = SparkConf().setAppName("spark_tutorial").setMaster("spark://hadoopvm:7077")
sc = SparkContext(conf=conf)
```

4.2 Nasa Logs

4.2.1 RDDs

The core structures in Spark are the *Resilient Distributed Datasets*, RDDs, fault-tolerant collections of elements that can be operated in parallel. RDDs can be created from external data sources, such as HDFS. They can also be obtained by applying transformations to another RDD. For instance, we can create and RDD from the nasa logs file:

```
raw_logs = sc.textFile("hdfs://localhost:9000/data/logs/NASA_access_log_Jul95")
```

Note that in Spark, execution is done lazily: only when results are required, the executions are performed. We can force Spark to read the file, for instance, by getting the first 10 elements:

```
raw_logs.take(10)
```

Now we can start to apply operations to the data. First, we are going to parse the data, so it is conveniently structured. First we define a parsing function with the same regular expression as before:

```
import re
def parse_log_line(line):
    m = re.search('([^ ]*) - - \\[(.*)\\] "([^ ]*) ([^ ]*).*" ([^ ]*) ([^ ]*)', line)
    if m:
        record = (m.group(1),m.group(2),m.group(3), m.group(4))
        record += (int(m.group(5)),) if m.group(5).isdigit() else (None,)
        record += (int(m.group(6)),) if m.group(6).isdigit() else (0,)
        return record
    else:
        return None
```

Then we apply this function to each line and filter void tuples. As we can see it is possible to chain transformations as the result of one transformation is also an RDD. Here we use two types of transformations:

- `map`: which applies a function to all the elements of the RDD and
- `filter`: which filters out those elements for which the provided function returns false.

```
logs = raw_logs.map(parse_log_line).filter(lambda r: r).cache()
```

As said before, Spark executes operations lazily. If we want something to happen, we can execute an action like showing the first element:

```
logs.first()
```

Another important thing we have done is caching. Since `logs` is an RDD that we are going to reuse, we can cache it in memory so that all the previous transformations done to generate it do not need to be executed each time. To indicate that we use the method `cache()`.

At this point we can reexecute the examples shown before with Spark.

Method Count. We just apply the same map and reduce functions as in MapReduce but with Spark's syntax.

```
count_methods = logs.map(lambda record: (record[2], 1))\
                    .reduceByKey(lambda a, b: a + b)
```

We can show all the results with the action `collect()`[2].

```
count_methods.collect()
```

Web Page Popularity. To enrich this example, after performing the grouping and count, we sort the results by popularity. We need to swap the elements of each tuple and then use the transformation `sortByKey()`. The option passed to this method indicates that the ordering should be done in descending order.

```
web_page_urls = logs.map(lambda record: record[3])\
                    .filter(lambda url: url.endswith(".html") or url.endswith("/"))
web_page_popularity = web_page_urls.map(lambda url: (url, 1))\
```

[2]We illustrate the use of `collect()` here because we know the number of results is low. As a rule of thumb, do not use this command as it can lead to problems of memory, since it will load all the RDD elements in a variable.

```
                                    .reduceByKey(lambda a, b: a + b)\
                                    .map(lambda kv: (kv[1], kv[0]))\
                                    .sortByKey(False)
web_page_popularity.take(20)
```

Image Traffic. For the image traffic example we use the same strategy as before with an additional filter to remove images with less that the minimum traffic.

```
min_traffic = 150
bytes_per_image = logs.map(lambda r: (r[3], int(r[5])))\
                      .filter(lambda url_bytes: url_bytes[0].endswith(".gif"))
img_traffic = bytes_per_image.reduceByKey(lambda a, b: a + b)\
                             .map(lambda kv: (kv[1] / (1024 * 1024), kv[0]))\
                             .sortByKey(False)
selected_img_traffic = img_traffic.filter(lambda a: a[0] > min_traffic)
selected_img_traffic.take(20)
```

4.2.2 Spark SQL

Apache Spark also offers the user the possibility of querying the RDDs by using SQL statements. For that purpose, we first need to create a `SQLContext`.

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
```

Spark SQL provides different mechanisms to define SQL tables. In this first example we are going to automatically infer the table schema. Other possible methods to define SQL tables will be presented with the MovieLens dataset.

