Experience in distributed system, graph computation and comparing with GraphX

2018.05.13

Linna Hu, Hongyu Huang, Jiashuo Sun

AMS 560 project

Stony Brook University

Table of contents

[**I. Background and description of problems**](#_z6moobsbu7b) **4**

[1.1 The importance in graph computation and distributed system](#_504l6cgrqpxq) 4

[1.2 The shortness of GraphX and the improvement from TuX2](#_lfwtgi3dvrse) 5

[1.3 Main research objectives](#_njz9cxtxwxv3) 6

[**II. Methodology**](#_pmm7qcmqa803) **6**

[2.1 Set up and apply the distributed system](#_hux1r49ayf31) 6

[2.2 Do some interesting applications and understand GraphX](#_3no8ighr6q15) 6

[2.3 Shortness collection and comparison](#_hghm2x3jq4gv) 7

[2.4 Methods for optimization in GraphX.](#_wy72fqptk80h) 7

[**III. main work**](#_1tcaovhdhat1) **8**

[3.1 Build a new cluster](#_8fnmeoazog3s) 8

[3.1.1 Create an SSH key and a cluster](#_4my98m1a7v1x) 8

[3.1.2 SSH to a node in a cluster](#_9uhtuitwc6h0) 8

[3.2 Set up the distributed system](#_7tryfvre54k9) 8

[3.2.1 Deploy java environment](#_u2r4xc30aicj) 8

[3.2.2 Setup passwordless SSH](#_9gwqjb8ttdls) 9

[3.2.3 Download and Extract Apache Hadoop](#_nslljfobj7o8) 9

[3.2.4 Setup HDFS](#_xbcv8geu3g9j) 10

[3.2.6 Test Yarn](#_52q2ym6hoal2) 13

[3.3 Set up Spark](#_o0p7xv8f9j) 13

[3.3.1 Setup Spark on Yarn](#_3ugksbr0ecqc) 13

[3.4 Import GraphX and the usage](#_u3o6hgxjlre5) 15

[Example Property Graph](#_ke2tkgc2qw68) 15

[3.5 Applications on GraphX](#_yf4bcch1o17b) 18

[**IV. Results**](#_35j7vyhct2wp) **20**

[4.1 The result of test Yarn](#_kvjd4clmy0ep) 20

[4.2 The result of test Spark](#_c2r3px52nxda) 20

[4.3 The result of GraphX application](#_gnp05mop42vo) 22

[4.4 The optimization of TUX2](#_1apvcggcwp2k) 23

[4.4.1 Stale Synchronous Parallel](#_kqwuy22rw1op) 23

[4.4.2 Heterogeneity](#_6u08slmxssen) 23

[4.4.3 Mini-Batch](#_p0uk2og4joga) 24

[4.5 The comparison of TuX2 and GraphX](#_tz1ce9kfxrwc) 24

[4.5.1 Efficiency of processing data](#_czq825g8bnuy) 25

[4.5.2 Partitioning Strategies](#_w4g6x39beql5) 25

[4.5.3 Algorithms](#_ax8me1b3krqr) 26

[**V. Evaluation**](#_jk9e813l6i34) **27**

[5.1 Discussion of GraphX](#_eat8ctjzsx0k) 27

[5.2 Discussion of TuX2](#_lnhur54jb3pi) 27

[5.3 Ideas to improve](#_xro3i4arawms) 28

[5.3.1 For GraphX](#_2hpyzbxqtmdm) 28

[5.3.2 For TuX2](#_45b6dvctlesj) 28

[**VI. Challenges and Future Work**](#_7peoufj2iy7) **28**

[6.1 Problems that we have met but have been fixed](#_1g9iwoo576j) 28

[6.1.1 Yarn didn’t run](#_5ypsbwfhhx6l) 28

[6.1.2 GraphX computation](#_lqoi857alf0a) 29

[6.2 Problems that still exists](#_xin5kylwijuy) 29

[6.2.1 Spark runs workload just on one node](#_fypx33vgnkpe) 29

[6.2.2 GraphX needs to run in distributed system](#_wca0cd9ljyyg) 30

[6.3 Future work](#_scbgz7nr0eg8) 30

[**VII. References**](#_5ohsneh88fhu) **30**

# 

# 

# I. Background and description of problems

## 1.1 The importance in graph computation and distributed system

Large-scale graph processing is an important part of data infrastructure services in our daily life. Distributed graph engines, such as PowerGraph, embrace a vertex-program abstraction to express iterative computation over large-scale graphs. A graph engine effectively encodes an index of the data in a graph structure to expedite graph-traversal-based data access along edges, and supports elegant graph computation models such as Gather-Apply-Scatter (GAS) for ease of programming. The developing scalable and efficient graph engines through data layout, partitioning, scheduling, and balanced parallelism have been shown that the distributed graph engines can scale to graphs with more than a trillion edges for simple graph algorithms such as PageRank.

However, most subsequent work on graph engines adopts a simplistic graph computation model, driven by basic graph benchmarks such as PageRank. The resulting graph engines lack flexibility and other key capabilities for efficient distributed machine learning.

is a new distributed graph engine that bridges graph computation and distributed machine learning. inherits the benefits of an elegant graph computation model, efficient graph layout, and balanced parallelism to scale to billion-edge graphs; we extend and optimize it for distributed machine learning to support heterogeneity, a Stale Synchronous Parallel model, and a new MEGA (Mini-batch, Exchange, GlobalSync, and Apply) model.

Our goal used to make some optimization on , but it’s not an open source. We have to switch our objective to other graph engines. Compared with other graph engines, GraphX shows a lot good performances when processing dataset. GraphX is the new (alpha) Spark (written in Scala) API for graphs and graph-parallel computation. At a high-level, GraphX extends the Spark RDD by introducing the Resilient Distributed Property Graph: a directed multigraph with properties attached to each vertex and edge. Spark GraphX offered several features that made the development of graph data processing solutions easier.GraphX allows graphs to be read from Hive using an SQL-like query, allowing arbitrary column transformations.

## 1.2 The shortness of GraphX and the improvement from TuX2

Even though GraphX can handle large graph size, there is a large gap in efficiency in terms of machine hours required, and its performance was not stable. Here are some shortness we can tell.

**The shortness of GraphX(standard graph engines):**

*Data models:* Standard graph engines assume a homogeneous set of vertices, but the graphs often have different types of vertices playing distinct roles. GraphX adopts a vertex-cut approach to distributed graph partitioning.

*Programming models:* The standard GAS model is unable to express the computation patterns which involve multiple rounds of propagations between different types of vertices.

*Execution scheduling:* Machine learning algorithms typically describe the process to converge to a “good” solution according to an objective function and the convergence process itself is robust to variations and slack that can be leveraged to improve efficiency and parallelism. Graph engines typically operate on individual vertices, or define an “iteration” or a batch on the entire graph.

**The improvement from TuX2:**

Compared to implementations on distributed machine learning platforms, hides the detailed management of data layout, partitioning, and parallelism from developers. Here are some advantages we can tell.

*Heterogeneous Data Layout:*  supports heterogeneity in multiple dimensions of data layout, including vertex type and partitioning approach and also supports heterogeneity between master and mirror vertex data types.

*Scheduling with SSP:*  supports the Stale Synchronous Parallel (SSP) model with bounded staleness and mini-batches. SSP is based on the notion of work-per-clock, where a clock corresponds to an iteration over a mini-batch executed by a set of concurrent tasks. SSP introduces an explicit slack parameter, which specifies in clocks how stale a task’s view of the globally shared state can be.

*MEGA Model in :*  It introduces a new stage-based MEGA model, where each stage is a computation on a set of vertices and their edges in a graph. supports four types of stage: Mini-batch, Exchange, GlobalSync, and Apply.

## 1.3 Main research objectives

Since is not an open source, we can’t optimize it directly. So our goal switches to “when using GraphX, one of the most efficient graph engines, to handle graphs, can we find some aspects that has not tried to or succeed to improve yet ? ”

As our goal is to find the shortness of GraphX in the best possible way, we decided to do some quantitative and qualitative work evaluations on GraphX. Our work includes:

* Implement some interesting applications on GraphX to understand the graph handling procedure.
* Compare the properties of GraphX with , pick up some of them that has not involved.
* Raise up some suggestions of GraphX from the experiments we did.
* List out the methods to optimize GraphX, where has not referred to.

# II. Methodology

The method best suited for our research objective, due to the flexibility and immediacy of response, was the experience in distributed system and run some examples on GraphX.

## 2.1 Set up and apply the distributed system

The Hadoop provided HDFS and YARN, at first we need to build a cluster on Cloudlab.us and set up the environment of distributed system we uploaded file to HDFS and run Spark on the top of YARN using the file. An experiment sample (movie information) sample was drawn from the sample frames by listing the information of users’ ID numbers, the ratings of movies and the ID of movies. We run this sample on Spark using only a master node or all three nodes(including two slave nodes) to see the difference of single machine processing work and distributed system handling workload. Those experiments could help us have a deeper understanding of distributed system.

