



UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI

PROJECT

Apache STORM: WordCount Application

Subject: CLOUD AND BIG DATA

Professor

Prof. Daniel Hagimont

Students

RANSON Antoine
TRAN Thi Hong Hanh

February 17, 2020

1 Motivation

The objective of this project is to experiment WordCount application with Apache Storm given a Docker based environment, which allows us to simulate a cluster on our laptop. This environment allows making an evaluation of the scalability of Apache Storm.

2 Experiment

2.1 Prerequisite & Installation

- Install Storm

```
1 $ tar xzf apache-storm-0.9.5.tar.gz
```

- Install Apache Zookeeper

```
1 $ tar xzf apache-zookeeper-3.5.6-bin.tar.gz
```

2.2 WordCount Application

To run an application on Storm, we need to define a topology. A topology consists in entry point(s), called Spout, and one or several Bolts which are the data processing units. Here, we will deal with a very basic topology composed of 1 spout, and 2 bolts called Split and Count.

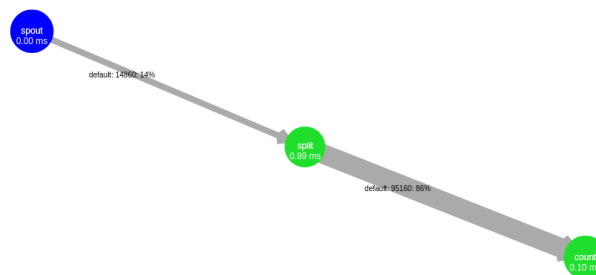


Figure 1: The topology of the project

As input we will use a list of sentences, from which we emit a random sentence on a regular time basis.

2.2.1 Local method

First of all, let us run the WordCount application, represented by the previous topology, locally:
Given the current directory is './project/workspace/wc/src/'.

- Compile the application:

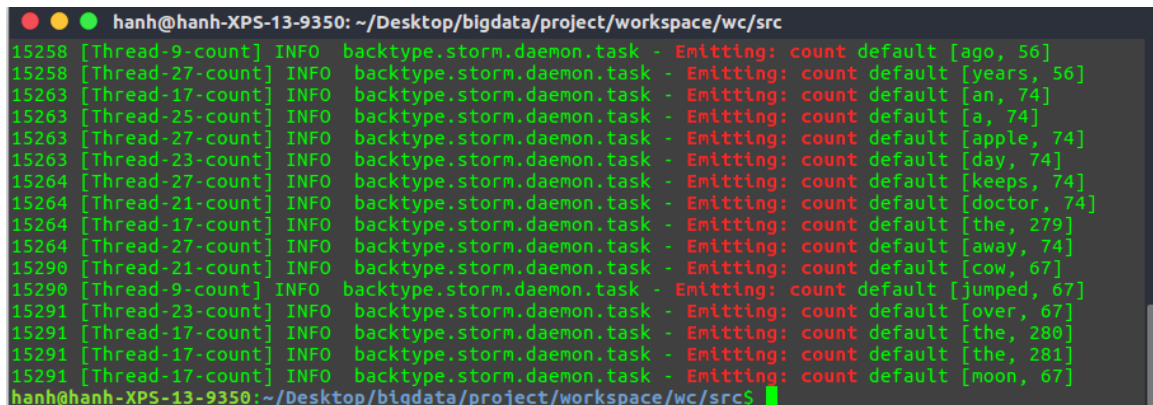
```
1 $ javac -cp "../..../apache-storm-0.9.5/lib/*" WordCountTopology.java
```

