THE UNIVERSITY OF SYDNEY

School of Information Technologies
Dr. Ying Zhou

COMP5349: Cloud Computing **Sem. 1/2018**

Week 6: Distributed Execution Engine

19.04.2018

In this lab, you will experience executing MapReduce and Spark program in a distributed environment. School of IT manages a cluster consisting of 30 Linux virtual machines. All machines have 4 core and 8G memory. Each has 250G storage. HDFS and YARN are running on the cluster. One machine is dedicated as the "master" node. while all others are slave nodes. The "master" node takes the coordinator role in both HDFS and YARN. It also runs the history server for MapReduce and for Spark. All machines in the cluster have names like `soit-hdp-pro-[X].ucc.usyd.edu.au`. The name of the "master" node is `soit-hdp-pro-1.ucc.usyd.edu.au`. The rest of nodes are named from `soit-hdp-pro-2.ucc.usyd.edu.au` to `soit-hdp-pro-30.ucc.usyd.edu.au`.

The cluster is configured and maintained by Cloud Computing course staff. All students in the cloud computing course have access to it. This cluster is always on but may become slow when there are many users competing for the limited resources.

Below are the services running on the "master" node as well as the respective WebUI:

- HDFS name node process, with a web UI:
  `http://soit-hdp-pro-1.ucc.usyd.edu.au:50070`.
- YARN Resource Manager process, with a web UI:
  `http://soit-hdp-pro-1.ucc.usyd.edu.au:8088`.
- MapReduce history server, with a web UI:
  `http://soit-hdp-pro-1.ucc.usyd.edu.au:19888`
- Spark history server, with a web UI:
  `http://soit-hdp-pro-1.ucc.usyd.edu.au:18080`

Question 1: Inspecting HDFS

Access the HDFS Web UI `http://soit-hdp-pro-1.ucc.usyd.edu.au:50070` from a browser. The web UI would look similar to the web UI you have seen in the pseudo distributed mode. The "Overview" tab would show general information of the cluster software,configuration as well as the file system information. You may notice that the cluster runs a slightly oder version of Hadoop (`2.7.2`). In pseudo distributed mode, you always see a single living data node, which is your local host. Our cluster has 29 living data nodes.

HDFS uses replication to achieve durability and high availability. The replication factor is a configurable parameter. In pseudo distributed mode, we set the replication factor to 1 since there is a single physical node. See configuration file `hdfs-site.xml`

```
<property>
    <name>dfs.replication</name>
    <value>1</value>
</property>
```

We configured our cluster to have a replication factor of 3. This is the most common replication factor value. It means there are 3 copies of each file stored on HDFS. The copies will be stored in different nodes. HDFS is designed to store very large files. It does not store large file as a single entity. In stead, files are partitioned into fixed sized blocks. The default size is 128M. For instance, a file of 400MB will be stored as 4 blocks, the size of the first three blocks are 128MB each. The size of the last block is 16M. Each block will have three copies. In total, this file would be stored in 12 blocks. HDFS ensures the blocks are distributed relatively even in the cluster.

The storage distribution information can be easily found using the HDFS web UI. From the HDFS Web UI, click the menu `Utilities` then the menu item `Browse the filesystem` and navigate to the folder `/share/photo`, click file `n06.txt` to inspect its block distribution. By default, it will show block 0's information. Basic information include the *block ID*, which is an internal ID HDFS used to identify each blocks; and `availability`, which shows the nodes that have a copy of this block.

The size of `n0.6txt` is 386.86MB. It is stored as 4 blocks in HDFS. You can use the drop down list to select block 1,2 and 3 and display their respective storage distribution information. Try to find out the location of the replica for each block and answer the following questions

- Are replicas of the same block distributed on different nodes?
- Is there any node which has all the blocks of this file?
- Which node has the meta data information such as the number of blocks and their replica locations?

Similar to local installation, you can interact with the distributed HDFS using HDFS shell command. You may issue the command from a remote client. It is more convenient to login to one of the slave node and use that as the client to issue commands there.

HDFS shell uses the Hadoop configuration file to find the location of name node and that becomes the default name node location. The default path of HDFS shell is `/user/<yourLogInName>`. You can use relative path for files or directories under your HDFS home.

The cluster configuration file `core-site.xml` has the following property:

```
<property>
        <name>fs.defaultFS</name>
        <value>hdfs://soit-hdp-pro-1.ucc.usyd.edu.au:8020</value>
    </property>
```

You can use the following HDFS command to list the content of file `place.txt` under your cluster's HDFS home: `/user/<yourLogIn>/place.txt`

```
    hdfs dfs -cat place.txt
```

Referring to files or directories not under your HDFS home needs absolute path. For instance, the following command lists the content of directory `/share/movie` on cluster HDFS:

```
    hdfs dfs -ls /share/movie
```

Question 2: MapReduce Execution on YARN

a) **Job Submission**
The easiest way to submit a MapReduce job to a cluster is to SSH to one of the slave nodes and submit from there. Your home directories are mounted to the cluster nodes.