## 2.2 Do some interesting applications and understand GraphX

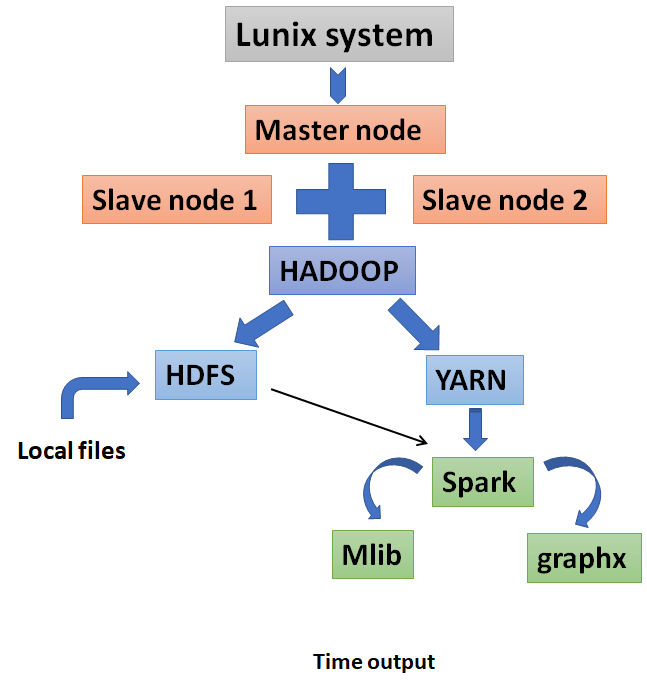
GraphX provided scala language, we can build the model by ourselves to generate different kinds of algorithms that we want. For this part, we choose a simple sample. And we should use graphloader to load our dataset into graph. After that, we need to initialize our vertices as a vacant map. Then we are supposed to use aggregateMessages to transfer VerticeID and totalRounds out to degree point. Output of the degree point combines the information into a big MAP. The updated vertices will join the original graph, this aims to update the changed properties in original graph. Then repeat the steps, we will get an output of the relationship of vertices. This will help us understand how GraphX handle dataset and get a deeper understanding of GraphX.

## 2.3 Shortness collection and comparison

After the experiments of using GraphX, we get a better understanding in graph computation, then we can tell some limitations of GraphX when we using it. We are supposed to do some comparisons with .Then we should pick up some aspects that has not involved.

## 2.4 Methods for optimization in GraphX.

This part is aiming at searching online and read papers to find out the ideas that could make some progress on GraphX. Combined with the limitations that we have met during the experiments, we can tell where needs to be optimized. According to the paper in , we can list out the aspects that has not referred to. Then we are supposed to arise the ideas to make some progress in GraphX.



# III. main work

## 3.1 Build a new cluster

### 3.1.1 Create an SSH key and a cluster

Once we have an account on cloudlab.us, we are supposed to get ssh key to upload on the website to start our new experiment to set up a cluster.

*ssh-keygen -t rsa -b 4096 -C "*[*your\_email@example.com*](mailto:your_email@example.com)*”*

### 3.1.2 SSH to a node in a cluster

In the cloudlab.us, we can check our view list of our cluster to see our ssh key for each node.

*ssh -p 22 @ms0826.utah.cloudlab.us*

*ssh -p 22 @ms0824.utah.cloudlab.us*

*ssh -p 22 @ms0830.utah.cloudlab.us*

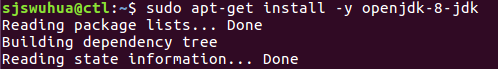
We can just use this command on our local host node, we can switch to the cluster.

## 3.2 Set up the distributed system

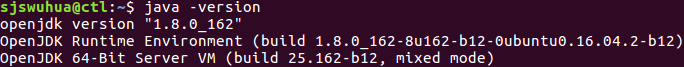
### 3.2.1 Deploy java environment

As prerequisites of setup Apache Hadoop Yarn, we need to need install JDK first.

*sudo apt-get install -y openjdk-8-jdk*

**

*java -version*



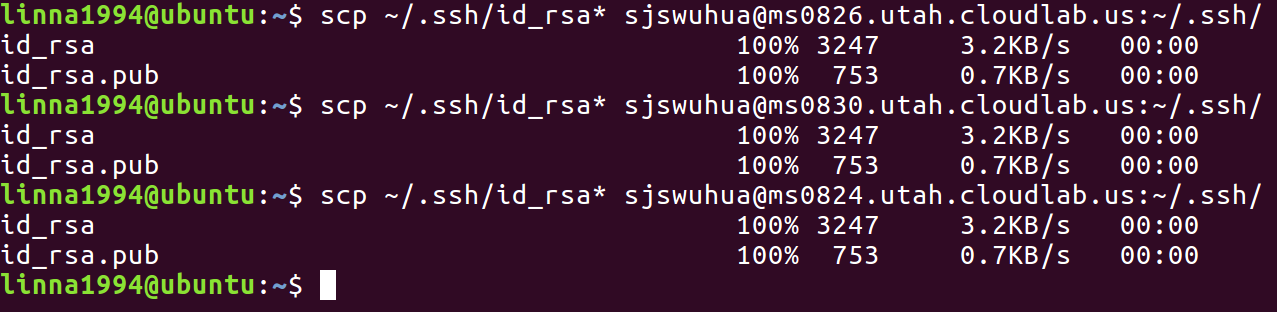
### 3.2.2 Setup passwordless SSH

We have already created a passwordless SSH in local host, now we need to upload the key to three nodes.

*scp ~/.ssh/id\_rsa\* sjswuhua@ms0826.utah.cloudlab.us:~/.ssh/*

*scp ~/.ssh/id\_rsa\* sjswuhua@ms0830.utah.cloudlab.us:~/.ssh/*

*scp ~/.ssh/id\_rsa\* sjswuhua@ms0824.utah.cloudlab.us:~/.ssh/*

**

Than we set the modify the permission and add related paths

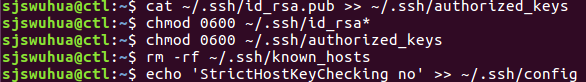
*cat ~/.ssh/id\_rsa.pub >> ~/.ssh/authorized\_keys*

*chmod 0600 ~/.ssh/id\_rsa\**

*chmod 0600 ~/.ssh/authorized\_keys*

*rm -rf ~/.ssh/known\_hosts*

*echo 'StrictHostKeyChecking no' >> ~/.ssh/config*

**

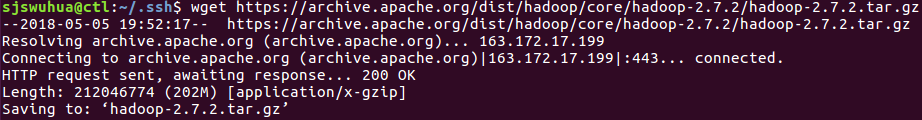
Further, we check the “config” file to ensure the setup.



### 3.2.3 Download and Extract Apache Hadoop

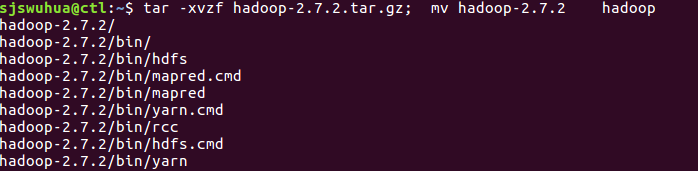
We download Apache Hadoop 2.7.2 from website for each node.

*wget https://archive.apache.org/dist/hadoop/core/hadoop-2.7.2/hadoop-2.7.2.tar.gz*



Then extract the .gz file and rename the directory in each node.

*tar -xvzf hadoop-2.7.2.tar.gz; mv hadoop-2.7.2 hadoop*



### 3.2.4 Setup HDFS

In each node, setup Hadoop paths.

*export JAVA\_HOME=/usr/lib/jvm/java-8-openjdk-amd64/ >> .bashrc*

*echo export HADOOP\_PREFIX=~/hadoop >> .bashrc;*

*echo export HADOOP\_YARN\_HOME=~/hadoop >> .bashrc;*

*echo export HADOOP\_HOME=~/hadoop >> .bashrc;*

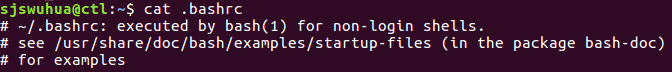
*echo export HADOOP\_CONF\_DIR=~/hadoop/etc/hadoop >> .bashrc;*

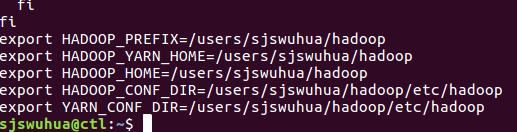
*echo export YARN\_CONF\_DIR=~/hadoop/etc/hadoop >> .bashrc;*

*source .bashrc*

*sed -i '1iexport JAVA\_HOME=/usr/lib/jvm/java-8-openjdk-amd64' hadoop/etc/hadoop/hadoop-env.sh*

Still, we check each file to make our setup is right.