- Build jar file:

```
1 $ jar -cvf WordCountTopology.jar *.class
```

- Observe traces in debug mode:

```
1 $ java -cp "../..../apache-storm-0.9.5/lib/*" WordCountTopology | grep "Emitting: count"
```



```
hanh@hanh-XPS-13-9350: ~/Desktop/bigdata/project/workspace/wc/src
15258 [Thread-9-count] INFO backtype.storm.daemon.task - Emitting: count default [ago, 56]
15258 [Thread-27-count] INFO backtype.storm.daemon.task - Emitting: count default [years, 56]
15263 [Thread-17-count] INFO backtype.storm.daemon.task - Emitting: count default [an, 74]
15263 [Thread-25-count] INFO backtype.storm.daemon.task - Emitting: count default [a, 74]
15263 [Thread-27-count] INFO backtype.storm.daemon.task - Emitting: count default [apple, 74]
15263 [Thread-23-count] INFO backtype.storm.daemon.task - Emitting: count default [day, 74]
15264 [Thread-27-count] INFO backtype.storm.daemon.task - Emitting: count default [keeps, 74]
15264 [Thread-21-count] INFO backtype.storm.daemon.task - Emitting: count default [doctor, 74]
15264 [Thread-17-count] INFO backtype.storm.daemon.task - Emitting: count default [the, 279]
15264 [Thread-27-count] INFO backtype.storm.daemon.task - Emitting: count default [away, 74]
15290 [Thread-21-count] INFO backtype.storm.daemon.task - Emitting: count default [cow, 67]
15290 [Thread-9-count] INFO backtype.storm.daemon.task - Emitting: count default [jumped, 67]
15291 [Thread-23-count] INFO backtype.storm.daemon.task - Emitting: count default [over, 67]
15291 [Thread-17-count] INFO backtype.storm.daemon.task - Emitting: count default [the, 280]
15291 [Thread-17-count] INFO backtype.storm.daemon.task - Emitting: count default [the, 281]
15291 [Thread-17-count] INFO backtype.storm.daemon.task - Emitting: count default [moon, 67]
hanh@hanh-XPS-13-9350:~/Desktop/bigdata/project/workspace/wc/src$
```

Figure 2: The result of WordCount Application

Using the grep command allows us to check that the Word Count is executing correctly. What we can see on the picture above is that some words have the same count. This is normal, because our input datas are taken from a small list of sentences, and each sentences is emitted several times. So most of the words in one sentence are counted together, so their count are the same. But for some meaningless words, like "the", they appear in every sentences, so their count are different from the others.

2.2.2 First method

Then, we try to set up a Storm cluster by the following steps:

- First, we start the Apache Zookeeper. Since the Zookeeper "fails fast" it's better to always restart it:

```
1 $ docker run -d --restart always --name some-zookeeper zookeeper
```

- After that, we connect the Nimbus daemon with the Zookeeper:

```
1 $ docker run -d --restart always --name some-nimbus --link some-zookeeper:
zookeeper storm:0.9.7 storm nimbus
```

- Then, we start as many Supervisor node as required, because here we will use one Supervisor for one worker. The Supervisor will talk to the Nimbus and Zookeeper:

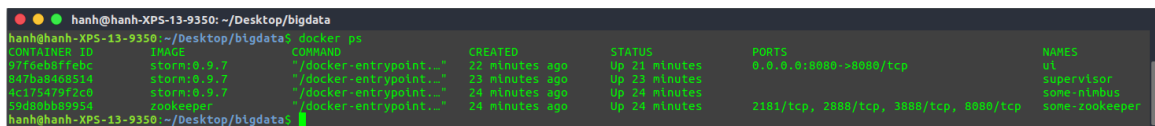
```
1 $ docker run -d --restart always --name supervisor --link some-zookeeper:
zookeeper --link some-nimbus:nimbus storm:0.9.7 storm supervisor
```

- We can submit a topology to our cluster with number of workers (for examples, 3):

```
1 $ docker run --link some-nimbus:nimbus -it --rm -v $(pwd)/WordCountTopology.
jar:/topology.jar storm:0.9.7 storm jar /topology.jar WordCountTopology
topology 3
```

- For better visualization, we run the Storm UI:

```
1 $ docker run -d -p 8080:8080 --restart always --name ui --link some-nimbus:
nimbus storm:0.9.7 storm ui
```



CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
9776e8a7fabc	storm:0.9.7	"/docker-entrypoint..."	22 minutes ago	Up 21 minutes	0.0.0.0:8080->8080/tcp	ui
447ba8468514	storm:0.9.7	"/docker-entrypoint..."	23 minutes ago	Up 23 minutes		supervisor
4c175479f2c0	storm:0.9.7	"/docker-entrypoint..."	24 minutes ago	Up 24 minutes		some-nimbus
59d88bb89954	zookeeper	"/docker-entrypoint..."	24 minutes ago	Up 24 minutes	2181/tcp, 2888/tcp, 3888/tcp, 8080/tcp	some-zookeeper

Figure 3: Number of Dockers we use in Storm clusters

2.2.3 Second method

For easier use of these commands above, given the current directory './project/workspace/', we write a **start-containers.sh** file which :

- Stop all the existing containers.
- Restart the network "storm-project", on which runs the cluster.
- Start a new zookeeper, a new nimbus called master, 1 or more slaves depending of the configuration wanted, and the Storm UI for better visualization.

```
1 $ ./start-containers.sh
```

Then the file run.sh is used to launch the application on the cluster.

```
1 $ ./run.sh
```

2.2.4 Third method

Then, we figure out the existence of **docker-compose** (Compose), which is a tool for defining and running multi-container Docker applications. A YAML file **docker-compose.yml** is used to configure our application's services. Then, with a single command, we create and start all the services from our configuration.

- Download the current stable release of Docker Compose (we use version of 1.25.0):

```
1 $ sudo apt-get install curl
2 $ sudo curl -L "https://github.com/docker/compose/releases/download/1.25.0/
3 docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

- Apply executable permissions to the binary:

```
2 $ sudo chmod +x /usr/local/bin/docker-compose
```

- Define services in **docker-compose.yml**.
- Build and run our app with Compose:

```
1 $ docker-compose up -d
```

- Submit the topology to our cluster with X workers:

```
1 $ docker run --network=project_storm_network --link nimbus:nimbus -it --rm --
v $(pwd)/workspace/WordCountTopology.jar:/topology.jar storm:0.9.7 storm jar
/topology.jar WordCountTopology topology 3
```

- Remove the containers entirely:

```
1 $ docker-compose down
```

3 Result

To check the scalability, let us check at the bolt's capacity depending on the architecture used to run the application. As we only have 4 cores at our disposal and the master needs one core to run, we will compare the capacity for each bolts between 1,2 and 3 cores. Here is a look at the Storm UI from where we extract the experimental values :

Storm UI

Cluster Summary

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.9.7	58m 19s	2	2	6	8	27	27

Topology summary

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks
topology	topology-1-1578122325	ACTIVE	56m 59s	2	27	27

Supervisor summary

Id	Host	Uptime	Slots	Used slots
7c0e054f-6855-45d4-96d0-27a504311a87	78d1429585e2	58m 10s	4	1
79fb3d3e-47c7-4148-bda3-1cad1d7c6bld	ecd253e9e324	58m 12s	4	1

Nimbus Configuration

Key	Value
dev.zookeeper.path	/tmp/dev-storm-zookeeper
drpc.chldopts	-Xmx768m
drpc.invocations.port	3773
drpc.port	3772
drpc.queue.size	128

Figure 4: The Storm UI home page

This is the home page. We can see the current cluster configuration running, the number of workers, and the nimbus configuration. To see the detail of the topology running on the cluster, click on the "topology" link :

Storm UI

Topology summary

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks
topology	topology-1-1578122325	ACTIVE	1h 3m 28s	2	27	27

Topology actions

[Activate](#)
[Deactivate](#)
[Rebalance](#)
[Kill](#)

Topology stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	3776460	2024060	0.000	0	0
3h 0m 0s	24498960	13136080	0.000	0	0
1d 0h 0m 0s	24498960	13136080	0.000	0	0
All time	24498960	13136080	0.000	0	0

Spouts (All time)

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Error Host	Error Port	Last error
spout	5	5	1775300	1774960	0.000	0	0			

Bolts (All time)

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Error Host	Error Port	Last error
_acker	2	2	140	0	0.000	0.000	0	0.000	0	0			
count	12	12	11361880	0	0.064	0.077	11361160	0.069	11361120	0			
split	8	8	11361640	11361120	0.034	0.559	1774980	0.558	1774960	0			

Topology Visualization

[Show Visualization](#)

Topology Configuration

Key	Value
-----	-------

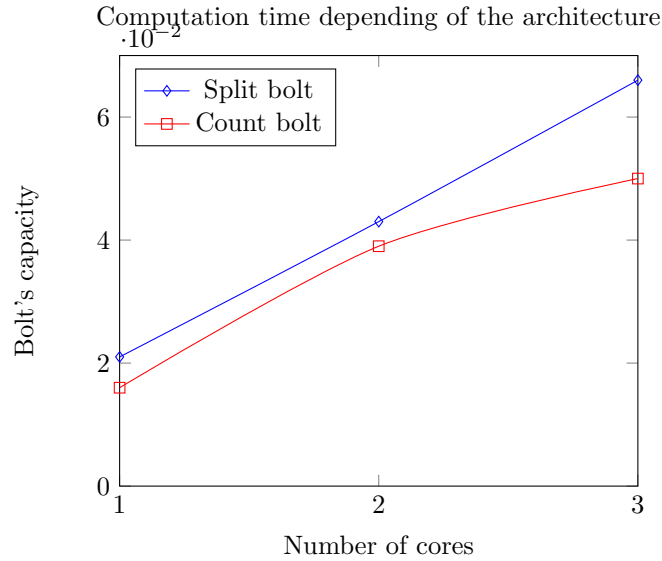
Figure 5: The storm UI topology's details page

On this mask, one can check all the details of the topology running on the cluster : the different spouts and bolts, their capacities and latencies, and the complete topology configuration. Here, we will only be interested by the capacity value for each bolt.

Architecture used	Capacity of the bolt split	Capacity of the bolt count
On a cluster with 1 worker (1 cores)	0.021	0.016
On a cluster with 2 workers (2 cores)	0.043	0.039
On a cluster with 3 workers (3 cores)	0.066	0.050

Table 1: Computational bolt capacity among different architectures.

The capacity is defined as the number of tuples executed times the average process latency, over a constant window of time. We can first notice that the capacity for the count bolt is always a bit less than the one of the split bolt. This is because when the measure of the capacity is taken, the split bolt processed N tuples. On these N tuples, the count bolt is still processing K tuples, so the capacity measurement for the two bolts are done on N and $(N-K)$ executed tuples each. Logically, the higher the capacity, the more data we processed, in the same amount of time. As increasing the number of workers increases the capacity of each bolt, we can conclude that the storm application is scalable.



We can observe that the count bolt's capacity tendency is incurving in the end. Probably because the amount of data to process is not big enough. Furthermore, the capacity is up-limited by one. If the capacity reaches 1, then the application requires more computational power, i.e requires to increase the number of workers.

4 Conclusion

In conclusion, we ran the Apache Storm application, using docker container's to emulate the zookeeper, nimbus and workers. The measurement of the bolt's capacity, visible on Storm UI, allowed us to demonstrate the scalability of this application. A futur work would be to re-evaluate this application with more computations power and bigger input datas.

References

- [1] https://hub.docker.com/_/storm/
- [2] <https://docs.docker.com/compose/>
- [3] <https://storm.apache.org/releases/current/Tutorial.html>
- [4] <https://github.com/apache/storm/blob/v0.9.7/examples/storm-starter/src/jvm/storm/starter/WordCountTopology.java>