Once you log in one of the slave nodes, you will the same home directory as that in your lab workstation. you can use the following script to run the week 4 hadoop Java program on a large data set.

```
hadoop \
    jar userTag.jar \
    usertag.TagDriver \
    /share/photo/n08.txt \
    n08_out_java
```

Below is the script for submitting Python program

```
hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-2.7.2.jar \
-D mapreduce.job.reduces=3 \
-D mapreduce.job.name='Tag owner inverted list' \
-file tag_mapper.py \
-mapper tag_mapper.py \
-file tag_reducer.py \
-reducer tag_reducer.py \
-input /share/photo/n08.txt \
-output n08_out_java
```

b) **Job Progress Information**
After submitting the job successfully. MapReduce framework will start to run the job and print many information on the console. Below is the output regarding job progress after running the about submit command :

```
INFO client.RMProxy: Connecting to ResourceManager at soit-hdp-pro-1.ucc.usyd.edu.au:8032
INFO input.FileInputFormat: Total input files to process : 1
INFO mapreduce.JobSubmitter: number of splits:4

INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1522231243385_0346
INFO impl.YarnClientImpl: Submitted application application_1522231243385_0346
INFO mapreduce.Job: The url to track the job: http://soit-hdp-pro-1.ucc.usyd.edu.au:8088...
INFO mapreduce.Job: Running job: job_1522231243385_0346

INFO mapreduce.Job:  map 0% reduce 0%
INFO mapreduce.Job:  map 25% reduce 0%
...
INFO mapreduce.Job:  map 69% reduce 0%
INFO mapreduce.Job:  map 69% reduce 6%
INFO mapreduce.Job:  map 69% reduce 8%
INFO mapreduce.Job:  map 75% reduce 8%
INFO mapreduce.Job:  map 83% reduce 8%
INFO mapreduce.Job:  map 98% reduce 8%
INFO mapreduce.Job:  map 100% reduce 19%
INFO mapreduce.Job:  map 100% reduce 25%
...
INFO mapreduce.Job:  map 100% reduce 78%
INFO mapreduce.Job:  map 100% reduce 90%
INFO mapreduce.Job:  map 100% reduce 100%
INFO mapreduce.Job: Job job_1522231243385_0346 completed successfully
```

You should have seen similar output when running jobs in standalone or pseudo distributed mode. The progress report might be much shorter for small input data. You may see reduce starts after map has finished 100% most of the time. It is not the case with large input data.

In the above sample output, you see that the reduce phase starts when the map phase is about 69% completes. It is quite common to see reducers starts before mapper finishes in large MapReduce jobs. The first step any reducer does is to obtain intermediate results from mappers. This is call `shuffle` in MapReduce terminology. To improve execution time, reducer starts shuffling data before mapper finishes execution. This is possible because mapper emits results continuously on each input key value pair. The mapper calls the user supplied map function on every line of the input file and each call generates some key value pairs as the result. The results will be buffered in memory and spill to the disk. Reducers establish RPC calls with the mapper nodes to obtain the intermediate results on the disk. Theoretically, reducers can start to request intermediate results after the first batch of intermediate results are spilled to disk by mapper.

The next part of the output include important execution statistics. Those execution statistics are logged and can be viewed from the history server webUI afterwards. The first section of the execution statistics contains File System counters. Below is the sample output of application `application_1522231243385_0346`:

```
File System Counters
    FILE: Number of bytes read=1215622626
    FILE: Number of bytes written=1838285658
    FILE: Number of read operations=0
    FILE: Number of large read operations=0
    FILE: Number of write operations=0
    HDFS: Number of bytes read=422008718
    HDFS: Number of bytes written=49335739
    HDFS: Number of read operations=21
    HDFS: Number of large read operations=0
    HDFS: Number of write operations=6
```