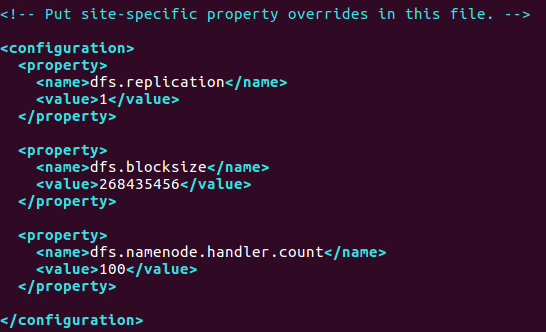
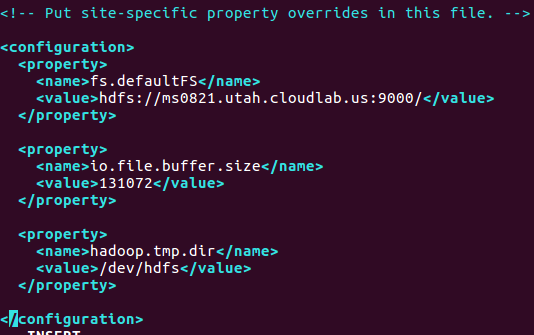


Then, we create HDFS fold and grant permission to read and write.

*sudo mkdir /dev/hdfs; sudo chmod 777 /dev/hdfs*



Then, on each node, we edit core-site.sh and hdfs-site.sh files.



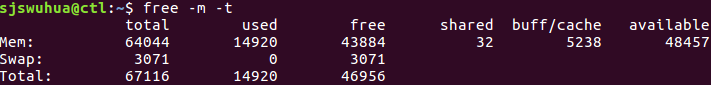
3.2.5 Setup Yarn

In order to setup yarn, first we need to know the number of CPU cores and amount of memory on each node.

*cat /proc/cpuinfo| grep "cpu cores"| uniq*

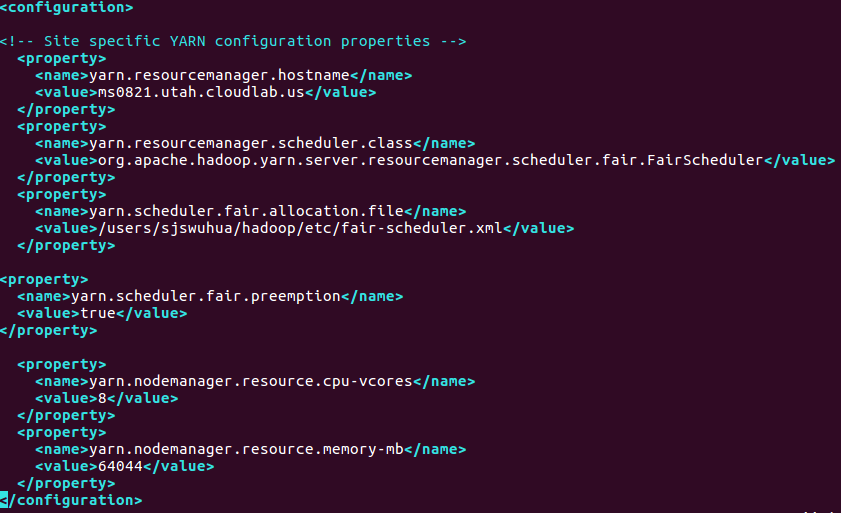


*free -m -t*

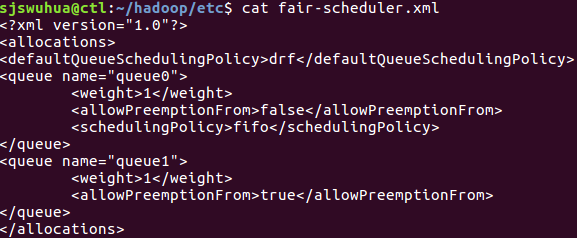


Our nodes all have 8 CPU cores and the amount of memory is 64044 MB.

Then, we edit yarn-site.sh file on each node.



And, we create fair-scheduler.xml file and edit it on each node.

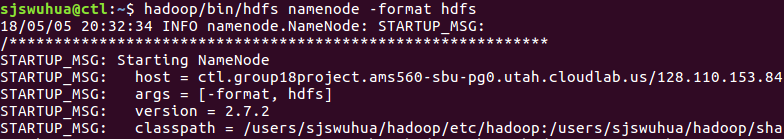


Meanwhile, we create slaves file as follow:



At last, on the master node, we format HDFS.

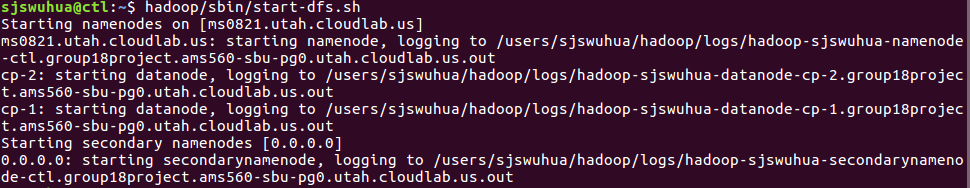
*hadoop/bin/hdfs namenode -format hdfs*



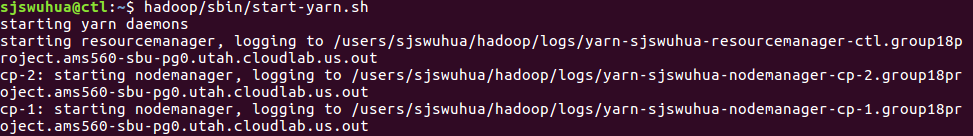
### 3.2.6 Test Yarn

First, we start HDFS and Yarn operating on master node.

*hadoop/sbin/start-dfs.sh*



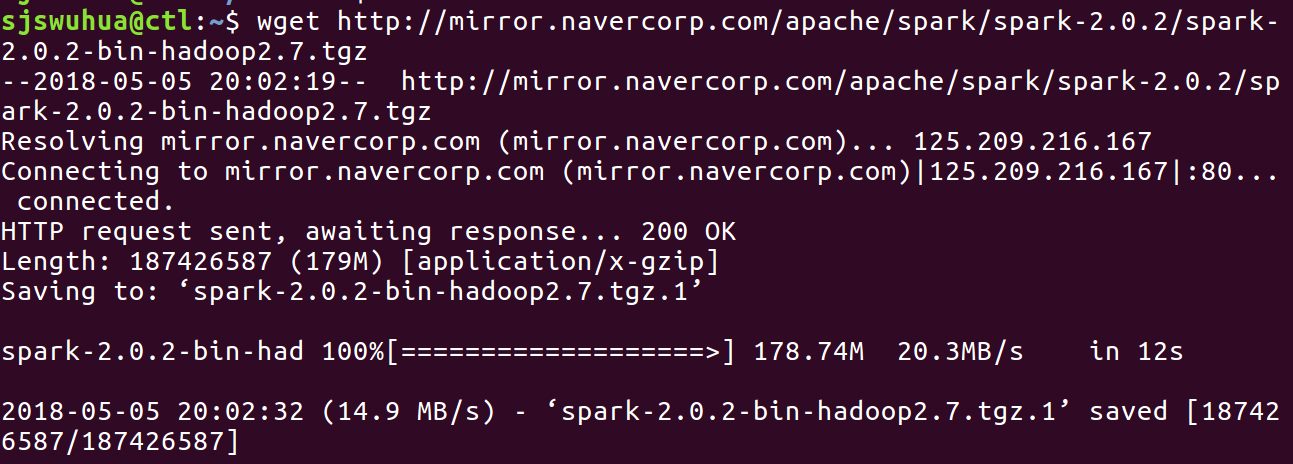
*hadoop/sbin/start-yarn.sh*



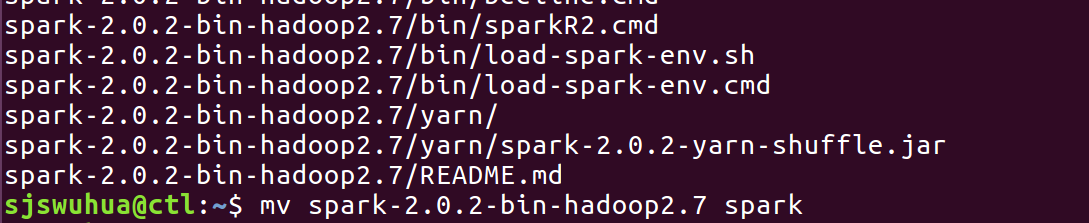
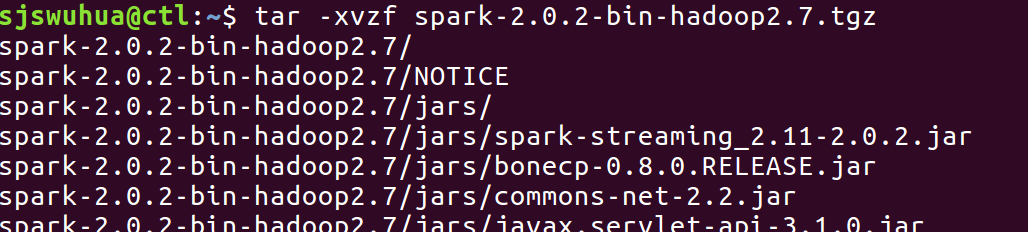
## 3.3 Set up Spark

### 3.3.1 Setup Spark on Yarn

* On the each nodes, we first download Spark.



