Apache Hadoop –

A course for undergraduates

Homework Labs, Lecture 8

Lab: Importing Data with Sqoop

In this lab you will import data from a relational database using Sqoop. The data you load here will be used subsequent labs.

Consider the MySQL database `movielens`, derived from the MovieLens project from University of Minnesota. (See note at the end of this lab.) The database consists of several related tables, but we will import only two of these: `movie`, which contains about 3,900 movies; and `movierating`, which has about 1,000,000 ratings of those movies.

Review the Database Tables

First, review the database tables to be loaded into Hadoop.

1. Log on to MySQL:

```
$ mysql --user=training --password=training movielens
```

2. Review the structure and contents of the `movie` table:

```
mysql> DESCRIBE movie;
. . .
mysql> SELECT * FROM movie LIMIT 5;
```

3. Note the column names for the table:

____

4. Review the structure and contents of the `movierating` table:

```
mysql> DESCRIBE movierating;
…
mysql> SELECT * FROM movierating LIMIT 5;
```

5. Note these column names:

____

6. Exit mysql:

```
mysql> quit
```

Import with Sqoop

You invoke Sqoop on the command line to perform several commands. With it you can connect to your database server to list the databases (schemas) to which you have access, and list the tables available for loading. For database access, you provide a connect string to identify the server, and - if required - your username and password.

1. Show the commands available in Sqoop:

```
$ sqoop help
```

2. List the databases (schemas) in your database server:

```
$ sqoop list-databases \
--connect jdbc:mysql://localhost \
--username training --password training
```

(Note: Instead of entering `--password training` on your command line, you may prefer to enter `-P`, and let Sqoop prompt you for the password, which is then not visible when you type it.)

3. List the tables in the `movielens` database:

```
$ sqoop list-tables \
  --connect jdbc:mysql://localhost/movielens \
  --username training --password training
```

4. Import the `movie` table into Hadoop:

```
$ sqoop import \
  --connect jdbc:mysql://localhost/movielens \
  --username training --password training \
  --fields-terminated-by '\t' --table movie
```

5. Verify that the command has worked.

```
$ hadoop fs -ls movie
$ hadoop fs -tail movie/part-m-00000
```

6. Import the `movierating` table into Hadoop.

   Repeat the last two steps, but for the `movierating` table.

This is the end of the lab.

Note:

This lab uses the MovieLens data set, or subsets thereof. This data is freely available for academic purposes, and is used and distributed by Cloudera with the express permission of the UMN GroupLens Research Group. If you would like to use this data for your own research purposes, you are free to do so, as long as you cite the GroupLens Research Group in any resulting publications. If you would like to use this data for commercial purposes, you must obtain explicit permission. You may find the full dataset, as well as detailed license terms, at http://www.grouplens.org/node/73

Lab: Running an Oozie Workflow

Files and Directories Used in this Exercise

Exercise directory: `~/workspace/oozie_labs`

Oozie job folders:

`lab1-java-mapreduce`

`lab2-sort-wordcount`

In this lab, you will inspect and run Oozie workflows.

1. Start the Oozie server

```
$ sudo /etc/init.d/oozie start
```

2. Change directories to the lab directory:

```
$ cd ~/workspace/oozie-labs
```

3. Inspect the contents of the `job.properties` and `workflow.xml` files in the `lab1-java-mapreduce/job` folder. You will see that this is the standard WordCount job.

In the `job.properties` file, take note of the job's base directory (`lab1-java-mapreduce`), and the input and output directories relative to that. (These are HDFS directories.)

4. We have provided a simple shell script to submit the Oozie workflow. Inspect the `run.sh` script and then run:

```
$ ./run.sh lab1-java-mapreduce
```

Notice that Oozie returns a job identification number.

5. Inspect the progress of the job:

```
$ oozie job -oozie http://localhost:11000/oozie \
-info job_id
```

6. When the job has completed, review the job output directory in HDFS to confirm that the output has been produced as expected.

7. Repeat the above procedure for `lab2-sort-wordcount`. Notice when you inspect `workflow.xml` that this workflow includes two MapReduce jobs which run one after the other, in which the output of the first is the input for the second. When you inspect the output in HDFS you will see that the second job sorts the output of the first job into descending numerical order.