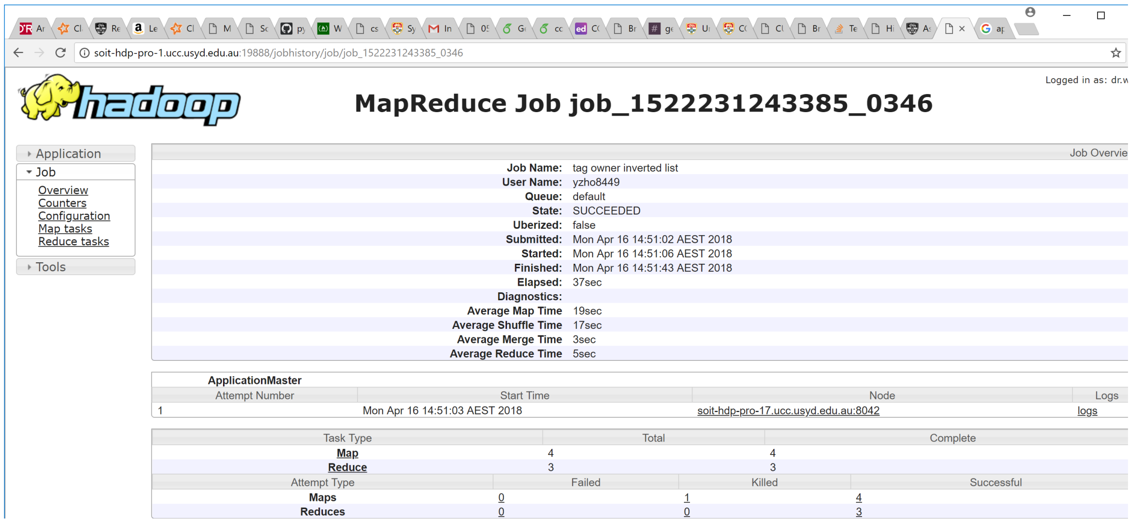
The section reports statistics about local and HDFS read/write. The program's input is located on HDFS `/share/photo/n08.txt`. The file is about 402MB, roughly the same as the reported HDFS bytes read: `422008718`. The program's output is written on HDFS and is about 48MB in total, roughly the same as the reported HDFS byte written `49335739`.The local file system read and write happens during the shuffle period.

c) **Overall Job Statistics**
The next section reports statistics about the overall job. Below is the sample output from the same application:

```
Job Counters
    Killed map tasks=1
    Launched map tasks=5
    Launched reduce tasks=3
    Data-local map tasks=1
    Rack-local map tasks=4
    Total time spent by all maps in occupied slots (ms)=181580
    Total time spent by all reduces in occupied slots (ms)=320832
    Total time spent by all map tasks (ms)=90790
    Total time spent by all reduce tasks (ms)=80208
    Total vcore-milliseconds taken by all map tasks=90790
    Total vcore-milliseconds taken by all reduce tasks=80208
    Total megabyte-milliseconds taken by all map tasks=185937920
    Total megabyte-milliseconds taken by all reduce tasks=328531968
```

In total five map tasks are launched and one is killed; three reduce tasks are launched. Our input file `n08.txt` is about 402MB in size and is stored as four blocks in HDFS. By default, there will be four map tasks, each work on a block of data. The fifth is started for speculative purpose to compensate for node slower than average. We have also specified to use three reducers in the code, hence three reduce tasks are launched. There is no speculative reduce tasks. For map tasks, it also report the number of `data-local` and `rack-local` tasks. `Data-local` task means the mapper is running on a node where a copy of its data is stored. This is the most efficient way, involving no network transfer. `Rack-local` task means the mapper is running on a node with no copy of the data, but the data is stored on a node in the same rack of the task node. `Rack-local` task involves transferring data between nodes, but the transfer happens within the rack. In our cluster, all nodes are assumed to be in the same rack. You won't see task with cross rack data location.

The history server webUI provides more detailed information on speculative tasks as well as data-local or rack-local tasks. Figure 1 shows the history overview of application `application_1522231243385_0346`. It shows four completed tasks and three completed reduce tasks. For the map tasks, 5 attempts have been started with four successful and one killed. There are three attempts for reduce tasks they all are successful.



soit-hdp-pro-1.ucc.usyd.edu.au:19888/jobhistory/job/job_1522231243385_0346

Logged in as: dr.w

MapReduce Job job_1522231243385_0346

Job Name:	tag owner inverted list
User Name:	yzho8449
Queue:	default
State:	SUCCEEDED
Uberized:	false
Submitted:	Mon Apr 16 14:51:02 AEST 2018
Started:	Mon Apr 16 14:51:06 AEST 2018
Finished:	Mon Apr 16 14:51:43 AEST 2018
Elapsed:	37sec
Diagnostics:	
Average Map Time	19sec
Average Shuffle Time	17sec
Average Merge Time	3sec
Average Reduce Time	5sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Mon Apr 16 14:51:03 AEST 2018	soit-hdp-pro-17.ucc.usyd.edu.au:8042	logs