* Then we decompression.

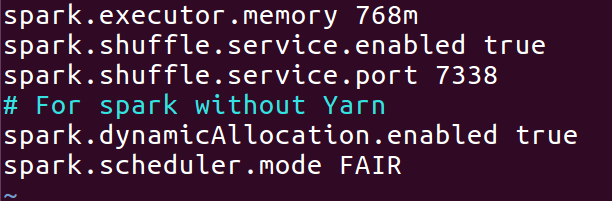


* Then we create and edit “slaves” that contains slave nodes.

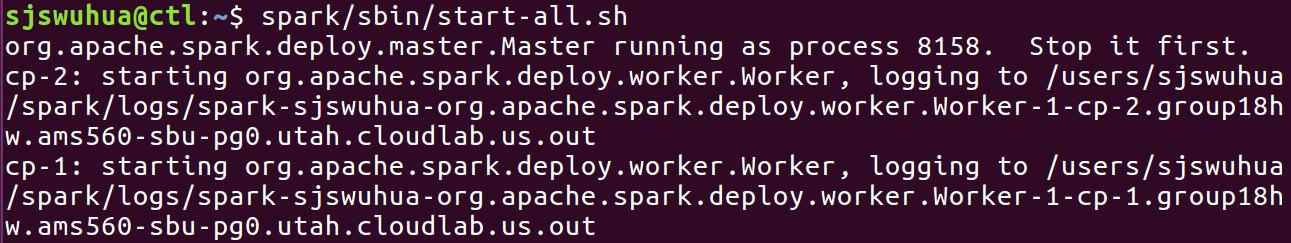




On each nodes, we create “spark-defaults.conf” at ~/spark/conf



* Then we start spark:



* We use a script to run.

## 3.4 Import GraphX and the usage

To get started you first need to import Spark and GraphX into your project, as follows:

**import** **org.apache.spark.\_**  
**import** **org.apache.spark.graphx.\_**  
*// To make some of the examples work we will also need RDD*  
**import** **org.apache.spark.rdd.RDD**

GraphX optimizes the representation of vertex and edge types when they are primitive data types (e.g., int, double, etc…) reducing the in memory footprint by storing them in specialized arrays.

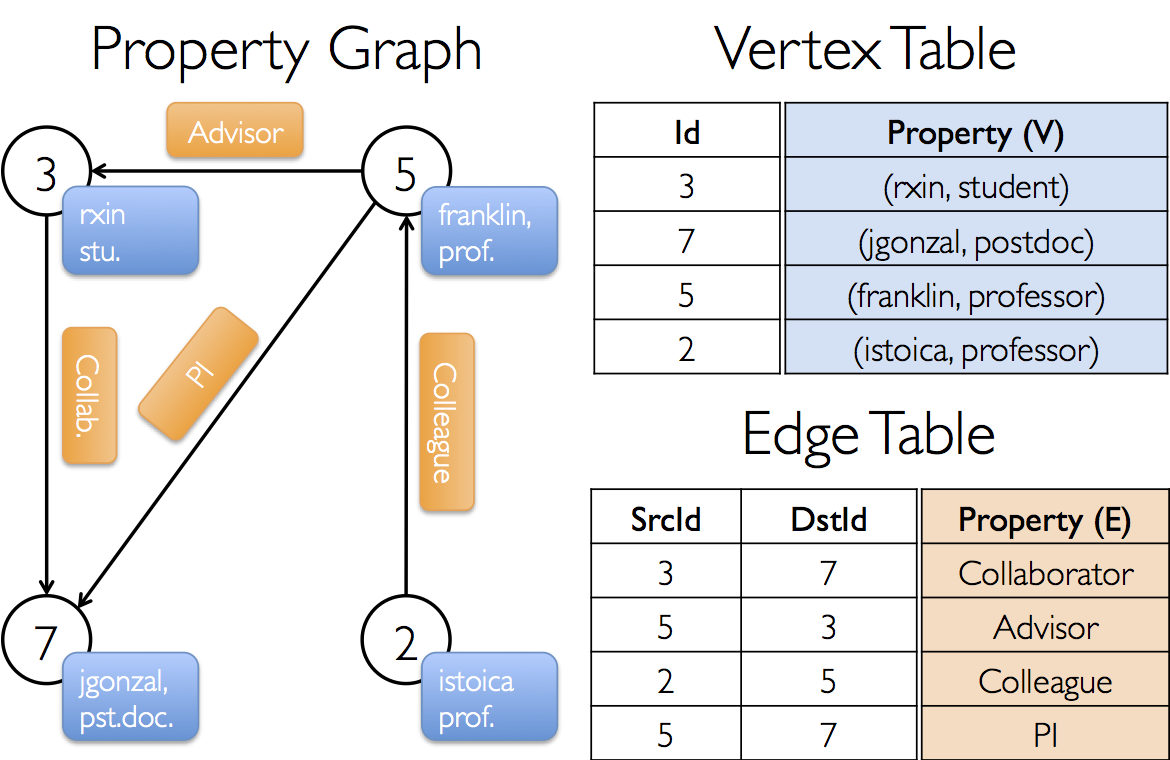
In some cases it may be desirable to have vertices with different property types in the same graph. This can be accomplished through inheritance. For example to model users and products as a bipartite graph we might do the following:

**class** **VertexProperty**()  
**case** **class** **UserProperty**(**val** name**:** String) **extends** **VertexProperty**  
**case** **class** **ProductProperty**(**val** name**:** String, **val** price**:** Double) **extends** **VertexProperty**  
*// The graph might then have the type:*  
**var** graph**:** Graph[VertexProperty, String] **=** **null**

### 

### Example Property Graph

Suppose we want to construct a property graph consisting of the various collaborators on the GraphX project. The vertex property might contain the username and occupation. We could annotate edges with a string describing the relationships between collaborators:

****

The resulting graph would have the type signature:

**val userGraph: Graph[(String, String), String]**

There are numerous ways to construct a property graph from raw files, RDDs, and even synthetic generators and these are discussed in more detail in the section on graph builders. Probably the most general method is to use thegraph object. For example the following code constructs a graph from a collection of RDDs:

***// Assume the SparkContext has already been constructed*  
val sc: SparkContext  
*// Create an RDD for the vertices*  
val users: RDD[(VertexId, (String, String))] =  
 sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),  
 (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))  
*// Create an RDD for edges*  
val relationships: RDD[Edge[String]] =  
 sc.parallelize(Array(Edge(3L, 7L, "collab"), Edge(5L, 3L, "advisor"),  
 Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))  
*// Define a default user in case there are relationship with missing user*  
val defaultUser = ("John Doe", "Missing")  
*// Build the initial Graph*  
val graph = Graph(users, relationships, defaultUser)**

We can deconstruct a graph into the respective vertex and edge views by using the graph.vertices and graph.edges members respectively.

**val graph: Graph[(String, String), String] *// Constructed from above*  
*// Count all users which are postdocs*  
graph.vertices.filter { case (id, (name, pos)) => pos == "postdoc" }.count  
*// Count all the edges where src > dst*  
graph.edges.filter(e => e.srcId > e.dstId).count**

Join operations

In many cases it is necessary to join data from external collections (RDDs) with graphs. For example, we might have extra user properties that we want to merge with an existing graph or we might want to pull vertex properties from one graph into another. These tasks can be accomplished using the *join* operators. Below we list the key join operators:

**class** **Graph**[VD, ED] {  
 **def** joinVertices[U](table**:** RDD[(VertexId, U)])(map**:** (VertexId, VD, U) **=>** **VD**)  
 **:** Graph[VD, ED]  
 **def** outerJoinVertices[U, VD2](table**:** RDD[(VertexId, U)])(map**:** (VertexId, VD, **Option**[U]) **=>** **VD2**)  
 **:** Graph[VD2, ED]  
}

The core aggregation operation in GraphX is [aggregateMessages](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.graphx.Graph@aggregateMessages%5BA%5D((EdgeContext%5BVD,ED,A%5D)%E2%87%92Unit,(A,A)%E2%87%92A,TripletFields)(ClassTag%5BA%5D):VertexRDD%5BA%5D). This operator applies a user defined sendMsg function to each *edge triplet* in the graph and then uses the mergeMsg function to aggregate those messages at their destination vertex.