This is the end of the lab.

Bonus Lab: Exploring a Secondary Sort Example

Files and Directories Used in this Exercise

Eclipse project: `secondarysort`

Data files:

`~/training_materials/developer/data/nameyeartestdata`

Exercise directory: `~/workspace/secondarysort`

In this lab, you will run a MapReduce job in different ways to see the effects of various components in a secondary sort program.

The program accepts lines in the form

```
lastname firstname birthdate
```

The goal is to identify the youngest person with each last name. For example, for input:

```
Murphy Joanne 1963-08-12
Murphy Douglas 1832-01-20
Murphy Alice 2004-06-02
```

We want to write out:

```
Murphy Alice 2004-06-02
```

All the code is provided to do this. Following the steps below you are going to progressively add each component to the job to accomplish the final goal.

Build the Program

1. In Eclipse, review but do not modify the code in the `secondarysort` project `example` package.

2. In particular, note the `NameYearDriver` class, in which the code to set the partitioner, sort comparator, and group comparator for the job is commented out. This allows us to set those values on the command line instead.

3. Export the jar file for the program as `secsort.jar`.

4. A small test datafile called `nameyeartestdata` has been provided for you, located in the secondary sort project folder. Copy the datafile to HDFS, if you did not already do so in the Writables lab.

Run as a Map-only Job

5. The Mapper for this job constructs a composite key using the `StringPairWritable` type. See the output of just the mapper by running this program as a Map-only job:

```
$ hadoop jar secsort.jar example.NameYearDriver \
-Dmapred.reduce.tasks=0 nameyeartestdata secsortout
```

6. Review the output. Note the key is a string pair of last name and birth year.

Run using the default Partitioner and Comparators

7. Re-run the job, setting the number of reduce tasks to `2` instead of `0`.

8. Note that the output now consists of two files; one each for the two reduce tasks. Within each file, the output is sorted by last name (ascending) and year (ascending). But it isn't sorted between files, and records with the same last name may be in different files (meaning they went to different reducers).

Run using the custom partitioner

9. Review the code of the custom partitioner class: `NameYearPartitioner`.

10. Re-run the job, adding a second parameter to set the partitioner class to use:
    `-Dmapreduce.partitioner.class=example.NameYearPartitioner`

11. Review the output again, this time noting that all records with the same last name have been partitioned to the same reducer.

    However, they are still being sorted into the default sort order (name, year ascending). We want it sorted by name ascending/year descending.

Run using the custom sort comparator

12. The `NameYearComparator` class compares Name/Year pairs, first comparing the names and, if equal, compares the year (in descending order; i.e. later years are considered "less than" earlier years, and thus earlier in the sort order.) Re-run the job using NameYearComparator as the sort comparator by adding a third parameter:
    ```
    -D mapred.output.key.comparator.class=
    example.NameYearComparator
    ```

13. Review the output and note that each reducer's output is now correctly partitioned and sorted.

Run with the NameYearReducer

14. So far we've been running with the default reducer, which is the Identity Reducer, which simply writes each key/value pair it receives. The actual goal of this job is to emit the record for the *youngest person* with each last name. We can do this easily if all records for a given last name are passed to a single reduce call, sorted in descending order, which can then simply emit the first value passed in each call.

15. Review the NameYearReducer code and note that it emits

16. Re-run the job, using the reducer by adding a fourth parameter:
    `-Dmapreduce.reduce.class=example.NameYearReducer`

    Alas, the job still isn't correct, because the data being passed to the reduce method is being grouped according to the full key (name and year), so multiple records with the same last name (but different years) are being output. We want it to be grouped by name only.

Run with the custom group comparator

17. The `NameComparator` class compares two string pairs by comparing only the name field and disregarding the year field. Pairs with the same name will be grouped into the same reduce call, regardless of the year. Add the group comparator to the job by adding a final parameter:
    ```
    -Dmapred.output.value.groupfn.class=
    example.NameComparator
    ```

18. Note the final output now correctly includes only a single record for each different last name, and that that record is the youngest person with that last name.

This is the end of the lab.