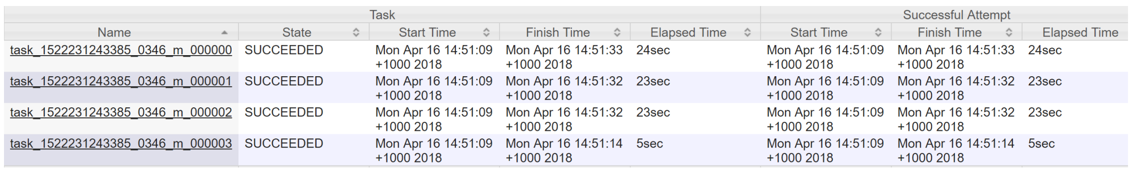
Task Type	Total	Complete
Map	4	4
Reduce	3	3

Attempt Type	Failed	Killed	Successful
Maps	0	1	4
Reduces	0	0	3

Figure 1: MapReduce Job Hisotry View on Cluster

Clicking either 'map' or 'reduce' would bring the summary screen for map and reduce tasks. Figure 2 shows the summary screen for all map tasks. It shows the start, end time of each task and the total elapsed time. The total elapsed time for the first three tasks are around 20 seconds. You will notice that the last task `task_1522231243385_0346_m_000003` takes much less time than the rest. It's total elapsed time is only 5 seconds. The map tasks' execution time depends a lot on the input data size. The size of the first three blocks of input file `n08.txt` is 128M each, while the size of the last block is around 19MB. This explains the shorter execution time.

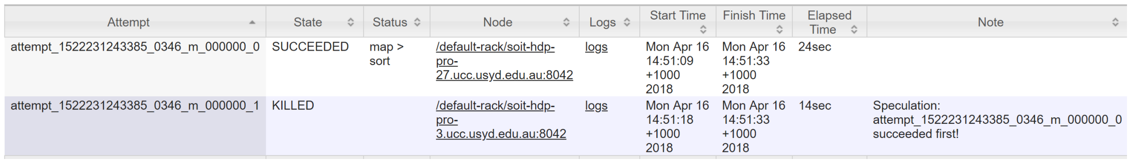
Clicking on each task would bring the detail screen for each task. The first task takes slightly longer time than the rest. Figure 3 shows the detail screen of the first map task. The detail screen shows the node executes the task and if multiple attempts have been started. In this example two attempted have been started. The first one `attempt_1522231243385_0346_m_000000_0` is started on node



Task					Successful Attempt		
Name	State	Start Time	Finish Time	Elapsed Time	Start Time	Finish Time	Elapsed Time
task_1522231243385_0346_m_000000	SUCCEEDED	Mon Apr 16 14:51:09 +1000 2018	Mon Apr 16 14:51:33 +1000 2018	24sec	Mon Apr 16 14:51:09 +1000 2018	Mon Apr 16 14:51:33 +1000 2018	24sec
task_1522231243385_0346_m_000001	SUCCEEDED	Mon Apr 16 14:51:09 +1000 2018	Mon Apr 16 14:51:32 +1000 2018	23sec	Mon Apr 16 14:51:09 +1000 2018	Mon Apr 16 14:51:32 +1000 2018	23sec
task_1522231243385_0346_m_000002	SUCCEEDED	Mon Apr 16 14:51:09 +1000 2018	Mon Apr 16 14:51:32 +1000 2018	23sec	Mon Apr 16 14:51:09 +1000 2018	Mon Apr 16 14:51:32 +1000 2018	23sec
task_1522231243385_0346_m_000003	SUCCEEDED	Mon Apr 16 14:51:09 +1000 2018	Mon Apr 16 14:51:14 +1000 2018	5sec	Mon Apr 16 14:51:09 +1000 2018	Mon Apr 16 14:51:14 +1000 2018	5sec

Figure 2: Summary screen of map tasks

`soit-hdp-pro-27.ucc.usyd.edu.au` at `Mon Apr 16 14:51:09 +1000 2018`. The attempt's progress is considered as slower than average and at `Mon Apr 16 14:51:18 +1000 2018`, another attempt was started on node `soit-hdp-pro-3.ucc.usyd.edu.au` trying to catch up the speed. The second attempt was for speculation and was killed because the first attempt finishes first.



Attempt	State	Status	Node	Logs	Start Time	Finish Time	Elapsed Time	Note
attempt_1522231243385_0346_m_000000_0	SUCCEEDED	map > sort	/default-rack/soit-hdp-pro-27.ucc.usyd.edu.au:8042	logs	Mon Apr 16 14:51:09 +1000 2018	Mon Apr 16 14:51:33 +1000 2018	24sec	
attempt_1522231243385_0346_m_000000_1	KILLED		/default-rack/soit-hdp-pro-3.ucc.usyd.edu.au:8042	logs	Mon Apr 16 14:51:18 +1000 2018	Mon Apr 16 14:51:33 +1000 2018	14sec	Speculation: attempt_1522231243385_0346_m_000000_0 succeeded first!