**class** **Graph**[VD, ED] {  
 **def** aggregateMessages[Msg: ClassTag](  
 sendMsg**:** EdgeContext[VD, ED, Msg] **=>** **Unit**,  
 mergeMsg**:** (Msg, Msg) **=>** **Msg**,  
 tripletFields**:** TripletFields = **TripletFields**.**All**)  
 **:** VertexRDD[Msg]  
}

Output

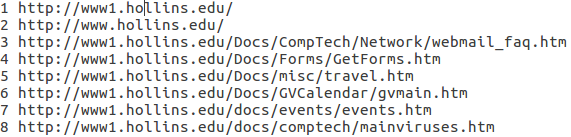
val result = roundVertices.map(\_.\_1).collect

## 3.5 Applications on GraphX

**Dataset: Kenneth Massey's Information Retrieval webpages**

The dataset come from a web crawler accessing web pages[6,7]. It recorded visit information and merge the final results. There are 6012 web pages and 23875 visits between each other.

We treat all the web pages as vertices recording in the following form:



And treat the visit information as edges recording in the following form:

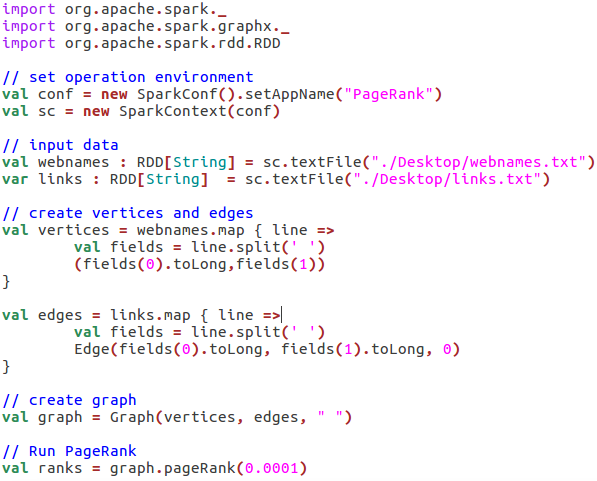


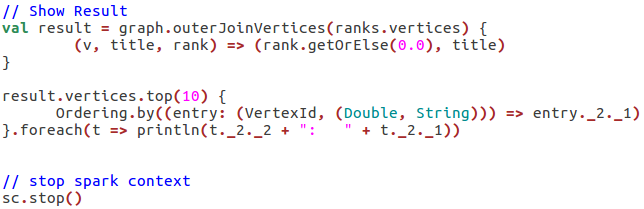
In this way, we create a graph with vertices and edges.

**Algorithm: PageRank**

We use PageRank to deal with the graph we create above. Through it, we try to find the most contributing and most visited web pages. Algorithm accuracy set to be 0.0001. The Program will eventually print out the top ten most visited web pages and their ranks.

**Code**





All these scripts are using scala programming language and run in the spark-shell environment.

# IV. Results

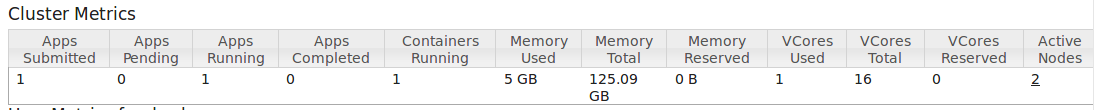
## 4.1 The result of test Yarn

As we can tell, after setting up Yarn the workload could run successfully on Yarn. This is an important part for we will run spark on Yarn to make our distributed graph computation come true. The result is the following:

* From http://ms0826.utah.cloudlab.us:50070/, we can check the HDFS status.



* And from http://ms0826.utah.cloudlab.us:8088/, we can check yarn status.



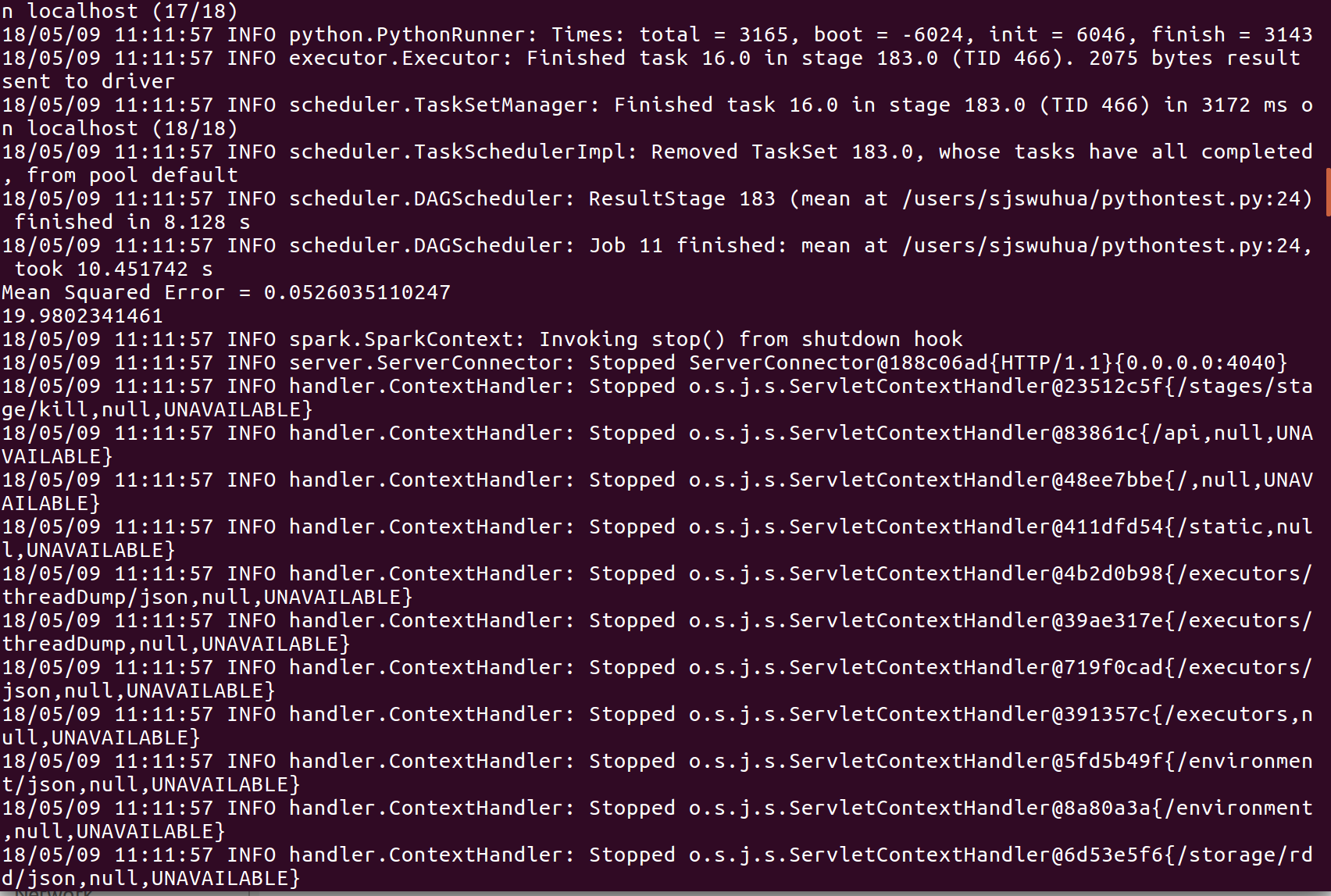
This result shows that our Yarn was set up successfully.

## 4.2 The result of test Spark

After we set up **Spark** and Mapreduce on Yarn. We need to run the pyspark in the spark, and import dataset and address dataset on the pyspark.We first uploaded the dataset to the hdfs and run the script that we edited in the pyspark to address the dataset.