We first load the data and store it in a RDD. We need a different parsing function as we need to create elements of type `Row`. Note that the name of the optional parameters passed in the row construction define the names of the columns of the SQL table.

```
import re
from pyspark.sql import Row

def parse_log_line(line):
    m = re.search(’([^ ]*) - - \\[(.*)\\] "([^ ]*) ([^ ]*).*" ([^ ]*) ([^ ]*)’, line)
    if m:
        return Row(host=m.group(1),
                   req_date=m.group(2),
                   req_method=m.group(3),
                   req_url=m.group(4),
                   rep_code=int(m.group(5)) if m.group(5).isdigit() else None,
                   rep_bytes=int(m.group(6)) if m.group(6).isdigit() else 0)
    else:
        return None

logs = raw_logs.map(parse_log_line).filter(lambda r: r).cache()
```

Then we can use the schema inferring mechanism to generate the table. We also need to register the table so that it can be accessed by SQL statements. Finally, we show the obtained schema:

```
schemaLogs = sqlContext.inferSchema(logs)
schemaLogs.registerTempTable("logs")
schemaLogs.printSchema()
```

Now we can use SQL to query the data. For instance, the first example would be done like this:

```
count_methods = sqlContext.sql(
    "select req_method, count(*) as num_requests "
    "from logs "
    "group by req_method")
count_methods.collect()
```

You can try to execute the other examples by reusing the SQL commands executed in the Hive part of the tutorial.

4.3 MovieLens Dataset

For the Movielens dataset we are going to use three additional mechanisms to load each of the three tables. Additionally, we use a `HiveContext`, which is a superset of `SQLContext` as it provides the same methods plus HiveQL compatibility.

```
from pyspark.sql import HiveContext
sqlContext = HiveContext(sc)
```

4.3.1 User-defined Schema

In Spark SQL, the user can manually specify the schema of the table. The first step, as always, is to load the RDD and parse it. We define a new parsing function that applies the corresponding type conversions.

```
def parse_user(line):
    fields = line.split("*")
    fields[0] = int(fields[0])
    fields[2] = int(fields[2])
    fields[3] = int(fields[3])
    return fields

users_lines = \
    sc.textFile("hdfs://localhost:9000/data/movielens_1m_simple/users/users.dat")
users_tuples = users_lines.map(parse_user)
```

Now, we can manually define the columns by specifying its name, type and whether the values can be null or not. Then, we use this schema to create and register the table.

```
from pyspark.sql.types import StructField, StringType, IntegerType, StructType

fields = [
    StructField("id", IntegerType(), False),
    StructField("gender", StringType(), False),
    StructField("age", IntegerType(), False),
    StructField("occupation", IntegerType(), True),
    StructField("zipcode", StringType(), False),
]
```

```
users_schema = StructType(fields)
users = sqlContext.applySchema(users_tuples, users_schema)
users.registerTempTable("users")
users.printSchema()
```

4.3.2 JSON

Spark, either within its core or as third-party libraries, provides mechanisms to load standard file formats, such as JSON, Parquet files or csv files. Here we will load the movies table from a JSON file. We can observe that this mechanism is very simple to use.

```
movies = sqlContext.jsonFile(hdfs_prefix + "/data/movielens_1m_mod/movies.json")
movies.registerTempTable("movies")
movies.printSchema()
```

4.3.3 Hive Table Loading

Finally, tables can be loaded as in Hive. It is not the case of this tutorial, but if we have correctly linked Hive and Spark and use this mechanism, changes made in both systems will be reflected in the same metastore and both systems can operate over the same schemas.

```
sqlContext.sql(
    "create table ratings ( "
    "user_id int, "
    "movie_id int, "
    "rating int, "
    "ts bigint) "
    "row format delimited "
    "fields terminated by '*' "
    "lines terminated by '\n' "
    "stored as textfile")

sqlContext.sql(
    "load data inpath '/data/movielens_1m_simple/ratings/ratings.dat' "
    "into table ratings")
```

4.3.4 Hive Queries

When using a `HiveContext` we can use the HiveQL syntax specific features to access our tables. For instance, we can use the expression `explode` that we have seen before:

```
genres = sqlContext.sql(
    "select explode(genres) as genre "
    "from movies")
genres.take(10)
```

4.3.5 Data Frames

Maybe you have notice that when creating a schema, a `DataFrame` is created, instead of a RDD. Apart from allowing the usage of SQL statements, `DataFrame`s also provide additional methods to query its contents.

```
age_oc2 = users.filter("gender = 'M'")\
               .select("occupation","age")\
               .groupBy("occupation")\
               .avg("age")\
               .orderBy("AVG(age)", ascending=0)
age_oc2.take(10)
```