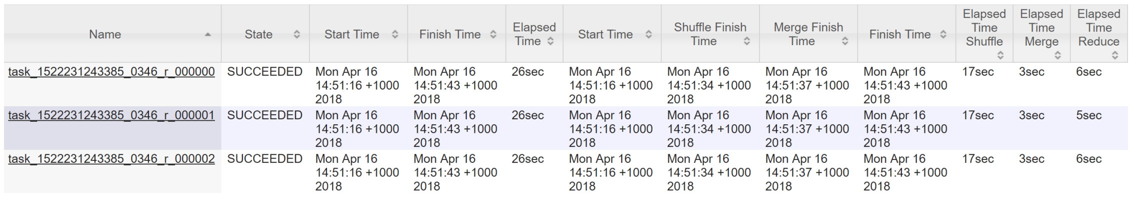
Figure 3: Detail screen of individual map task

Now inspect details of all individual map tasks and figure out which attempt is data-local, which are rack-local. You will need to check HDFS webUI to figure out the location of each block. And double check the log file of each task to see which block it is supposed to process.

The rest of the statistics in this section include execution times measured at various scales. The history server webUI provides a better view of time spent. Figure 4 shows the summary screen of reduce tasks. There are in total three reduce tasks with similar execution time. The execution time of reducer also depends a lot on the data it needs to process. Similar execution time indicates that the default partitioner partition the map results evenly to three partitions. The actual execution time of reducers are divided into three stages: shuffle, merge and reduce. The last stage reduce measures the time taken to run the user defined reduce function. The shuffle stage measures the time taken to transfer the map results to reducer nodes. The merge stage measures the time taken for reducer node to prepare merge results from different mapper and to group them for reducer input. It is clear that in this application, where the reduce function does simple processing, majority of the reducer time are spent on obtaining and preparing the data.

d) I/O Statistics and Comparison with Combiner Version Output

The next output section reports statistics about I/O and execution time for map and reduce stage. Again, the same information can be found from history server's `Counters` page. Below is a subset of the sample output, highlighting the size of shuffled data.



Name	State	Start Time	Finish Time	Elapsed Time	Start Time	Shuffle Finish Time	Merge Finish Time	Finish Time	Elapsed Time Shuffle	Elapsed Time Merge	Elapsed Time Reduce
task_1522231243385_0346_r_000000	SUCCEEDED	Mon Apr 16 14:51:16 +1000 2018	Mon Apr 16 14:51:43 +1000 2018	26sec	Mon Apr 16 14:51:16 +1000 2018	Mon Apr 16 14:51:34 +1000 2018	Mon Apr 16 14:51:37 +1000 2018	Mon Apr 16 14:51:43 +1000 2018	17sec	3sec	6sec
task_1522231243385_0346_r_000001	SUCCEEDED	Mon Apr 16 14:51:16 +1000 2018	Mon Apr 16 14:51:43 +1000 2018	26sec	Mon Apr 16 14:51:16 +1000 2018	Mon Apr 16 14:51:34 +1000 2018	Mon Apr 16 14:51:37 +1000 2018	Mon Apr 16 14:51:43 +1000 2018	17sec	3sec	5sec
task_1522231243385_0346_r_000002	SUCCEEDED	Mon Apr 16 14:51:16 +1000 2018	Mon Apr 16 14:51:43 +1000 2018	26sec	Mon Apr 16 14:51:16 +1000 2018	Mon Apr 16 14:51:34 +1000 2018	Mon Apr 16 14:51:37 +1000 2018	Mon Apr 16 14:51:43 +1000 2018	17sec	3sec	6sec

Figure 4: Summary screen of reduce tasks

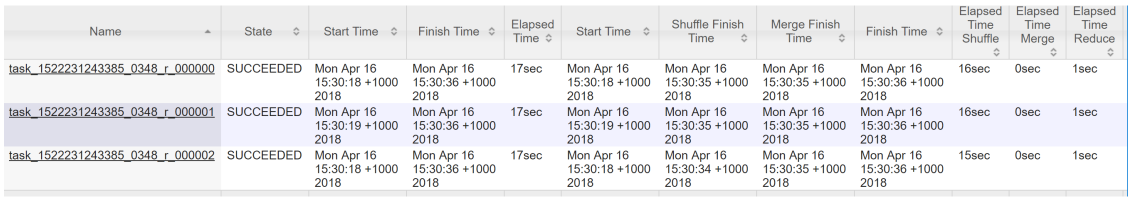
The application does not use combiner, all mapper's output are transferred to reducers. You can see that the `Map output materialized byptes` is exactly the same as `Reduce Shuffle bytes`. The `Map output records` is also exactly the same as `Reduce input records`.

```
Map-Reduce Framework
    Map output records=26162800
    Map output bytes=569076115
    Map output materialized bytes=621405024
    ...
    Combine input records=0
    Combine output records=0
    ...
    Reduce shuffle bytes=621405024
    Reduce input records=26162800
    Reduce output records=759579
    ...
```