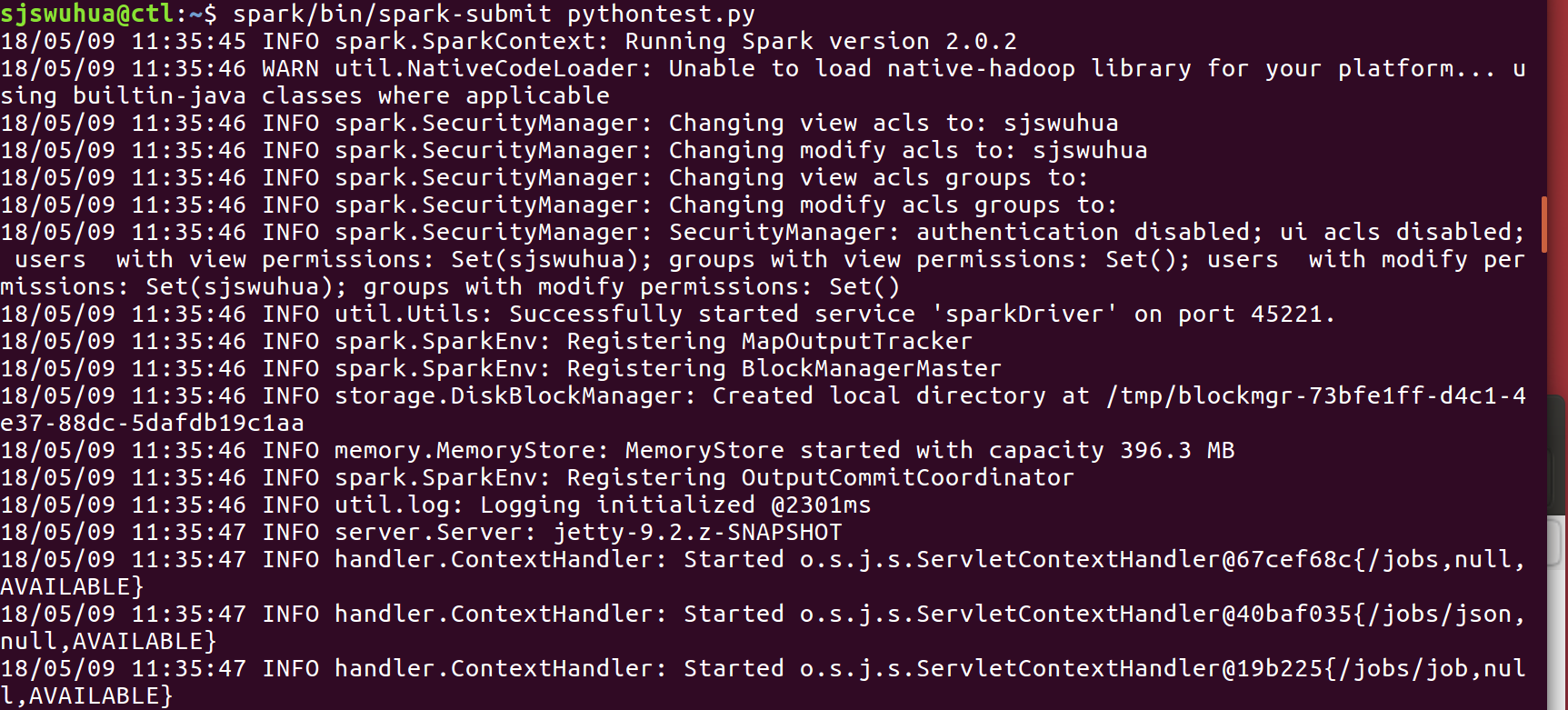
We want to compare the time that running application just in ctl to the time running in all of nodes, 2 slaves and a master.

* When we run 1MB dataset, the result is following:



We can see that the running time is 19.9802341461.

* Then when we run a 10MB dataset. The result is following:



* In the website of Spark, we can see the working status.

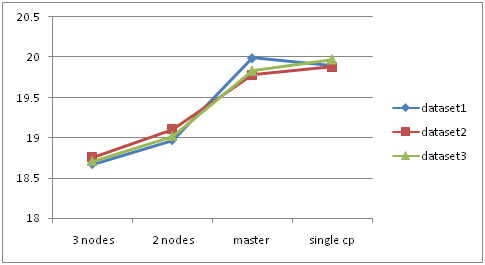
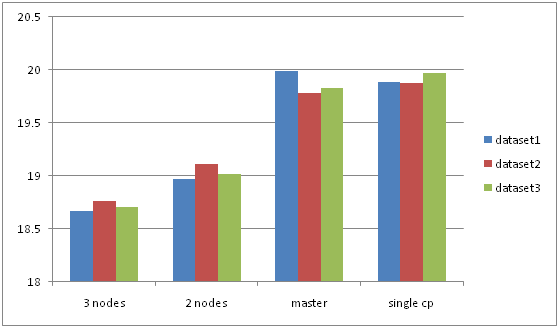


* The completed status.

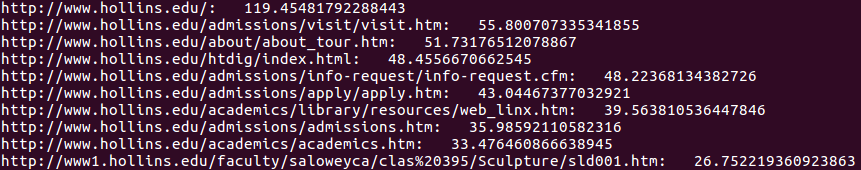


* The runtime when we use distributed system and standalone spark.

For this part, we uploaded 1 MB dataset on HDFS, and then we run spark at 3 nodes, 2 nodes, one master and single machine to record the runtime. The following is what we get:



## 4.3 The result of GraphX application



We create the graph with vertices which are 6012 web pages and edges which are 23875 visits of a web crawler.

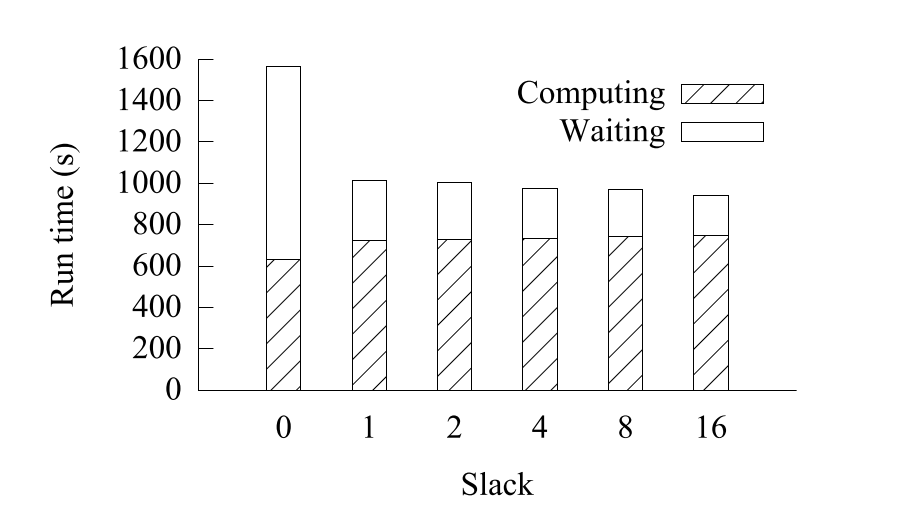
After using PageRank, we have the top ten most visited web pages and their ranks.

It is not difficult to see that the websites’ official web page is the most contributing one, which is also in line with our expectation.

## 4.4 The optimization of

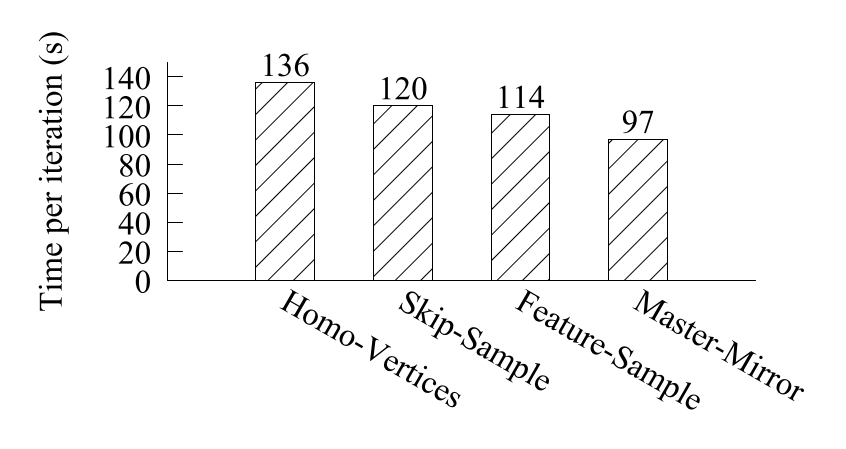
### 4.4.1 Stale Synchronous Parallel

supports the configuration of slack as a staleness bound for SSP, to allow users to tune the parameter for desirable convergence. The effect of slack varies by algorithm.