We also run a combiner version of the same application, the application id is `application_152223124`
You can find detailed execution statistics of this application from the history server WebUI and compare it with the non-combiner version. Below is the console output highlighting the difference on shuffle size

```
Map-Reduce Framework
    Map input records=2723874
    Map output records=26162800
    Map output bytes=647564515
    Map output materialized bytes=53968201
    ...
    Combine input records=27421648
    Combine output records=2241794
    Reduce shuffle bytes=53968201
    Reduce input records=982946
    Reduce output records=759579
```

In the combiner version, the `Reduce shuffle bytes=53968201`. It is much smaller than the naive version with `Reduce shuffle bytes=621405024`. The `Reduce input records=982946` is also much smaller than the `Map output records=26162800`. The combiner decreases the record size by 10 fold: `Combine input records=27421648` becomes `Combine output records=2241794`. As a result the overall reduce tasks take 17 seconds (see Figure 5).



Name	State	Start Time	Finish Time	Elapsed Time	Start Time	Shuffle Finish Time	Merge Finish Time	Finish Time	Elapsed Time Shuffle	Elapsed Time Merge	Elapsed Time Reduce
task_1522231243385_0348_r_000000	SUCCEEDED	Mon Apr 16 15:30:18 +1000 2018	Mon Apr 16 15:30:36 +1000 2018	17sec	Mon Apr 16 15:30:18 +1000 2018	Mon Apr 16 15:30:35 +1000 2018	Mon Apr 16 15:30:35 +1000 2018	Mon Apr 16 15:30:36 +1000 2018	16sec	0sec	1sec
task_1522231243385_0348_r_000001	SUCCEEDED	Mon Apr 16 15:30:19 +1000 2018	Mon Apr 16 15:30:36 +1000 2018	17sec	Mon Apr 16 15:30:19 +1000 2018	Mon Apr 16 15:30:35 +1000 2018	Mon Apr 16 15:30:35 +1000 2018	Mon Apr 16 15:30:36 +1000 2018	16sec	0sec	1sec
task_1522231243385_0348_r_000002	SUCCEEDED	Mon Apr 16 15:30:18 +1000 2018	Mon Apr 16 15:30:36 +1000 2018	17sec	Mon Apr 16 15:30:18 +1000 2018	Mon Apr 16 15:30:34 +1000 2018	Mon Apr 16 15:30:35 +1000 2018	Mon Apr 16 15:30:36 +1000 2018	15sec	0sec	1sec

Figure 5: Summary screen of reduce tasks in Combiner version

e) `Run the Sample Application on Different Input`
Now try to run the two versions of the sample application on a different input and inspect the execution statistics. You can pick any file under the `/share/photo` directory as the input.

Question 3: Spark Execution on YARN

Spark program can be deployed and managed by YARN. The spark driver program can run inside the cluster (cluster deploy mode) or as an external client program (client deploy mode). `Client` mode is the default option. The deploy mode option can be specified explicitly in the spark-submit script. In this section, we will run the Spark sample program you have seen in week 5 lab on a larger data set stored on HDFS under `/share/movie`

Below is the script for submitting the Java sample Program in week 5 lab:

```
spark-submit  \
  --class ml.MovieLensLarge \
  --master yarn \
  --deploy-mode cluster \
  --num-executors 3 \
  sparkML.jar \
  /share/movie/ \
  week6-out-java
```

Pyspark by default uses python 2. Most of the sample code we provide are in Python3. To make it work with the particular Python3 version installed in cluster node, you need to set the following three environment variables:

```
    export PYSPARK_PYTHON=python3.3
    export PYTHONHASHSEED=0
    export SPARK_YARN_USER_ENV=PYTHONHASHSEED=0
```

An updated login template has been pushed to `lab_commons` repository

Below is the script for submitting the Python sample Program in week 5 lab. Different to Java, Python files are not prepackaged. To specify multiple files, you need to use `--py-files` option, the last one should be the "main" script.

```
spark-submit \
    --master yarn \
    --deploy-mode client \
    --py-files ml_utils.py AverageRatingPerGenre.py \
    --num-executors 3 \
    --input /share/movie/ \
    --output week6-out-python
```

Spark by default is not configured to print out a lot of running statistics on the console. In stead, it provides a history server to visually view the statistics. The first information you want to see is usually the job execution DAG. A spark program may have many jobs, each representing an independent data flow leading to some result. The week 5 sample program has only one job. Figure 6 shows execution DAG of the Java version. The

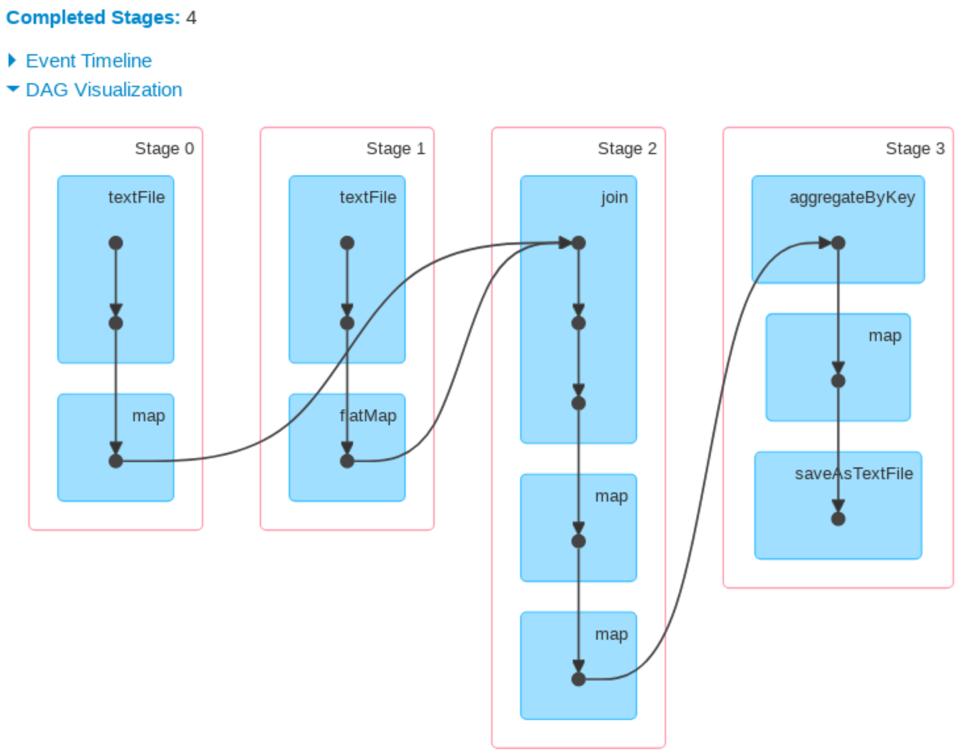
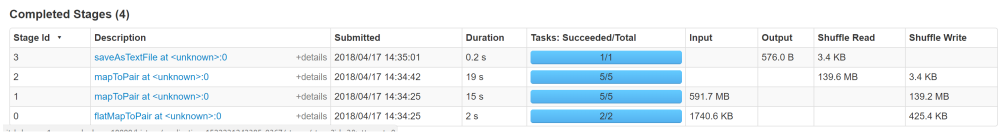


Figure 6: Average Rating Per Genre Java DAG

application has one job and 4 stages, each with a variable number of tasks. The number of tasks are show in Figure 7 We specify in the submit script to have 3 executors to run the application. You may run this program yourself. Or you can find a complete execution history of this program in `application_1522231243385_0369`. Check the history to find:



Completed Stages (4)

Stage Id ▾	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3	saveAsTextFile at <unknown>:0	+details	2018/04/17 14:35:01	0.2 s	1/1		576.0 B	3.4 KB	
2	mapToPair at <unknown>:0	+details	2018/04/17 14:34:42	19 s	5/5			139.6 MB	3.4 KB
1	mapToPair at <unknown>:0	+details	2018/04/17 14:34:25	15 s	5/5	591.7 MB			139.2 MB
0	flatMapToPair at <unknown>:0	+details	2018/04/17 14:34:25	2 s	2/2	1740.6 KB			425.4 KB

Figure 7: Average Rating Per Genre Java Task Details

- How many tasks are there for each executor?
- The locality of various tasks

Question 4: Understand DAG: job, stage and task

a) **Another Spark Sample Program** We provide another application to help you understand the important spark execution concepts: job, stage and task. In particular, it helps to understand when shuffle will or will not happen. This application finds the top movies per genre based on the number of ratings a movie receives. It runs on the same movie rating data set. Both Java and Python versions are provided. Because Python DAG is not consistent with the actual code. We use the Java example here.