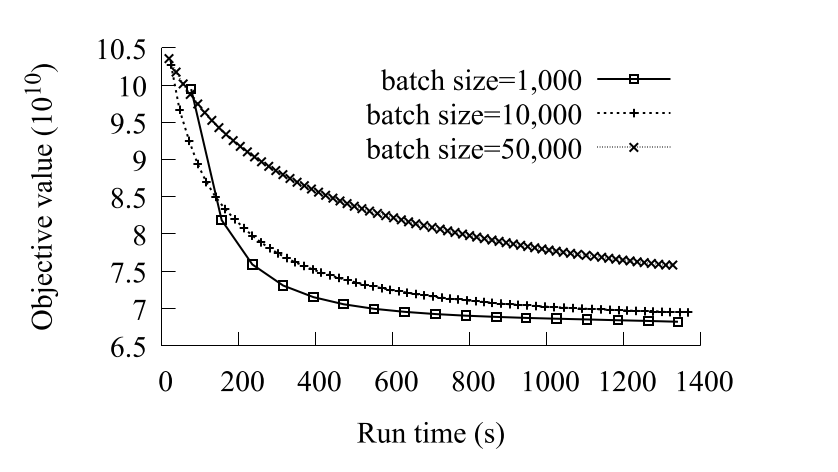
As shown in this figure, the overall convergence accelerates as the slack increases. The breakdown confirms that increasing slack reduces waiting time, while increasing computing time only slightly. Indicating that it takes about a slightly larger number of iterations to reach the same convergence point.

### 4.4.2 Heterogeneity



As we can see in this figure, if they model all vertices as the same vertex type, each iteration takes 136s (Homo-Vertices in this figure). For algorithm (BlockPG), because only feature vertices have mirrors, they can specify to enumerate only feature vertices in each mini-batch. This setup leads to a reduction of 16s per iteration (SKip-Sample). And, if they define different vertex data types for features and samples for a more compact representation, each iteration can save an additional 6s (Feature-Sample). Then they can allow masters and mirrors to have different types and can indicate which data need to be synchronized.

### 4.4.3 Mini-Batch



This figure shows the convergence to objective values over time with slack. It shows each iteration as a point on the corresponding curve to demonstrate the effect of mini-batch size on the execution time of each iteration.

## 4.5 The comparison of and GraphX

After applying some examples on GraphX, we get a better understanding in GraphX and graph computation. But before we doing the comparison of and GraphX, we have to notice that is not an open source, so we have to make this happen by just finding some connections between them. The first thing we think is the best way is to figure out the efficiency of processing dataset. Although we cannot compare them directly, there are many resources showing some comparisons in and GraphX separately with other graph engines.

### 4.5.1 Efficiency of processing data

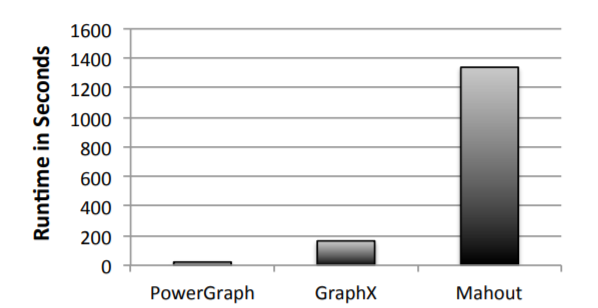


Figure1. Runtime of using PowerGraph, GraphX and Mahout to process datasets.

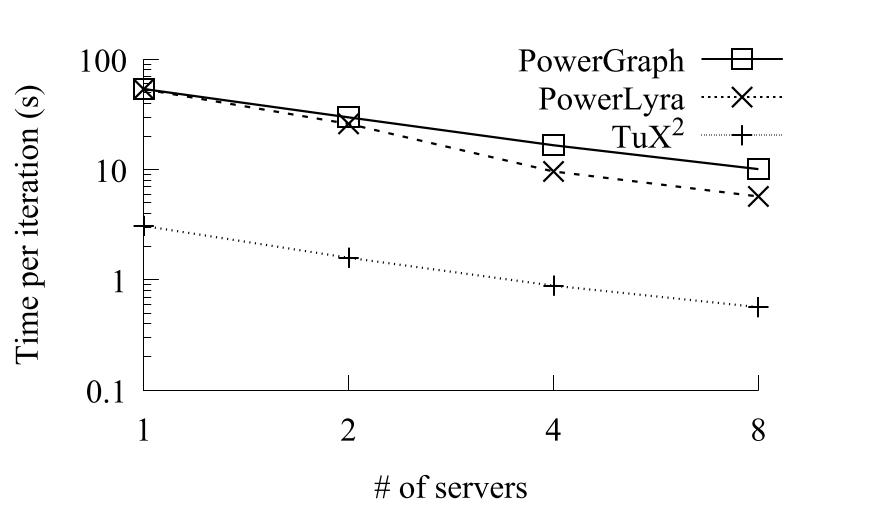


Figure 2. The runtime of using , PowerGraph and PowerLyra to process iteration.

From these two figures, we can tell that run less time when processing dataset. Just from this point, did better job on processing data.

### 4.5.2 Partitioning Strategies

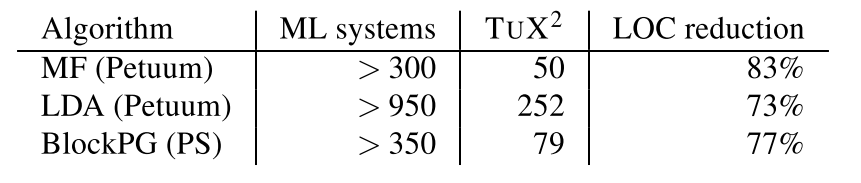
To distribute graph computation over multiple machines in a cluster, the input graph first needs to be partitioned before computation starts by assigning graph elements to individual machines.

The partitions created have a significant impact on the performance and resource usage in the computation stage. To avoid excess communication between different partitions during computation, systems typically use vertex mirroring, whereby some vertices may have images in multiple partitions. If a partitioning strategy results in a large number of mirrors, then is will lead to higher communication costs, memory usage, and synchronization costs which lead to higher job completion times.

Graph partitioning itself must also be fast and efficient; for some graph applications, the time it takes to load and partition the graph can be much larger than the time it takes to do the actual computation.[5]

|  |  |
| --- | --- |
| System | Partitioning Strategies |
| PowerGraph | Random, Grid, Oblivious, HDRF, PDS |
| PowerLyra | Random, Grid, Oblivious, Hybrid, Hybrid-Ginger, PDS |
| GraphX | Random, Canonical Random, 1D, 2D |
|  | only talk about ***vertex-cut*** approachin the paper |

### 4.5.3 Algorithms

* Algorithms in 

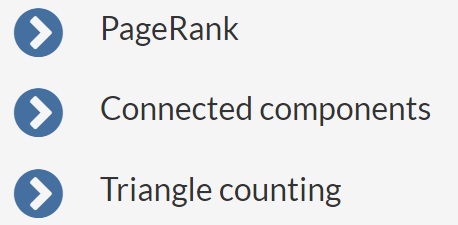
**Matrix Factorization (MF):**  models training data as a bipartite graph with users and items being vertices and user-item ratings being edges, and solves MF using SGD. They use stageSequenceBuilder, ExchangeStage, ApplyStage, and GlobalSyncStage.Exchange to improve.

**Latent Dirichlet Allocation (LDA):**  implements SparseLDA, a widely used algorithm for large-scale distributed LDA training.

**Block Proximal Gradient (BlockPG):** BloackPG is a state-of-the-art logistic regression algorithm. It randomly divides features into blocks and enumerates each block as a mini-batch.

* Algorithms in GraphX

GraphX supports sophisticated Graph processing and while you can build your own graph algorithms, GraphX provides a number of algorithms as a part of GraphX directly available as methods of graph or GraphOps objects.