Pull the week6 code from Java or Python repository respectively. The two implementations have exactly the same logic. The Java implementation contains two source files: `MLGenreTopMoviesNaive.java` and `MovieRatingCount.java`. The Python version contains a single script: `Top5MoviesPerGenreNaive.py`. Inspect the source code in your favorite text editor.

The overall processing is similar to the previous one. Two new operations `reduceByKey` and `groupByKey` are used. Both operations have wide dependencies between the parent and child RDD.

The `reduceByKey` operation is applied on rating data to compute the total number of ratings per movie. The `mapToPair` and `reduceByKey` operation sequence closely mimics the classic word count `MapReduce` program.

The `groupByKey` operation is used to group all movies for each genre. In Java version a custom type `MovieRatingCount` is used to store movie title and rating count pair. Note that in Java all custom types used in operations should be serializable. Spark by default uses Java serialization framework and can work with any class you create that implements `java.io.Serializable` interface. Spark can also use the Kryo library `https://github.com/EsotericSoftware/kryo` to serialize objects more quickly. In Python, the title and rating count pair is simply stored as a tuple, no special serialization is needed.

A `mapToPair` operation is applied on the result RDD of the `groupByKey` operation. Language build-in features such as Java's `Collections` class, or Python's `Sorted` function is used to sort all movies in a genre based on its rating count in descending order. The top 5 movies are returned as the final result.

For Java version, run `ant` command to compile the source code. The command will create a `sparkML.jar`. The jar file contains another executable: `MLGenreTopMoviesNaive`. This is the application that finds top movies per genre.

Submit this application to YARN, with the following command:

```
spark-submit  \
  --class ml.MLGenreTopMoviesNaive \
  --master yarn \
  --deploy-mode cluster \
  --num-executors 3 \
  sparkML.jar \
  /share/movie/ \
  week6-topmovies-java
```

Figure 8 shows the DAG involved in computing the final RDD.

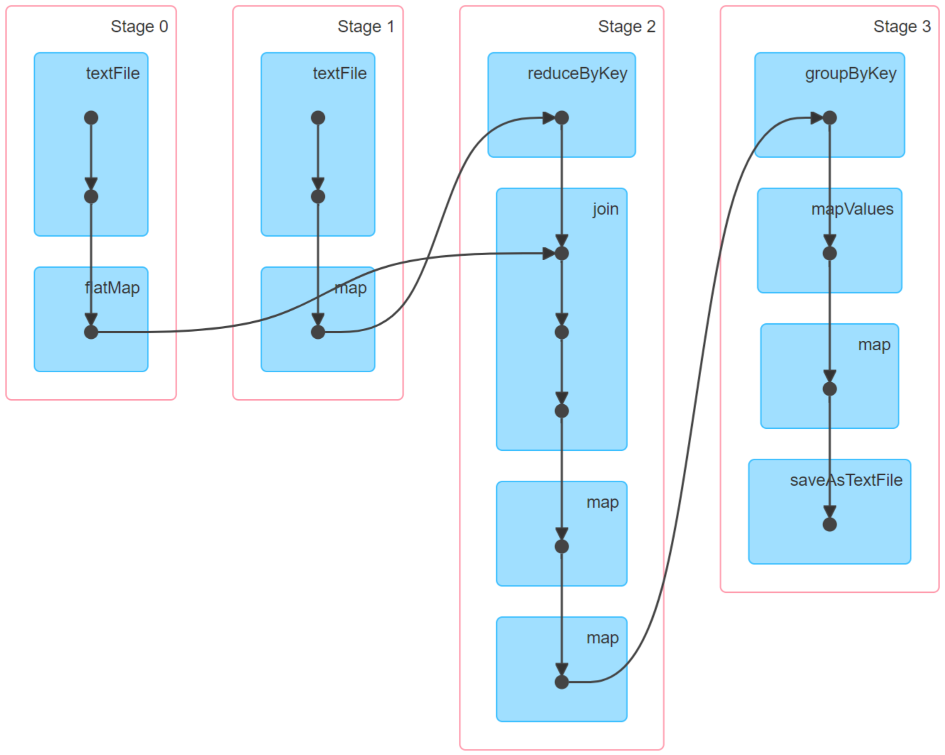


Figure 8: Spark DAG of Genre Top 5 Program

Inspect the history of this application to see how many jobs, stages and tasks there are. In particular, find out how many partitions are there in the `join` operations?

b) **Shuffling Behavior controlled by Partition Number** Modify the source code to make the `join` operation to have only one partition. Build the jar and re-run the application. Make sure you either delete the previous output directory or specify a different output directory for the second run. Inspect the application history to see how many stages and tasks there are. Explain the difference in stage and job numbers between the two runs.