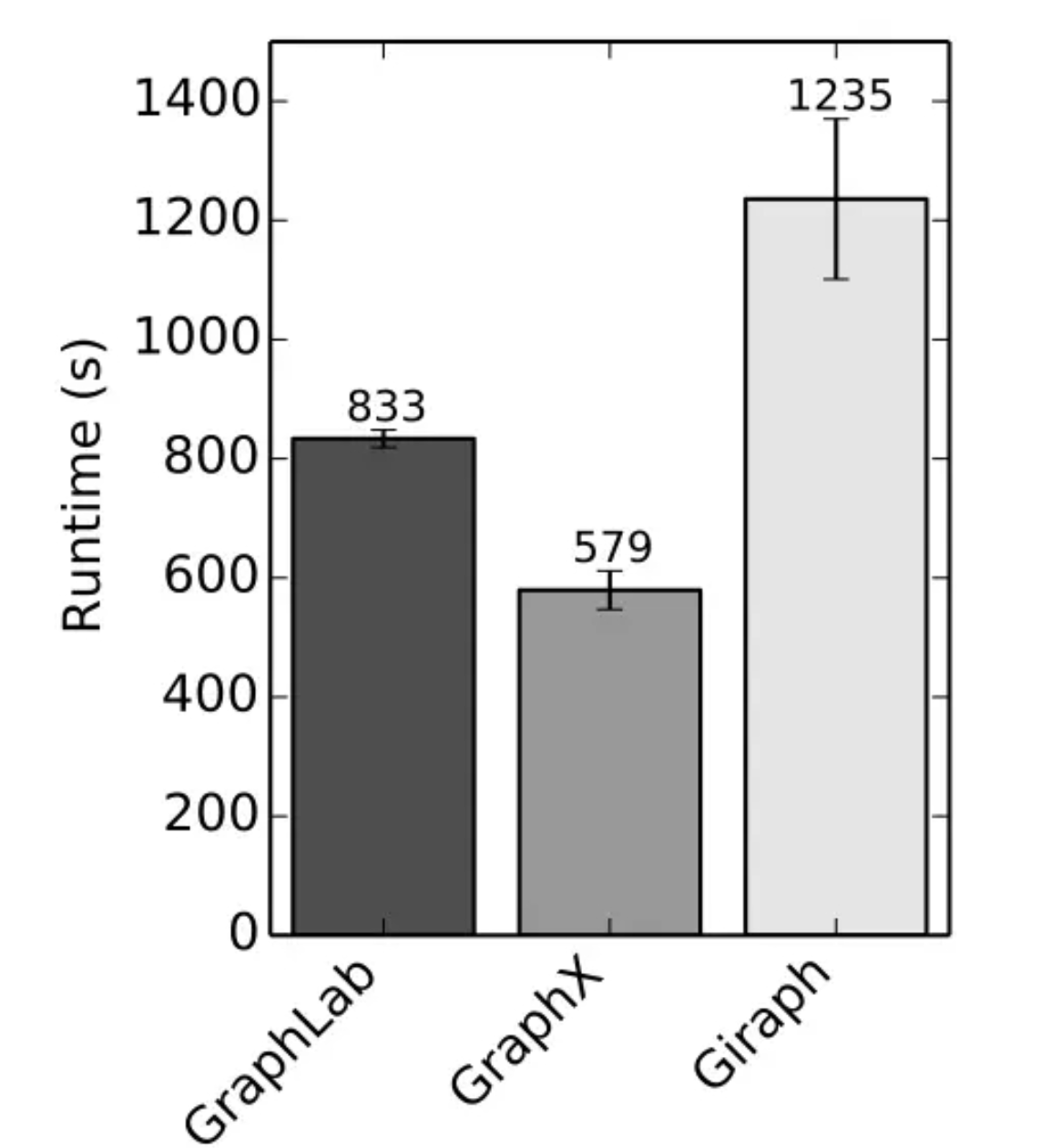


From this part, we can tell that gives more algorithms in machine Learning. GraphX has a shortness that if you want to implement the machine learning algorithms in Graph, you have to code by yourself, which is not convenient for newcomer in Graph computation.

# V. Evaluation

## 5.1 Discussion of GraphX

From the efficiency part, we can tell that GraphX runs more efficiently. The following is the result we can tell from.



But as we did some applications on GraphX and did some comparisons with , there are some shortnesses of GraphX we can list out. Here are the followings:

* It has insufficient algorithms when we handle dataset, especially when we use machine learning method. We have to code by ourselves, which is not convenient.
* When runs larger dataset, it doesn’t show the efficiency when compared with Giraph. (Giraph runs PageRank on the Twitter graph 4.5 times faster than GraphX with 16 workers, and four times faster with eight workers. [3]
* The graph calculation algorithm interface provided is imperfect and needs to be modified in most cases.
* Unlike PowerGraph/PowerLyra, GraphX only has hash-based partitioning schemes. For GraphX, Canonical Random should be used with low-degree graphs and 2D partitioning with power-lawlike graphs.[4]

## 5.2 Discussion of

Comparing to GraphX, has mainly three improvements:

* **Algorithms**: implements MF, BlockPG and LDA to get higher efficiency and computation speed.
* **Heterogeneity**: Supporting heterogeneity is critical to the performance of machine learning algorithms (e.g using BlockPG). A total performance improvement of 40% is over the homogeneous setting.
* **Extensions for Machine Learning**: SSP and mini-batch express a great performance because GraphX does not support mini-batch.

As we can see above, also has disadvantages:

* Although it improves on the algorithm and computational framework, it still needs to rely on distributed systems for task assignment, which means the problems in the graph calculation algorithm interface are not solved.
* In paper, they do not talk about the method of partitioning, they just mention that they use vertex-cut approach. needs partitioning strategies to improve the speed of partitioning to address the data efficiently and quickly.

## 5.3 Ideas to improve

### 5.3.1 For GraphX

* Try to package and share the required algorithms.
* Wait for the version update of GraphX with more built-in algorithms.
* In the distributed system, try to use other partitioning schemes, not just hash-based.

### 5.3.2 For

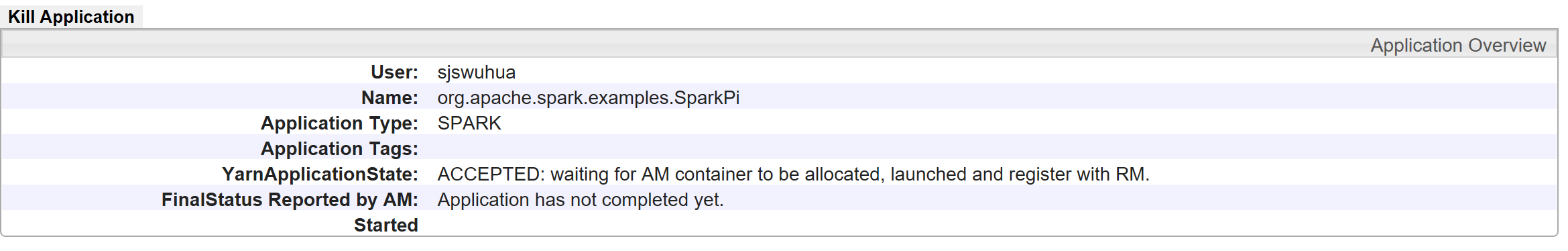
* Find a more suitable task distribution network system to optimize its use environment and to promote it.
* Provide partitioning methods to adapt to more practical situations.
* Improve built-in algorithms and codes to reduce users’ programming workload.

# VI. Challenges and Future Work

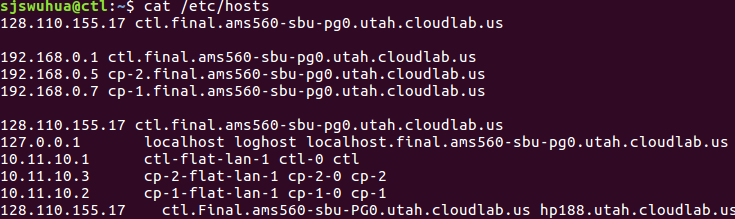
## 6.1 Problems that we have met but have been fixed

### 6.1.1 Yarn didn’t run

At the beginning, we build more than a dozen of clusters to fix with this problems but still failed. The status is always like accepted, and there is no log in Yarn. We stuck here all the time.



But finally, we found out it’s a problem of users address. We should connect each of them on the Yarn, we should modify the host name on Yarn.



Then we can run it.



### 6.1.2 GraphX computation

Since GraphX is related with scala language, we don’t know how to use it to deal with the Netflix Dataset. But after learning online, we changed our goal to PageRank algorithm which is imported in GraphX itself. We use the dataset online, and change the code in the way we want. We finally get our result.

6.2 Problems that still exist

### 6.2.1 Spark runs workload just on one node

When we runs our large workload on spark, it always stuck. This made us think there are some problems in the spark. We just tried to fix with it, but after trying many times, like modifying the memory used or the parameters, it’s just stuck there.

### 6.2.2 GraphX needs to run in distributed system

Our example runs just on one machine. Because our spark has some problems to be fixed. And if we want compare more and do some improvement, we totally need to use it in distributed system.

## 6.3 Future work

At first, we need to fix with the problems with Spark on the top of Yarn. That’s the foundation of running GraphX in distributed system.

Then because we know if we want to optimize a data program，we have to combine it with some math work, like we have to combine a data program with its distribution.

# VII. References

[1] *Tux²: Distributed Graph Computation for Machine Learning*, Xiao et al, NSDI, 2017.

[2] https://spark.apache.org/

[3]https://code.facebook.com/posts/319004238457019/a-comparison-of-state-of-the-art- graph-processing-systems/

[4] Shiv Verma. An Experimental Comparison of Partitioning Strategies in Distributed Graph Processing[J] Proceedings of the VLDB Endowment, Vol. 10, No. 5

[5] Verma S. *An experimental comparison of partitioning strategies in distributed graph processing*[J]. Proceedings of the Vldb Endowment, 2016, 10(5):493-504.

[6] https://www.limfinity.com/ir/

[7] https://www.limfinity.com/ir/kristen\_thesis.pdf